Grammar and Model Extraction for Security Applications using Dynamic Program Binary Analysis

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Juan Caballero Bayerri

Telecommunications Engineer, Universidad Politécnica de Madrid
M.S., Electrical Engineering, KTH Royal Institute of Technology

Carnegie Mellon University
Pittsburgh, PA

September, 2010

This research was supported by the National Science Foundation under Grants No. 0311808, No. 0433540, No. 0448452, No. 0627511, and CCF-0424422, by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170, by the Office of Naval Research under MURI Grant No. N000140911081, by the Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, and by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. The author was also partially supported by a La Caixa Foundation Fellowship.

A mis padres

(Dedicated to my parents)

Abstract

In this thesis we develop techniques for analyzing security-relevant functionality in a program that do not require access to the program's source code, only to its binary form. Such techniques are needed to analyze closed-source programs such as commercial-off-the-shelf applications and malware, which are prevalent in computer systems. Our techniques are dynamic: they extract information from executions of the program. Dynamic techniques are precise because they can examine the exact run-time behavior of the program, without the approximations that static analysis requires.

In particular, we develop dynamic program binary analysis techniques to address three problems: protocol reverse-engineering, binary code reuse, and model extraction. We demonstrate our techniques on a variety of security applications including active botnet infiltration, deviation detection, attack generation, vulnerability-based signature generation, and vulnerability discovery.

Protocol reverse-engineering techniques infer the grammar of undocumented program inputs, such as network protocols and file formats. Such grammars are important for applications like network monitoring, signature generation, or botnet infiltration. When no specification is available, rich information about the protocol or file format can be reversed from a program that implements it. We develop a new approach to protocol reverse-engineering based on dynamic program binary analysis. Our approach reverses the format and semantics of protocol messages by monitoring how an implementation of the protocol processes them. To demonstrate our techniques, we extract the grammar of the previously undocumented C&C protocol used by MegaD, a prevalent spam botnet.

Binary code reuse techniques make a code fragment from a program binary reusable by external source code. We propose a novel approach to automatic binary code reuse that identifies the interface of a binary code fragment and extracts its instructions and data dependencies. The extracted code is self-contained and independent of the rest of the functionality in the program. To demonstrate our techniques, we use them to extract proprietary cryptographic routines used by malware and show how those routines enable infiltrating botnets that use encrypted protocols.

Model extraction techniques build a model of the functionality of a code fragment. Closed-source programs often contain undocumented, yet security-relevant, functionality such as filters or proprietary algorithms. To reason about the security properties of such functionality we develop model extraction techniques that work directly on program binaries. To produce models with high coverage, we extend previous dynamic symbolic execution techniques to programs that use string operations, programs that parse highly structured inputs, and programs that use complex functions like encryption or checksums. We demonstrate the utility of our techniques to discover vulnerabilities in malware and use the extracted models to automatically find subtle content-sniffing XSS attacks on Web applications, to identify deviations between different implementations of the same functionality, and to generate signatures for vulnerabilities in software.

Acknowledgements

Many people encouraged and supported this thesis. Foremost is my advisor, Dawn Song. She gave me the opportunity to join her research group, supported my research interests, and made me grow as a researcher. Her dedication and creativity are admirable, and her passion for research and good humor are contagious.

I am deeply grateful to Vern Paxson for his support and many helpful discussions and feedback, and to David Andersen and Adrian Perrig for serving on my thesis committee and providing insightful comments on this work and its presentation. I would also like to thank Christopher Kruegel and Xuxian Jiang for their support and valuable comments on this work.

Many thanks are in order to all members of the Bitblaze team including Domagoj Babíc, David Brumley, Chia Yuan Cho, Steve Hanna, Ivan Jager, Noah Johnson, Min Gyung Kang, Zhenkai Liang, Stephen McCamant, James Newsome, Pongsin Poosankam, Prateek Saxena, and Heng Yin. Whether it was by co-authoring papers or by providing help, feedback, and discussions, I am indebted to you all. I am also immensely grateful to Adam Barth and Christian Kreibich for co-authoring papers where some of the ideas in this thesis were first published.

Many other colleagues helped with parts of this thesis. Thanks are due to Bryan Parno for his help with this manuscript; Robin Sommer for providing a stand-alone version of BinPac; Rhishikesh Limaye, Susmit Jha, and Sanjit A. Seshia for their help in designing an abstract string syntax; Fabrice Desclaux for his help with the binary code reuse project; Vijay Ganesh and David Dill for their support with STP; Edward Xuejun Wu for their help with malware vulnerabilities; and Rolf Rolles, Weidong Cui, Chris Karlof, and Adrian Mettler for valuable comments on the papers that shaped this thesis.

I am also indebted to Chenxi Wang and Christopher Olston for their help at the beginning of my Ph.D. studies, and to my co-authors Avrim Blum, Theocharis Kampouris, Chris Grier, Shobha Venkataraman, Jia Wang, and Jennifer Yates. Thanks are also due to the La Caixa Foundation and to Angel Álvarez, Gerald Q. “Chip” Maguire Jr., and Jose Luis Fernández-Sánchez, for helping me get into graduate school.

My life in Berkeley and Pittsburgh was brightened by many friends. Without you all, these six years would not have been the same. I would particularly like to thank Min, a great friend who shared much time with me. I would also like to thank the people in Pata Negra and Iberia-Berkeley for building great communities that make us feel at home while away. And, many thanks to all my old friends. One of the toughest parts of graduate school was to be away from you all.

Finally, I am indebted to my family. To my parents for their unwavering support, to my brother Alejandro, a great mentor and confident, and to my lovely sister Macarena. I am fortunate to have your support regardless of where I go.

Contents

List of Tables

List of Figures

Part I

Introduction

Chapter 1

Introduction

1.1 Introduction

In this thesis, we develop dynamic program binary analysis techniques to extract the grammar of undocumented program inputs and to model security-relevant functionality from program binaries. We demonstrate that our techniques enable previously unsolved security applications, such as active botnet infiltration through deep packet inspection and rewriting of encrypted protocols used by malware, and enable more accurate solutions for other important security applications such as detecting deviations between implementations of the same functionality, finding attacks on Web applications, vulnerability discovery, and generating signatures for intrusion detection systems.

The program binary analysis techniques presented in this thesis can be grouped into three main modules of functionality: *protocol reverse-engineering*, *binary code reuse*, and *model extraction*. Protocol reverse-engineering techniques infer the grammar of undocumented program inputs, such as network protocols and file formats. They are necessary because many widely used protocols and file formats, such as the Skype protocol [198] or the communication protocols used by malware, have no publicly available specification. Protocol reverse-engineering is challenging because protocols can be highly structured; can include a variety of elements such as variable-length fields, delimiters, length fields, and arrays; and can carry a variety of data such as timestamps, error codes, filenames, file data, and IP addresses.

Binary code reuse techniques extract a fragment of binary code from a program binary, so that it is self-contained and can be reused by external source code. Binary code reuse is necessary when the code to be reused is only available in binary form. It is specially useful when the code fragment is complex but the application does not require a low level understanding of how the code fragment works; it only requires reusing its functionality. For example, in this thesis we use our binary code reuse techniques for extracting the cryptographic functions and keys used by malware to protect its network protocols. Then, we deploy those functions and keys in a network intrusion detection system (NIDS), enabling the NIDS to decrypt the encrypted traffic. Binary code reuse is challenging because binary code is not designed to be reusable and lacks high-level abstractions such as functions, variables, and types.

Model extraction techniques build a model of the functionality of a code fragment. Extracting a model of security-sensitive code is important because such a model enables automatic reasoning about the security properties of the code. For example, in this thesis we use such models to automatically find subtle attacks on Web applications, to identify deviations between different implementations of the same functionality, and to generate signatures for vulnerabilities in software. Model extraction is necessary because for most programs models of their security-sensitive functionality are not available. Model extraction is challenging because the code to model may be complex and comprise many execution paths. A critical challenge for model extraction techniques is to build high-coverage models that cover many execution paths in the code.

Dynamic program binary analysis for security applications. Our techniques work on program binaries and do not require the availability of source code or any debugging information in the binaries. This provides several benefits. First, our program binary analysis techniques are widely applicable. They can be applied to programs even when the source code is not available, which is important because closed-source programs are prevalent in computer systems and only distributed in binary form. They can also be applied independently of the programming language, programming style, and compiler used to create the program.

Second, our techniques do not require cooperation from the program authors. This is important because some program authors may not support the security analysis of their programs. For example, malware is a large class of closed-source programs where it is important not to rely on the program authors. Another large class of closed-source programs is commercial-off-the-shelf (COTS) applications. Our techniques enable users of COTS applications to analyze the programs they are deploying for security issues. This is necessary because program authors may not be aware of the existence of security issues in their programs, may be reluctant to document security-relevant functionality, may be too slow to react to disclosed security issues, or may not provide security fixes for legacy versions of their programs.

Third, our techniques have high fidelity because they analyze the binary, which is what gets executed. Compared to the program source code, the program binary is a lower abstraction representation of the program. This can be challenging, but it enables the analysis of security issues that may be hidden by the abstractions provided by the programming language such as arithmetic overflows and security issues related to the memory layout of the program.

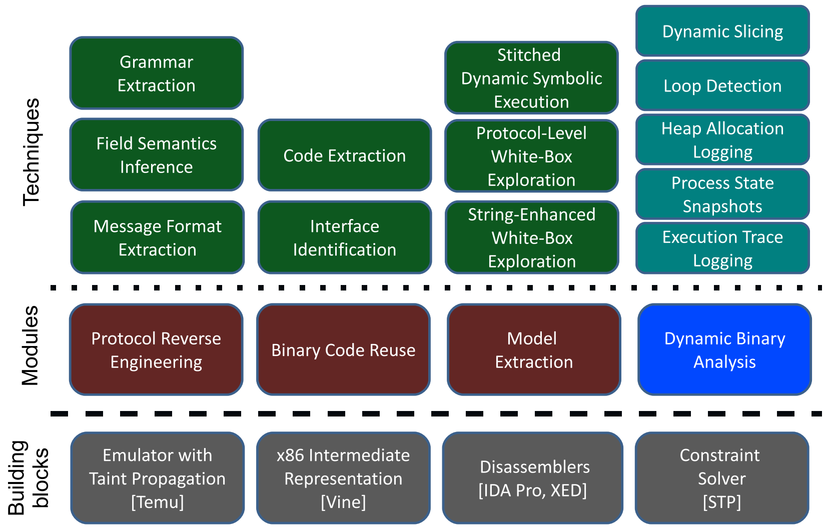

Techniques
Grammar Extraction
Field Semantics Inference
Message Format Extraction
Code Extraction
Interface Identification
Stitched Dynamic Symbolic Execution
Protocol-Level White-Box Exploration
String-Enhanced White-Box Exploration
Dynamic Slicing
Loop Detection
Heap Allocation Logging
Process State Snapshots
Execution Trace Logging
Modules
Protocol Reverse Engineering
Binary Code Reuse
Model Extraction
Dynamic Binary Analysis
Building blocks
Emulator with Taint Propagation [Temu]
x86 Intermediate Representation [Vine]
Disassemblers [IDA Pro, XED]
Constraint Solver [STP]

Figure 1.1: Techniques summary.

Our techniques are dynamic: they are applied over executions of the code. The main advantage of dynamic analysis over static analysis is that dynamic analysis is precise because it can examine the exact run-time behavior of the program, without approximations or abstractions. This property provides several benefits to dynamic analysis: 1) it can analyze programs binaries that are encrypted or packed because code and data will be decrypted or unpacked before being used; 2) there is no control flow uncertainty as the executed instructions are revealed at run-time; 3) pointer aliasing is simple as the accessed memory locations are known at run-time; 4) it can analyze the interactions of the program with other elements in the run-time environment such as the operating system or external libraries. The main limitation of dynamic analysis is limited coverage, as an execution covers only one path in the program. To address this limitation we use white-box exploration techniques that execute many paths in the program. Then, we generalize the results from all the executed paths.

Application and technique summary. This thesis comprises both program binary analysis techniques and their applications. Figure 1.1 summarizes the techniques. It shows between the dashed and dotted lines the four modules of functionality we have developed in this thesis. The three modules on the left: protocol reverse-engineering, binary code reuse, and model extraction comprise the novel techniques proposed in this thesis, while the dynamic binary analysis module on the right comprises previously proposed dynamic binary analysis techniques that we have implemented. On top of each functionality module, above the dotted line, the figure shows the techniques included in the module. The main techniques developed in this thesis comprise the leftmost three columns. The figure shows at the bottom, below the dashed line, some building blocks of functionality, which

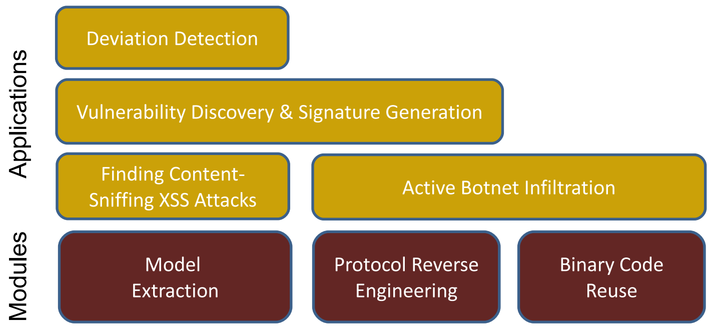

Applications
Deviation Detection
Vulnerability Discovery & Signature Generation
Finding Content-Sniffing XSS Attacks
Active Botnet Infiltration
Modules
Model Extraction
Protocol Reverse Engineering
Binary Code Reuse

Figure 1.2: Applications summary.

were available a priori and are not a contribution of this thesis. Figure 1.2 summarizes the applications covered in this thesis and shows which of the three main modules of functionality that we have developed in this thesis enable them.

The author would like to emphasize the fact that the techniques presented in this thesis are generally applicable to programs in binary form. Our applications target two large classes of binary programs: benign COTS applications and malware. It is important to note that when dealing with malware there exists an additional hurdle in that malware authors are highly motivated to avoid security analysis. Thus, there often exists an arms race between obfuscation techniques being developed by malware authors that take advantage of limitations in state-of-the-art analysis, and new analysis techniques being deployed to fix those limitations. Throughout this thesis, and specially on the two applications dealing with malware in this thesis, namely active botnet infiltration and vulnerability discovery, we discuss potential steps that malware authors may try to defeat them.

1.2 Applications and Techniques

In this section, we introduce the security applications addressed in this thesis and the techniques that we have developed to enable them.

1.2.1 Active Botnet Infiltration

Botnets, large distributed networks of infected computers under the control of an attacker, are one of the dominant threats in the Internet. The number of compromised computers in the Internet belonging to botnets, (i.e., the number of bots) ranges in the millions and a single botnet can grow to over 12 million bots [45]. They produce 85% of the all the spam in the Internet [205] and enable a wide variety of other abusive or fraudulent activities, such as click fraud and distributed denial-of-service attacks.

At the heart of a botnet is its command-and-control (C&C) protocol, used by the botmaster to coordinate malicious activity. Understanding the C&C protocol used by a botnet reveals a wealth of information about the capabilities of its bots and the overall intent of the botnet. In addition, it enables analysts to actively interact with the botnet, by introducing new messages or rewriting existing messages in the C&C channel, what we call *active botnet infiltration*.

Active botnet infiltration can be used to track over time the operations of the botnet and the botmaster (i.e., the attacker that controls the botnet). Some examples of active botnet infiltration are equipping a network intrusion detection system (NIDS) to perform deep packet inspection and rewriting of the encrypted C&C protocol used by a botnet, and building a fake bot that joins the botnet and simulates the network behavior of a real bot, without the nasty side effects (e.g., without sending spam). Once the botnet is infiltrated we can monitor its activities over time.

Problem overview. The main challenges for active botnet infiltration are the lack of a protocol specification for the C&C protocol and the use of encrypted C&C protocols. Imagine that our goal is to enable a network intrusion detection system (NIDS) to perform deep packet inspection and rewriting of the undocumented and encrypted C&C protocol used by a botnet. To rewrite an encrypted message from the botnet's C&C protocol, the NIDS needs to be able to decrypt the message, parse it to extract the underlying field structure, modify some field values, and re-encrypt the message. For this, the NIDS requires access to the cryptographic information and the protocol grammar, which are not commonly available.

In this thesis we propose techniques for inferring the grammar of an undocumented protocol and for extracting the cryptographic information used to protect the protocol. The cryptographic information comprises the decryption and encryption functions, and the session keys, while the protocol grammar captures the information about the different messages in the protocol, the format or field structure of each of those messages, and the semantics of each of the message fields.

Protocol reverse-engineering techniques can be used to infer the protocol specification of unknown or undocumented protocols and file formats. But, current protocol reverse-engineering techniques cannot analyze encrypted protocols and for unencrypted protocols they are manual [210, 148, 72] or take as input network traces, which contain limited protocol information [10, 52].

To address the issue of limited information in network traces, in this thesis we propose a new approach for automatic protocol reverse-engineering that leverages the fact that we often have access to the executable of a program that implements the protocol. Our approach leverages rich information about how the program processes the protocol data, not available in network traces, producing more accurate results than approaches purely based on network traces. Our protocol reverse-engineering techniques extract the message format and the field semantics of messages on both directions of the communication, even when only one endpoint's implementation of the protocol is available. This is of fundamental importance when analyzing a botnet's C&C protocol as often we have access to the bot program but not to the executable of the C&C server.

To analyze encrypted protocols, in this thesis we propose the first approach for automatic binary code reuse, the process of automatically identifying the interface and extracting the instructions and data dependencies of a code fragment from a program binary, so that it is self-contained and can be reused by external source code. Reusing binary code is useful because for many programs, such as commercial-off-the-shelf applications and malware, source code is not available. It is also challenging because binary code is not designed to be reusable even if the source code it has been generated from is. Binary code reuse enables automatic protocol reverse-engineering for encrypted protocols by extracting the cryptographic information needed to access the unencrypted data.

Intuition and approach. The intuition behind our automatic protocol reverse-engineering approach is that in absence of a specification, the richest information about the protocol is available in the programs that implement the protocol and that we can infer the protocol from the implementation by running the program on protocol messages and monitoring how it processes them.

Our automatic protocol reverse-engineering approach is based on dynamic program binary analysis. We execute the program in a monitored environment, feed it protocol messages and analyze how the program processes those input messages and how it builds the output messages that it sends in response. From one execution of the program on a given message, our techniques can extract the message format and infer the field semantics for the input message, as well as the output message that may be sent in response. Thus, our automatic protocol reverse-engineering techniques extract the message format and the field semantics of messages on both directions of the communication, even when only one endpoint's implementation of the protocol is available. The message format captures the field structure of the message while the field semantics capture the type of data in the field such as an IP address, a port number, or a filename.

The intuition behind our binary code reuse approach is that in many circumstances we do not need to understand the low level details of a piece of binary code, we just need to understand its goal and be able to reuse its functionality. For example, if we are interested in the unencrypted data of an encrypted C&C message, we may not need to understand the encryption algorithm used to protect the message, as long as we can reuse the decryption function and decrypt the message.

Our binary code reuse approach comprise three types of techniques: a technique for identifying the entry point of the interesting functionality (e.g., of the decryption function), an interface identification technique to infer a C prototype for the piece of binary code to reuse so that other C programs can interface with it, and a code extraction technique that extracts the instructions and data dependencies in the piece of binary code to reuse so that it can be made stand-alone, without dependencies to the rest of the program's functionality. Our entry point and interface identification techniques use dynamic program binary analysis. Our code extraction technique uses a combination of static and dynamic analysis that includes techniques for hybrid disassembly [156], symbolic execution [106], and jump table identification [39].

We use our entry point identification technique to identify encoding functions, which include functions such as encryption and decryption, compression and decompression, and other types of encoding. Once the encryption and decryption functions are located, we can apply our automatic protocol reverse-engineering techniques on the unencrypted data. For example, we can apply our message format extraction techniques for received messages on the output of the decryption function, rather than on the output of the network receive function, so that the data has already been decrypted by the program. Using our interface identification and code extraction techniques we extract the encryption and decryption functions and the session keys from the binary so that we can give them to a NIDS that can use it to decrypt and rewrite network traffic.

Results. As an end-to-end example of how our automatic protocol reverse-engineering and binary code reuse techniques enable active botnet infiltration, we have analyzed the previously undocumented, encrypted, C&C protocol used by MegaD, a prevalent spam botnet. We extract the C&C protocol grammar and the cryptographic routines used to protect it, and use them for active botnet infiltration, by enabling a NIDS to rewrite a capability report sent by a MegaD bot to make the botmaster believe that the bot can send spam, while all outgoing spam is blocked.

We have evaluated our protocol reverse-engineering techniques on 6 protocols: the C&C protocol used by the MegaD botnet, for which we have no specification, as well as 5 open protocols: DNS, FTP, HTTP, ICQ, and SMB. For the open protocols, we compare our results with the output of Wireshark [227], a popular network protocol analysis tool, which ships with many manually-crafted protocol grammars. Our results show that our message format extraction and field semantics techniques are accurate and produce results comparable to Wireshark. More importantly, our techniques operate without knowledge about the protocol specification, a pre-requisite for Wireshark.

We have evaluated our binary code reuse techniques by extracting the encryption and decryption routines used by two spam botnets, MegaD and Kraken, as well as the MD5 and SHA1 functions from the OpenSSL library [165]. To show that we can reuse code fragments that are not complete functions we also extract the unpacking functions from two samples of Zbot, a trojan, and use an unpacking fragment from one sample as part of the routine to unpack the other sample. Finally, we have applied software-based fault isolation [134] to the extracted code to prevent it from writing or jumping outside their own isolated memory regions.

Contributions.

- **A new approach for automatic protocol reverse-engineering.** We present a new approach for protocol reverse-engineering using dynamic program binary analysis. Our approach leverages the availability of a program binary that implements the protocol and infers the protocol by monitoring how the program processes the input messages and generates the output messages. Compared to previous approaches that take as input network traces, our approach infers more accurate information and can analyze encrypted protocols.
- **Message format extraction techniques.** We propose techniques for extracting the field structure for messages both received and sent by an application. Our techniques identify hard-to-find protocol elements such as length fields, delimiters, variable-length fields, multiple consecutive fixed-length fields, and protocol keywords.
- **Field semantics inference techniques.** We propose techniques for inferring field semantics. Our field semantics inference techniques leverage the rich semantic information available in the publicly available prototype of well-known functions and instructions. Our techniques infer the type of data that fields in the received and sent messages carry such as filenames, IP addresses, timestamps, and error codes.
- **A binary code reuse approach.** We propose the first approach for automatic binary code reuse, which enables automatically identifying the interface and extracting the instructions and data dependencies of a code fragment from an executable program, so that it is self-contained and can be reused by external code.
- **Interface Identification techniques.** We propose techniques to identify the interface of a binary code fragment, without access to its source code. The interface captures the inputs and outputs of the code fragment and enables reuse from external source code.
- **Code extraction technique.** We design a code extraction technique to extract the instructions and data dependencies of a binary code fragment so that it is self-contained and can be reused independently of the rest of the program's functionality. Our code extraction technique uses a combination of static and dynamic analysis and produces code that runs independently of the rest of the program, can be easily instrumented, and can be shared with other users.
- **Enabling active botnet infiltration.** We demonstrate how our protocol reverse-engineering and binary code reuse techniques enable active botnet infiltration by analyzing MegaD, a prevalent spam botnet that uses a previously undocumented, encrypted C&C protocol. We use our techniques to infer the C&C protocol grammar and to extract the cryptographic functions and keys used to protect the communication. We deploy the protocol grammar and the cryptographic information on a NIDS, enabling the NIDS to perform deep packet inspection

and rewriting on the encrypted C&C traffic. We use the modified NIDS to rewrite a capability report sent by the bot to the C&C server to state that the bot is able to send spam, when in reality all spam sent by the bot is being blocked.

1.2.2 Deviation Detection

Different implementations usually exist for the same specification. Due to the abundance of coding errors and specification ambiguities, these implementations usually contain *deviations*, i.e., differences in how they check and process some of their inputs. Automatically finding inputs that demonstrate these deviations is important for two applications: 1) *error detection*, since a deviation may indicate that at least one of the two implementations has an error, and 2) *fingerprint generation*, since an input that triggers a deviation, when given to two different implementations, will result in different output states.

For example, deviation detection is important for testing implementations of network protocols. The Internet Standards process requires that two independent implementations of a protocol from different code bases have been developed and tested for interoperability before advancing a protocol to Draft Standard [18]. Deviation detection can be used for interoperability testing or after the implementations are deployed, since experience shows that even after interoperability testing, differences still exist on how different protocol implementations handle some of the protocol inputs. Deviation detection can help identify errors in the implementations of the specification, as well as areas of the specification that are underspecified. For externally observable deviations, the inputs deviation detection finds can be used by fingerprinting tools like Nmap [162] to remotely identify the implementations.

Problem overview. We propose a novel approach to automatically discover deviations between different implementations of the same specification. We are given two implementations P_1 and P_2 of the same specification and wish to find inputs such that the same input, when given to the two implementations, will cause each implementation to result in a different output state.

Each implementation at a high level can be viewed as a mapping function from the input space I to the output state space S. Let $P_1, P_2 : I \to S$ represent the mapping function of the two implementations from inputs $x \in I$, to output states $s \in S$ resulting from processing the given input. Our goal is to find an input $x \in I$ such that $P_1(x) \neq P_2(x)$. Finding such an input through random testing is usually hard. However, in general it is easy to find an input $x \in I$ such that $P_1(x) = P_2(x) = s \in S$, i.e., most inputs will result in the same output state s for different implementations of the same specification.

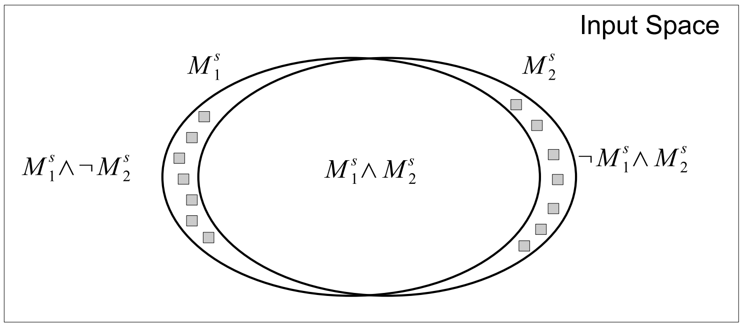


Figure 1.3: Deviations occur when the inputs that produce state s in an implementation P_1 do not exactly match the inputs that produce state s in another implementation P_2.

For example, given two implementations of a Web server, e.g., Apache [4] and MiniWeb [147], implementing the same HTTP protocol specification [65], it is easy to find inputs (e.g., HTTP requests) for which both servers, if configured similarly, will produce the same output state (e.g., an “HTTP 200 OK” response). However, it is not so easy to find deviations, inputs for which both servers will produce different protocol output states such as one server accepting the request and the other rejecting it, or one server accepting the request and the other crashing while processing it. Our approach finds such deviations.

Intuition and approach. The intuition behind our deviation detection approach is that the mapping function $P : I \rightarrow S$ can be seen as a conjunction of boolean models, one for each output state $s \in S$, representing the set of inputs x such that $M_1^s(x) = true \iff P_1(x) = s$. If we have two such models for the same output state s, one for each of two implementations of the same specification, then a deviation is just an input that satisfies the following predicate: $(M_1^s \wedge \neg M_2^s) \vee (\neg M_1^s \wedge M_2^s)$. Such an input is a deviation because it produces an output state s for one of the implementations and another output state $r \neq s$ for the other implementation. Figure 1.3 illustrates this situation. To find deviations, we propose techniques to extract the models M_1^s and M_2^s and then use a decision procedure to query for inputs that satisfy $(M_1^s \wedge \neg M_2^s) \vee (\neg M_1^s \wedge M_2^s)$.

We observe that the above method can still be used even when the extracted model M_P^s is incomplete, i.e., when $M_1^s(x) = true \implies P_1(x) = s$ but the converse is not necessarily true. In this case the model may not cover all possible inputs that reach state s. We have observed that we are able to find deviations, even when our models cover only a single program execution path leading to state s. In that case, the model M_P^s represents the subset of inputs that would follow that particular execution path and still reach the output state s. Thus, $M_P^s(x) = true \Rightarrow P(x) = s$, since if an input satisfies M_P^s then for sure it will make program P go to state s, but the converse is not necessarily true—an input which makes P go to state s may not satisfy M_P^s. In our problem, this means that the difference between M_1^s and M_2^s may not necessarily result in a true deviation. Instead, the difference between M_1^s and M_2^s is a good candidate to trigger a deviation. To verify that the candidate input indeed triggers a deviation, our approach sends such candidate inputs to the two programs and monitors their output states. If the two programs end up in two different output states, then we have successfully found a deviation between the two implementations, and the corresponding input that triggers the deviation.

The output state needs to be externally observable. We use two methods to observe the output state: (a) monitoring the output of the program (e.g., the network traffic), and (b) supervising its environment, which allows us to detect unexpected states such as program halt, reboot, crash, or resource starvation. We may use some domain knowledge about the application to determine when two output states are different. For example, when finding differences between two Web servers we may use the Status-Code in the HTTP response to identify the output state of the Web server after processing a given HTTP request.

Results. We have designed and implemented model extraction techniques that produce models covering a single execution path in the program, and found that such models are surprisingly effective at finding deviations between different implementations of the same functionality.

We have evaluated our approach using 3 HTTP server implementations and 2 NTP server implementations. Our approach successfully identifies deviations between the different server implementations for the same protocol and automatically generates inputs that trigger different server behaviors. These deviations include errors and differences in the interpretation of the protocol specification. For example it finds an HTTP request that is accepted by the MiniWeb Web server with a “HTTP/1.1 200 OK” response, while it is rejected by the Apache Web server with a “HTTP/1.1 400 Bad Request” response. Such deviation is due to an error in the MiniWeb server that fails to verify the value of the first byte in the URL. The evaluation shows that our approach is accurate: in one case, the relevant part of the input that triggers the deviation is only three bits.

Contributions.

- **A new approach for automatic deviation detection:** We propose an approach to automatically discover deviations in the way that two implementations of the same specification process their inputs. Our approach automatically extracts models for each implementation and then queries a solver to obtain inputs that will cause each implementation to result in a different output state. One fundamental advantage of our approach is that it does not require a manually-generated model of the specification, which is often complex, tedious, and error-prone to generate.

- **Single-path model extraction techniques:** We propose dynamic symbolic execution techniques to extract models that cover a single execution path in the program. Our model extraction techniques work directly on program binaries. By automatically building models from an implementation, our models are precisely faithful to the implementation.
- **Enabling error detection and fingerprint generation using deviation detection:** We implement our deviation detection approach and use it to find deviations between multiple implementations of two popular network protocols: HTTP and NTP. We show that such deviations flag implementation errors that need to be fixed, as well as areas of the specification that are underspecified. In addition, the inputs that trigger the deviation can be used as fingerprints to remotely identify the different implementations. Compared to previous approaches, our approach significantly reduces the number of inputs that need to be tested to discover a deviation.

1.2.3 Generating Filtering-Failure Attacks for Web Applications

There exists a broad class of security issues where a filter, intended to block malicious inputs destined for an application, incorrectly models how the application interprets those inputs. A *filtering-failure attack* is an evasion attack where the attacker takes advantage of those differences between the filter's and the application's interpretation of the same input, to bypass the filter and still compromise the application.

One important class of filtering-failure attacks that we investigate in this thesis are *content-sniffing cross-site scripting attacks*, a class of cross-site scripting (XSS) attacks in which the attacker uploads some malicious content to a benign Web site (e.g., a research paper uploaded to a conference management system). The malicious content looks benign to the content filter used by the Web site (e.g., looks like a PostScript document) and is accepted by the Web site but, when the malicious content is accessed by a user (e.g., a reviewer for the conference), it is interpreted as HTML by the user's Web browser (i.e., rather than as a PostScript document). We call such contents *chameleon* documents and show an example in Figure 1.4. Thus, the attacker can run JavaScript, embedded in the malicious content, in the user's Web browser in the context of the site that accepted the content. For the conference management system example, this means that when the reviewer downloads the research paper for evaluation, the research paper can automatically execute JavaScript, that the malicious author embedded in it, on the reviewers' computer. That JavaScript code can submit back to the conference management system a high score review, without the reviewer's knowledge. Thus, an attacker can use a content-sniffing XSS attack to create research papers that review themselves.

Another example of a filtering-failure attack is a network intrusion detection system (NIDS), which deploys a vulnerability signature to protect some unpatched application. If the signature

```
%!PS-Adobe-2.0
%%Creator: <script> ... </script>
%%Title: attack.dvi
```

Figure 1.4: A chameleon PostScript document that Internet Explorer 7 treats as HTML.

incorrectly models which network inputs exploit the vulnerability in the application, then an attacker can potentially construct a network input that is not matched by the NIDS' signature but still exploits the application.

Problem overview. Our goal is to automatically find content-sniffing XSS attacks, which are inputs that if uploaded to a benign Web site that takes external content will be accepted by the filter the Web site runs on all uploaded content. When the user downloads it from the Web site, the user's Web browser will interpret the content as privileged (e.g., *text/html*). Thus, any executable code (e.g., JavaScript) embedded by the attacker in the content will be executed by the user's Web browser in the context of the benign Web site.

For compatibility, every Web browser employs a *content-sniffing algorithm* that takes as input the payload of an HTTP response, the URL of the request, and the response's *Content-Type* header, and produces as output a MIME type for the content in the HTTP response. That MIME type may differ from the one provided by the server in the *Content-Type* header and is used by the Web browser to invoke an application to handle the content such as the image viewer for *image/jpeg* or Adobe's PDF viewer plugin for *application/pdf*. The content-sniffing algorithm is needed because approximately 1% of all HTTP responses either lack a *Content-Type* header or provide an incorrect value in that header [9].

To find content-sniffing XSS attacks, we model the Web site's upload filter as a boolean predicate on an input ($M_{filter}^{accepted}(x)$), which returns true if the input x is considered safe (i.e., accepted) and false if the input is considered dangerous (i.e., rejected). We model the content-sniffing algorithm in the user's Web browser as a deterministic multi-class classifier that takes as input the payload of an HTTP response, the URL of the request, and the response's *Content-Type* header, and produces as output a MIME type. This multi-class classifier can be split into binary classifiers, one per MIME type returned by the content-sniffing algorithm. Each binary classifier is a boolean predicate that returns true if the payload of the HTTP response is considered to belong to that MIME type and false otherwise (e.g., $M_{csa}^{html}(x)$). To find content-sniffing XSS attacks, we only need to model the binary classifiers for MIME types that can contain active content. Thus, we can model the Web browser's content-sniffing algorithm as a binary classifier that returns true if the content is considered *text/html*: $M_{csa}^{html}(x)$. To find a content-sniffing XSS attack we construct the following query: $M_{filter}^{accepted}(x) \wedge M_{csa}^{html}(x)$. If the solver returns an input that satisfies such query, then we have found a content-sniffing XSS attack.

Intuition and approach. The intuition behind our approach is that different contents have different privilege levels in the Web browser with *text/html* having the highest privilege since it can execute script. An attack is possible because the website's upload filter has a different view than the user's Web browser about which content should be considered privileged. This discrepancy often occurs due to a lack of information by the website's developers about the content-sniffing algorithm in the Web browser. In particular for Web browsers such as Internet Explorer and Safari, the content-sniffing algorithm is closed-source and there is little documentation about its inner workings. Thus, filter developers are forced to infer how the content-sniffing algorithm interprets different contents.

To overcome this problem we can extract models of the content-sniffing algorithm directly from the Web browser's binary, without access to its source code. Our model extraction approach builds high-coverage models using white-box exploration techniques, which explore multiple execution paths inside a program by feeding an input to the program, generating a path predicate for the execution of the program on the input using dynamic symbolic execution, querying a solver for an input that traverses a different path, and iterating by sending the new input to the program so another path is explored. We focus the exploration on a fragment of binary code (e.g., the content-sniffing algorithm) rather than the whole program and produce a model that is the disjunction of all the path predicates.

An important characteristic of many security applications, such as the content-sniffing algorithm in a Web browser, an IDS signature matching engine, or a host filter to block inappropriate Web content, is that they rely heavily on string operations. Current white-box exploration techniques [78, 33, 79] are not efficient at dealing with such applications because they contain a large number of loops, potentially unbounded if they depend on the input. Each loop iteration introduces a number of constraints in the path predicate, which can grow very large, creating a huge exploration space. To improve the coverage of the exploration per unit of time, we propose *string-enhanced white-box exploration*, an extension to current white-box exploration techniques that increases the coverage for programs that heavily use strings operations by reasoning directly about strings, rather than individual bytes that form the strings.

In a nutshell, our string-enhanced white-box exploration comprises four steps. First, it replaces constraints generated inside string functions with constraints on the output of those string functions. Then, the constraints on the output of the string functions are translated into abstract string operators. Next, it translates the abstract string operators into a representation that is understood by an off-the-shelf solver that supports a theory of arrays and integers. Finally, it uses the answer of the solver to build an input that starts a new iteration of the exploration.

Results. We generate content-sniffing XSS attacks by extracting high-coverage models of four applications: the content-sniffing algorithms used by two popular Web browsers, Internet Explorer 7 and Safari 3.1, and the upload filters used by two popular Web applications: MediaWiki [135], an open-source wiki application used by many sites including Wikipedia [224], and HotCRP [87], an open-source conference management application. We extract the model of both upload filters using manual analysis of the source code. For Internet Explorer 7 and Safari 3.1, which use closed-source content-sniffing algorithms, we extract the model using our string-enhanced white-box exploration technique on the Web browsers' binaries.

Using these models we are able to identify previously unknown content-sniffing XSS attacks affecting both MediaWiki and HotCRP. For MediaWiki, we find that there exists at least 6 different MIME types for which an attacker could build a *chameleon* document that will be accepted by MediaWiki as being from a safe MIME type, but interpreted as HTML by Internet Explorer 7. Similarly, there exists 6 MIME types for which content-sniffing XSS attacks are possible if the user downloads the content using Safari 3.1. We have disclosed these issues to MediaWiki's developers, which have confirmed and fixed them. Furthermore, we find that the attacks are due to the use by MediaWiki's upload filter of the MIME detection functions of PHP, which means that other sites that also use PHP's MIME detection functions in their upload filter could also be affected. For HotCRP, we find that an attacker could build chameleon PostScript or PDF documents that would be accepted by the HotCRP Web application and interpreted as HTML by Internet Explorer 7. Thus, an attacker could create a research paper that reviews itself.

Contributions.

- **Multi-path model extraction using string-enhanced white-box exploration:** We propose a new approach to generate high-coverage models for a fragment of binary code. Our approach uses string-enhanced white-box exploration, an extension to white-box exploration that significantly increases the coverage that the exploration achieves per unit of time on programs that heavily use string operations. The coverage increase is obtained by reasoning directly about the string operations performed by the program, rather than the byte-level operations that comprise them.

- **Enabling finding content-sniffing XSS attacks:** We implement our multi-path model extraction technique and use it to extract models of the closed-source content-sniffing algorithms for two popular Web browsers: Internet Explorer 7 and Safari 3.1. Using those models we perform the first systematic study of content-sniffing XSS attacks and find previously unknown attacks that affect two popular Web applications: MediaWiki and HotCRP.

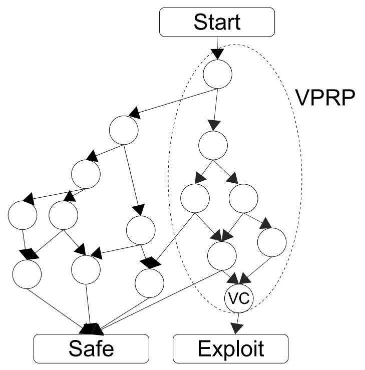


Figure 1.5: The vulnerability point reachability predicate (VPRP) captures the inputs that reach the vulnerability point. A vulnerability-based signature is the conjunction of the VPRP and the vulnerability condition (VC).

1.2.4 Protocol-Level Vulnerability-Based Signature Generation

Software vulnerabilities are prevalent with over 4,500 new publicly disclosed vulnerabilities in 2009 [205, 56]. One popular defense mechanism for software vulnerabilities, widely deployed in intrusion protection and detection systems, is *signature-based input filtering*, which matches program inputs against a set of signatures, flagging matched inputs as attacks.

Compared to software patches, signature-based input filtering can respond to attacks faster, within minutes instead of the hours, days, or even years it can take to generate and test a software patch. For example, Symantec reports that 14% of all new vulnerabilities in Web browsers found in 2009, and 18% of all new vulnerabilities in Web browsers found in 2008, remain unpatched as of April 2010 [205]. Moreover, for legacy systems where patches are no longer provided by the manufacturer, or critical systems where any changes to the code might require a lengthy re-certification process, signature-based input filtering is often the only practical solution to protect the vulnerable program.

Problem overview. Many different approaches for signature-based input filtering have been proposed. Particularly attractive are *vulnerability-based signatures* [218, 47, 21], which are based on the properties of the vulnerability, rather than on the properties of the exploits for the vulnerability. This is important because many different inputs may exploit a vulnerability and there exists tools like Metasploit [137] that can automatically generate exploit variants for a vulnerability.

A vulnerability is a point in a program where execution might "go wrong". We call this point the *vulnerability point*. A vulnerability is only exploited when a certain condition, the *vulnerability condition* (VC), holds on the program state when the vulnerability point is reached. Thus, for an input to exploit a vulnerability, it needs to satisfy two conditions: (1) it needs to lead the program execution to reach the vulnerability point, and (2) the program state needs to satisfy the vulnerability condition at the vulnerability point. We call the condition that denotes whether an input message will make the program execution reach the vulnerability point the *vulnerability point reachability predicate* (VPRP). Figure 1.5 illustrates the VPRP and VC for a vulnerability. Thus, the problem of automatically generating a vulnerability-based signature can be decomposed into two: identifying the vulnerability condition and identifying the vulnerability point reachability predicate. A vulnerability-based signature is simply the conjunction of the two.

We call a vulnerability-based signature that captures all inputs that would exploit the vulnerability *complete*. Such a signature produces zero false negatives, i.e., it matches all exploits for the vulnerability. We call a vulnerability-based signature that only matches inputs that exploit the vulnerability *sound*. Such a signature produces zero false positives, i.e., it does not match benign inputs. A *perfect* vulnerability-based signature is a signature that is both complete and sound. The goal of a vulnerability-based signature generation method is to produce perfect signatures. In addition to producing perfect signatures, given the rate at which vulnerabilities are discovered and that manual signature generation is slow and error-prone, vulnerability-based signature generation methods also need to be automated.

Vulnerability-based signature generation methods work by monitoring the program execution and analyzing the actual conditions needed to exploit the vulnerability and can guarantee a zero false positive rate [47, 21, 46]. Early approaches are limited in that they only capture a single path to the vulnerability point (i.e., their *vulnerability point reachability predicate* contains only one path). However, the number of paths leading to the vulnerability point can be very large, sometimes infinite. Thus, such signatures are easy to evade by an attacker with small modifications of the original exploit message, such as changing the size of variable-length fields, changing the ordering of the fields (e.g., permuting the order of the HTTP headers), or changing field values that drive the program through a different path to the vulnerability point.

Recognizing the importance of enhancing the coverage of vulnerability-based signatures, recent work tries to incorporate multiple paths into the vulnerability point reachability predicate either by static analysis [23], or by black-box probing [46,55]. However, due to the challenge of precise static analysis on binaries, the vulnerability point reachability predicates generated using static analysis are too big [23]. And, black-box probing techniques [46,55], which perturb the original exploit input using heuristics such as duplicating or removing parts of the input message or sampling certain field values to try to discover new paths leading to the vulnerability point, are highly inefficient and limited in their exploration. Hence, they generate vulnerability point reachability predicates with low coverage.

Thus, a key open problem for generating vulnerability-based signatures is how to generate vulnerability point reachability predicates with high coverage and in a compact form. In this thesis, we propose *protocol-level constraint-guided exploration*, a new approach to generate high-coverage, yet compact, VPRPs, which capture many paths to the vulnerability point.

Intuition and approach. To create high-coverage VPRPs, our protocol-level constraint-guided exploration approach leverages white-box exploration. But, programs that parse complex, highly-structured inputs are challenging for white-box exploration techniques because the parsing introduces a large number of execution paths. Thus, the exploration spends an enormous amount of time exploring those paths, and it does not effectively explore the program state after the parsing has finished. To address this problem, previous work proposed a compositional approach that generates higher-level path predicates that work on symbolic grammar tokens returned by the parser, instead of symbolic input bytes [77].

Such approach works well with programs where the parsing of the inputs is well-separated from the remaining functionality such as compilers and interpreters. But, programs that parse network and file specifications often do not have well-separated parsing and processing stages. The intuition behind our protocol-level constraint-guided exploration approach is that when such clean separation does not exist, the protocol specification (publicly available or extracted using protocol reverse-engineering techniques) can be used to identify the constraints introduced by the parsing process, i.e., the *parsing constraints*. The parsing constraints can be removed from the path predicate, introducing field symbols as the output of those parsing constraints, so that the generated path predicates operate on field symbols, rather than byte symbols.

In addition to the reduction in the number of paths to be explored, lifting the byte-level path predicates to field-level path predicates solves another important problem for vulnerability-based signatures. Path predicates that operate on byte symbols match only inputs that have exactly the same format than the input used to collect the path predicate. Without lifting the byte-level path predicates to field-level path predicates, the resulting signature would be very easy to evade by an attacker by applying small variations to the format of the exploit message. For example, without lifting the path predicates, the resulting VPRP would not match exploit variants where the variable-length fields have a different size.

In addition to using the protocol information, our approach also merges execution paths to further reduce the exploration space. As the exploration progresses, new discovered paths that reach the vulnerability point need to be added to the VPRP. A simple disjunction (i.e., an enumeration) of all paths leading to the vulnerability point would introduce many duplicated constraints into the VPRP, where every duplicated constraint effectively doubles the exploration space. Thus, the exploration space could increase exponentially. To avoid this we construct an exploration graph as the exploration progresses and use it to identify potential merging points. As a side benefit, merging paths also reduces the size of the resulting VPRP.

Results. We have used our approach to generate signatures for 6 vulnerabilities on real-world programs. The vulnerable programs take as input file formats as well as network protocols, run on multiple operating systems, and encompass both open-source and closed programs. Our approach is successful at removing the parsing constraints. In the four vulnerable programs that include variable-length strings, the removed parsing constraints accounted for 92.4% to 99.8% of all constraints in the byte-level path predicates.

The generated signatures achieve perfect or close-to-perfect results in terms of coverage. Using a 6 hour time limit for the exploration, our approach discovered all possible paths to the vulnerability point for 4 out of 6 vulnerabilities, thus generating a complete VPRP. For those four signatures, the generation time ranges from under one minute to 23 minutes. In addition, the number of constraints in the resulting VPRP is in most cases small. The small number of constraints in the VPRP and the fact that in many cases those constraints are small themselves, makes most of the signatures easy for humans to analyze.

Contributions.

- **Extracting vulnerability point reachability predicates using protocol-level constraint-guided exploration:** We propose a new approach for automatically generating vulnerability point reachability predicates that capture many paths to the vulnerability point. Our approach uses protocol-level constraint-guided exploration, a technique that leverages the availability of a protocol specification to lift byte-level path predicates to field-level path predicates. In addition, it merges program paths to avoid an explosion in the exploration space. Our approach significantly reduces the number of paths that need to be explored and enables the generation of harder to evade signatures.
- **Enabling high-coverage vulnerability-based signature generation:** We implement our protocol-level constraint-guided exploration technique and use it to generate vulnerability-based signatures for 6 real-world vulnerabilities. Our signatures have high coverage, achieving perfect coverage in 4 out of 6 cases. The increase in coverage translates into signatures that are more difficult to evade by attackers. In addition, the generated signatures are often small and can be analyzed by humans more easily than the signatures generated by previous approaches.

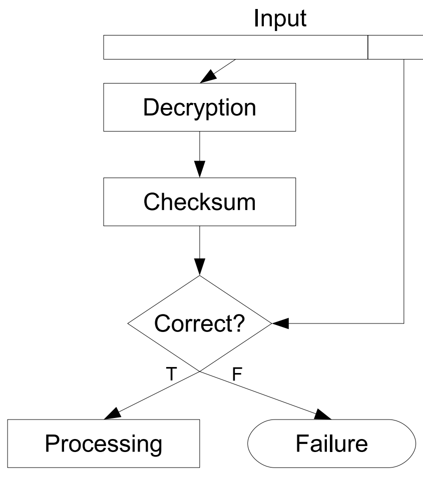


Figure 1.6: Common input processing. Current input generation techniques have trouble creating inputs that reach the processing stage.

1.2.5 Finding Bugs in Malware

Given the prevalence of software defects, vulnerability discovery has become a fundamental security task. It identifies software bugs that may be remotely exploitable and creates program inputs that demonstrate their existence. So far, vulnerability discovery has focused on *benign* programs and little research has addressed vulnerabilities in *malware* programs that the attacker installs in compromised computers. However, malware vulnerabilities have a great potential for different applications such as malware removal, malware genealogy, as a capability for law enforcement agencies, or as a strategic resource in state-to-state cyberwarfare.

For example, some malware programs such as botnet clients are deployed at a scale that rivals popular benign applications: the recently-disabled Mariposa botnet was sending messages from more than 12 million unique IP addresses at the point it was taken down, and stole data from more than 800,000 users [45]. A vulnerability in a botnet client could potentially be used to notify the user of infection, to terminate the bot, or even to clean the infected host. However, some of the potential applications of malware vulnerabilities raise ethical and legal concerns that need to be addressed by the community. While our goal in this research is demonstrating that finding vulnerabilities in widely-deployed malware such as botnet clients is technically feasible, we also hope that this work helps in raising awareness and spurring discussion in the community about the positives and negatives of the different uses of malware vulnerabilities.

Problem overview. Dynamic symbolic execution techniques [106] have recently been used for automatic generation of inputs that explore the execution space of a program for a variety applications such as vulnerability discovery [78, 33, 79] and automatic exploit generation [22, 104]. However, traditional dynamic symbolic execution (as well as other input generation techniques such as mutation fuzzing [63, 145] or taint-directed fuzzing [74]) is ineffective in the presence of certain operations such as the decryption and decompression of data, and the computation of checksums and hash functions; we call these *encoding functions*. Encoding functions can generate symbolic models that are difficult to solve, which is not surprising, given that some of these functions, e.g., cryptographic hash functions, are designed to be infeasible to invert.

Encoding functions are used widely in malware as well as benign applications. For example, malware will often decrypt the received network traffic, decompress its contents, and verify the integrity of the data using a checksum. This process is illustrated in Figure 1.6. In dynamic symbolic execution, those operations are expanded into a complex combination of constraints that mix together the influence of many input values and are hard to reason about [59]. The solver cannot easily recognize the high-level structure of the computation, such as that the internals of the decryption and checksum functions are independent of the message parsing and processing that follows. To address the challenges posed by the encoding functions, we propose *stitched dynamic symbolic execution*, a new approach to perform input generation in the presence of hard-to-reason operations.

Intuition and approach. The intuition behind our stitched dynamic symbolic execution approach is that it is possible to avoid the problems caused by encoding functions, by identifying and bypassing them to concentrate on the rest of the program execution. Stitched dynamic symbolic execution first decomposes the symbolic constraints from the execution, separating the (hard) constraints generated by each encoding function from the (easier) constraints in the rest of the execution. The solver does not attempt to solve the constraints introduced by the encoding functions. It solves the constraints from the remainder of the execution and then re-stitches the solver's output, using the encoding functions (e.g., computing the checksum for the partial input) or their inverses (e.g., encrypting the partial input), into a complete program input.

In a nutshell, our approach proceeds in two phases. As a first phase, it identifies encoding functions and their inverses (if applicable). For identifying encoding functions, we perform a type of dynamic taint analysis that detects functions that highly *mix* their input, i.e., where an output byte depends on many input bytes. The intuition is that high mixing is what makes constraints difficult to solve. To identify the inverses, we use the intuition that encoding functions and their inverses are often used in concert, so their implementations can often be found in the same binaries or in widely-available libraries (e.g., OpenSSL [165] or zlib [240]). We propose a technique that given an encoding function tests whether its inverse is present in a set of functions.

Then in the second phase, our approach augments traditional white-box exploration by adding decomposition and re-stitching. On each iteration of exploration, we decompose the generated constraints to separate those related to encoding functions, and pass the constraints unrelated to encoding functions to a solver. The constraint solution represents a partial input; the approach then re-stitches it, with concrete execution of encoding functions and their inverses, into a complete input used for a future iteration of exploration.

Results. We have implemented our approach and have used it to perform the first automated study of vulnerabilities in malware. Our approach finds 6 bugs in 4 prevalent malware families that include botnet clients (Cutwail, Gheg, and MegaD) and trojans (Zbot). A remote network attacker can use these bugs to terminate or subvert the malware. We demonstrate that at least one of the bugs can be exploited to take over the compromised host. We also find an input that cleanly exits a MegaD bot. Such *kill* commands may not be bugs but can still be used to disable the malware. They are specially interesting because their use could raise fewer ethical and legal questions than the use of an exploit would.

To confirm the value of our approach, we show that traditional dynamic symbolic execution is unable to find most of the bugs we report without our decomposition and re-stitching techniques, and that it takes significantly longer for those it finds. In addition, we use an array of variants from each of the malware families to test how prevalent over time the vulnerabilities are. The bugs reproduce across all tested variants. Thus, the bugs have persisted across malware revisions for months, and even years. These results are important because they demonstrate that there are components in bot software, such as the encryption functions and the C&C parsing code that tend to evolve slowly over time and thus could be used to identify the family to which an unknown binary belongs.

Contributions.

- **A new approach for input generation in the presence of hard-to-reason operations:** We propose stitched dynamic symbolic execution, a new approach to enable input generation in the presence of encoding functions that introduce hard-to-solve constraints. We describe techniques to identify the encoding functions and their inverses, to decompose a symbolic execution separating the hard constraints from the encoding functions from the easier constraints from the rest of the execution, and to re-stitch the partial input output by the solver into a complete program input.
- **Enabling finding bugs in malware:** We implement our approach and use it to perform the first automated study of vulnerabilities in malware. We find 6 bugs in 4 prevalent malware families that a remote attacker can use to terminate or subvert the malware. At least one of the bugs can be exploited to take over the compromised host. We show that traditional dynamic symbolic execution is unable to find most of these bugs and that it takes significantly longer

for those it finds. In addition, we show that the bugs persist across malware revisions for months, and even years.

1.3 Thesis Outline

This thesis is organized in five parts. Part I introduces the problems addressed in this thesis. It comprises this introductory chapter as well as Chapter 2, which presents background information on dynamic program binary analysis and the blocks that this thesis builds upon. Our techniques are described in three parts that correspond to the main modules of functionality we have developed. In Part II, we describe our protocol reverse-engineering techniques, which comprise techniques for message format extraction, field semantics inference, and grammar extraction. Then, in Part III, we describe our binary code reuse techniques, which comprise techniques for interface identification and code extraction.

Next, in Part IV, we describe our model extraction techniques. This part comprises four chapters. Chapter 5 presents single-path model extraction techniques for deviation detection. Chapter 6 describes string-enhanced white-box exploration, an extension to previous white-box exploration techniques for extracting models of programs that use string operations, and applies those multi-path model to finding content-sniffing XSS attacks. Chapter 7 details protocol-level constraint-guided exploration, an extension to white-box exploration for extracting models of programs that parse highly structured inputs, and applies it to generating high-coverage vulnerability point reachability predicates. Chapter 8 presents stitched dynamic symbolic execution, an approach that enables dynamic symbolic execution in the presence of complex encoding functions such as encryption and hashing, and applies it for finding bugs in real-world malware that uses such encoding functions.

We finalize in Part V where we provide further discussion and our concluding remarks.

Chapter 2

Dynamic Program Binary Analysis

2.1 Introduction

In this thesis we develop dynamic program binary analysis techniques. Dynamic program analysis is the process of automatically analyzing the behavior of a program from its execution on a set of inputs, called test cases. Dynamic program analysis uses white-box techniques that monitor the program's internals, as opposed to black-box techniques that also run the program on inputs but focus on input-output relationships without any knowledge of the program's inner workings. Dynamic program analysis leverages the program's internals to achieve finer-grained and more accurate analysis than black-box techniques.

Program analysis requires access to the program in either source code or binary form. In this thesis, we focus in the common scenario where the program source code is not available but the program binary is. Malware and commercial-off-the-shelf (COTS) programs are two large classes of programs where an external analyst only has access to the program binary. In addition to not requiring access to the program's source code, other benefits of using the program binary are that the analysis is independent of the programming language, programming style, and compiler used to create the program; it has high fidelity because the program binary is what gets executed; and it does not require cooperation from the program authors.

Binary analysis. Binary code is different than source code. Thus, we need to adapt and develop program analysis techniques and tools that are suitable for binary code. One challenging difference is that binary code lacks high-level abstractions present in source code such as:

- Functions. The source-level concept of a function is not directly reflected at the binary code level, since functions at the source level can be inlined, split into non-contiguous binary code fragments, or can exit using jumps instead of return instructions (e.g., due to tail-call optimizations).

- Variables. There are no variables at the binary level. There are only registers and memory. Some limited size information can be obtained from the length of the instructions operands (8, 16, 32, 64 bit) but there is no explicit information about larger structures.
- Types. Data at the binary level has no explicit type information. At the binary level type information needs to be inferred from external information. For example, strings do not exist at the binary level. Furthermore, there is no concept of a source-level buffer at the binary level. A buffer at the binary level is just a contiguous sequence of bytes in memory and the only real boundaries in memory are pages. This makes concepts such as buffer overflows hard to define without knowledge about higher-level semantics given by the source code.

Dynamic analysis. The main advantage of dynamic program binary analysis over static program binary analysis (dynamic analysis and static analysis, for short) is that dynamic analysis can examine the exact run-time behavior of the program, without approximations or abstractions. Dynamic analysis has access to the executed instructions as well as the content of the instruction's operands. Thus, there is no control flow uncertainty even if the program is packed, encrypted or heavily uses indirection, and there is no uncertainty on which memory addresses are accessed (i.e., no memory aliasing). In addition, dynamic analysis can analyze the interactions of the program with the operating system and external libraries.

The main limitation of dynamic analysis is that an execution covers only one path in the program. To address this limitation the program must be executed on a variety of test cases that make the program exhibit its full behavior. The extent of the program's behavior exhibited on the test cases over the program's set of possible behaviors is called *coverage*. Coverage can be measured using different metrics such as the percentage of all instruction or paths executed [11]. In Section 2.4 we introduce white-box exploration techniques, which automatically generate test cases that increase the coverage of the program.

This thesis deals predominantly with dynamic analysis. However, dynamic and static analysis have properties that complement each other. Static analysis can give more complete results as it covers different execution paths and dynamic analysis can address many of the challenges of static analysis such as pointer aliasing or indirect jumps. Thus, at times we combine them together to leverage the benefits of both. We expect that the combination of dynamic and static analysis will become prevalent in the near future.

Offline analysis. Dynamic analysis techniques can work online, as the program executes, or offline, on information logs collected during execution. All the novel dynamic binary analysis techniques proposed in this thesis work offline. We present the different information logs we collect during program execution in Section 2.3. The most important of these execution logs are *execution traces*, which capture instruction-level information. Offline dynamic analysis has the benefit that all needed information is recorded during execution. This enables the analysis to be rerun many times over the same execution logs without new non-deterministic behavior being introduced in each execution. In addition, execution logs can be easily shared with other analysts. The main disadvantages compared to online analysis is that the logs collected during program execution can grow very large and that their collection can significantly slow down execution.

Architecture. Although we design our dynamic program binary analysis techniques to be as general as possible, the focus on this thesis is on the x86 (32-bit) architecture, by far the most prevalent architecture for personal computers (PCs), with over one billion installed PCs worldwide [196] and over a million new ones being shipped each day [51]. Our techniques work on x86 binaries and do not require the availability of source code or any debugging information in the binaries.

The remainder of this chapter is structured as follows. Section 2.2 presents the previously-available building blocks used in this thesis. Section 2.3 presents the execution logs gathered during program execution, which are the input to the offline dynamic analysis techniques. Finally, Section 2.4 provides background information on previously-proposed dynamic program binary analysis techniques that our techniques build on.

2.2 Building Blocks

In this section we provide background information on the four previously-available building blocks, shown at the bottom of Figure 1.1, that this thesis builds on: an emulator with taint propagation capabilities (TEMU) [238], an static analysis platform that provides a translation from x86 instructions to an intermediate representation (Vine) [19], a set of static disassemblers [230, 90], and a constraint solver [73].

TEMU. TEMU is a dynamic analysis platform that enables user-defined, instruction-level execution monitoring and instrumentation [238]. TEMU is implemented on top of the QEMU open-source whole-system emulator and virtualizer [180]. As a whole-system emulator, QEMU uses dynamic translation to run an entire guest system, including the operating system and applications, made for one architecture (e.g. ARM) on a different host architecture (e.g. x86). As a virtualizer, QEMU can

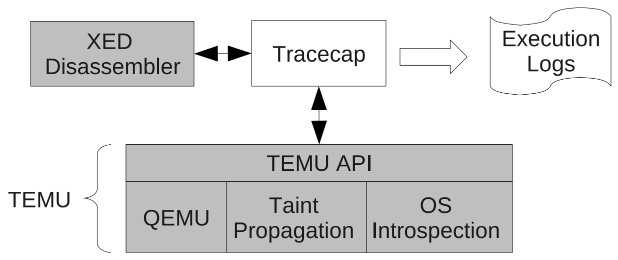


Figure 2.1: Architecture of the execution monitor used to generate the execution logs. The modules in gray were previously available.

be used to run an unmodified guest operating system (e.g., Windows) inside another host operating system (e.g., Linux) providing good isolation between the host and the guest systems.

TEMU provides three main components on top of QEMU. First, it provides an introspection module for extracting information from the operating system of the guest system such as the process and thread identifiers for the current instruction, the base address and size of a module that has been loaded in memory (i.e., an executable or dynamically linked library), and the list of processes running in the guest system. Currently, TEMU can extract information from Windows 2000, Windows XP, and Linux operating systems. Second, TEMU provides a module that implements taint propagation [38,47,49,160,204]. Taint propagation is a data-flow technique where data from a taint source (e.g, network data, a keystroke, or a file) is marked with additional taint information that is propagated along with the data, as the data moves through the system (e.g., into memory, registers, or the file system). TEMU's taint propagation module supports taint across different processes as well as tainting across memory swapping and the file system. Third, TEMU provides a clean interface (API) for user-defined activities. Using this API, users can write custom *plugins* that monitor or instrument the execution including the kernel as well as the different user-level processes. The API provides the plugin with important functionality for monitoring and instrumenting the execution such as reading or writing the system's registers and memory, querying the operating system information provided by the introspection module, saving and restoring the state of the emulated system, defining new taint sources, and querying or setting the taint information of memory and registers. It also provides callback functions for different system events such as a new block or instruction being executed, a new process being created, or a process loading a module in its address space.

The plugin architecture makes TEMU an extensible platform for monitoring and instrumenting the execution of a system. In this thesis we develop a plugin called *Tracecap* that enables saving detailed information about the execution for offline analysis. The execution logs produced by Tracecap include execution traces with instruction-level information, process state snapshots taken at a some point in the execution, and logs of the heap allocations requested by a process. We use the

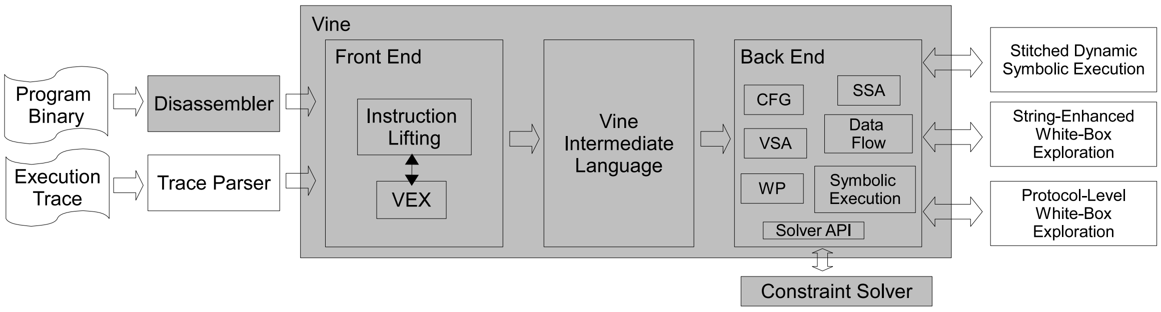


Figure 2.2: Vine architecture. Gray modules were previously available.

term *execution monitor* throughout this thesis to refer to the combination of TEMU and Tracecap. Figure 2.1 presents the architecture of the execution monitor. We describe Tracecap's functionality in Section 2.3. A more detailed description of TEMU is available at [199].

Vine. Vine is a static analysis platform that can translate assembly to a formally specified intermediate representation (IR), called the Vine intermediate language, and provides a set of common static analysis techniques that operate on the Vine intermediate language such as creating control flow graphs (CFGs) [151], transforming statements from the Vine language into Single Static Assignment (SSA) form [151], symbolic execution [106], value set analysis (VSA) [8], computing weakest pre-conditions [61], and data-flow analysis [151]. Figure 2.2 shows the Vine architecture. In this thesis, our main use of Vine is as a symbolic execution engine for our string-enhanced and protocol-level white-box exploration techniques, which are part of our model extraction module. We describe symbolic execution in Section 2.4.2 and refer the reader to [199] for a more detailed description of Vine.

Disassemblers. A disassembler is a program that translates machine code into assembly language. In this thesis, we use an external disassembler for two different purposes. Tracecap uses a disassembler while logging the execution trace to identify the list of operands of an instruction and associated information such as the operand length and the type of access to the operand (e.g., read or write). Currently, Tracecap uses the XED disassembler [230]. In addition, our binary code reuse module uses the commercial IDA Pro disassembler [90] to disassemble code sections in the process state snapshots that Tracecap can generate during the execution of a process.

Constraint solver. In this thesis the constraint solver is a decision procedure, which performs reasoning on symbolic expressions. We use the solver in our model extraction module to determine if a symbolic expression is satisfiable and to generate a counterexample if it is not. We interface with the solver through Vine, which contains a transformation from Vine expressions into a common language understood by different solvers, which makes it convenient to benefit from any advances on decision procedures. Currently, our constraint solver is STP [73], a complete decision procedure incorporating the theories of arrays and bit-vectors.

2.3 Execution Logs

In this section we present the information we collect during program execution. This information enables our offline dynamic analysis techniques. We use two main online components: the taint propagation module included in TEMU, and *Tracecap*, a TEMU plugin that we have developed to collect execution logs. Next, we provide some background information on taint propagation and describe the different execution logs that Tracecap can collect.

2.3.1 Taint Propagation

Taint propagation is a data-flow technique where data from a taint source is marked with additional taint information that is propagated along with the data, as the data moves through the system [38, 47, 49, 160, 204]. Using taint propagation usually comprises three steps: defining the taint sources, propagating the taint information, and checking the taint information at the taint sinks. The taint propagation module in TEMU provides functionality to define taint sources and takes care of the taint propagation as the program executes. Defining taint sinks is left up to plugin.

Using the TEMU API, a plugin can introduce taint at sources such as the keyboard, network interface, and the file system; as well as directly taint memory or registers during the execution. TEMU implements a whole-system taint propagation module in which taint is propagated through kernel and across different user-level processes, even if the tainted data is swapped out of memory or written to a file. TEMU uses a shadow memory to store the taint information for each byte of physical memory, registers, the hard disk and the network interface buffer. More details about TEMU's taint propagation module can be found in [199].

Each byte in a taint source is assigned a unique taint identifier. For convenience, the taint identifier comprises three parts: a taint source identifier, a taint origin, and a taint offset. The taint source identifier is coarse and captures the high level source of the taint such as the network, the keyboard, the file system, or memory. The taint origin provides finer-granularity and is used to group together related bytes in the taint source. For example, if the taint source is the network each TCP flow is assigned a different origin, and if the taint source is the file system each file is assigned a different origin. Finally, the taint offset captures the position of the byte in the taint stream identified by the pair of the taint source identifier and the taint origin. For example, offset

```
78061321:  mov  (%eax),%al   // (%eax) tainted, %al untainted
78061323:  cmp  $0x48,%al    // %al tainted
78061325:  jne  0x780a8a4c
```

Figure 2.3: A snippet of x86 code corresponding to the handling of an HTTP response by the Internet Explorer 7 Web browser.

zero is assigned to the first byte in a TCP flow or in a tainted file, offset one to the second byte and so on. Breaking the unique taint identifier into three components makes it easier to understand where the taint information comes from when analyzing it at the taint sinks.

Our use of taint propagation in this thesis is to define taint sources and to log into the execution trace the taint information for each operand of an instruction executed by the program. For example, Figure 2.3 shows the first three instructions executed by a Web browser that receives an HTTP response. The memory location pointed by the EAX register when the first instruction executes contains the first byte in the HTTP response. If the user has defined the network as a taint source then the location pointed by EAX is tainted. When the first instruction executes, the taint information for the first byte in the HTTP response is copied to the shadow memory corresponding to the lowest byte in EAX. When these instructions are written to the execution trace, the taint information for each operand is recorded. We detail execution traces in the next section.

2.3.2 Tracecap

Tracecap can collect different execution logs for a program execution: execution traces, process state snapshots, function hooks logs, and exported functions logs. We explain each of them next.

Execution traces. Execution traces are instruction-level logs of the execution. An execution trace usually contains the instructions executed by a single user-level process, but it can optionally include the kernel instructions and multiple user-level processes.

The format of an execution trace has evolved over the duration of this thesis and is currently in its seventh version. The current execution trace format has three elements: the header, the body, and an optional trailer. The header contains general trace information like the version, as well as the address and size of each module loaded by a process in the trace. The body of the trace is a sequence of executed instructions. The optional trailer contains additional module information for each process. This module information is often more complete than the one stored in the header, since modules can be loaded during execution and thus their information is not available when the header is written. Storing the module information for an execution is important because a module (e.g., DLL) could be loaded at an address different from its default.

For each executed instruction, the execution trace records, among others, the instruction's address, the identifiers for the process and thread that executed the instruction, the size of the instruction and its raw bytes, and a list of operands accessed by the instruction. For each operand in an instruction, it records, among others, the operand type (e.g., register, memory, immediate), the size, the address (i.e., memory address or unique identifier for each register), the access type (e.g., read or write), the content of the operand before the instruction executes, and the taint information for the operand. For each tainted byte in an operand, it includes a list of taint records, where each taint record comprises the taint source identifier, the taint origin, and the taint offset.

The execution trace uses variable-length encoding to minimize the space needed to record each instruction. Currently, our execution traces are not compressed. We plan to add support for compressed execution traces in the future, as the benefits of trace compression have been demonstrated in related work [13].

Process state snapshots. Tracecap can take snapshots of the state of a process at a desired point in the execution. A process state snapshot includes the content of all memory pages in the address space of the process that are not currently swapped out to disk. It can optionally include the content of all registers, as well as the taint information for both registers and memory addresses. To generate the process state snapshot Tracecap uses the TEMU API to walk the process page table and retrieve the content and taint information for each page.

Hook logs. The TEMU API enables specifying callbacks that are executed when a certain program point (i.e., value of the EIP register) is reached. We call those callbacks *hooks*. Hooks can be global or local. A global hook is executed whenever a process reaches the program point while a local hook is only triggered when a specific process reaches it. A *function hook* is a hook that executes at the entry point of a given function (before any instruction in the function is executed)[1]. A function hook can also define a *return hook*, another callback that is executed when the function returns (after the function's return instruction is executed). Function hooks can be used to perform different actions such as logging the parameters and output values of the function or tainting the output values of the function. Function hooks require access to the function's prototype, so that the callback can locate the parameters of the function in the stack and the output values of the function if a return hook is defined. We have developed over a hundred function hooks for Tracecap. The hooked functions have publicly available prototypes and cover functionality such as file and network processing, heap allocation, Windows registry accesses, string processing, and time operations.

[1]The TEMU API supports specifying the functions by name, to avoid having to update the function's start address when the module that contains the function is loaded at an address that is not the default one. The system also supports specifying the function by ordinal if it is not exported by name.

Tracecap can create function hooks logs with information about the invocations of hooked functions. For example, Tracecap can produce heap allocation logs with information about the calls to heap allocation and free functions invoked by a program during execution. For each heap allocation, the log provides information such as the current instruction counter in the trace, the start address of the heap buffer, the buffer size, and whether the buffer size was tainted. For each heap free operation, the log specifies the instruction counter and the start address of the buffer being freed.

Exported functions log. TEMU parses the header of PE (Windows) binaries that are loaded into memory and extracts the exported symbol names and offsets. Tracecap dumps this information to a file, so that offline analysis can use it to determine function entry points and function names.

2.4 Offline Dynamic Analysis

Offline dynamic analysis techniques work on information logs collected from program executions. All the novel dynamic binary analysis techniques proposed in this thesis operate offline, taking as input execution traces. In addition to execution traces, some of our techniques also take as input other logs from the execution such as the process state snapshots and the heap allocation logs described in the previous section. Although the input is always execution traces, we classify the techniques into *trace-based* if they operate on sequences of assembly instructions (i.e., directly on execution traces) and *IR-based* if they operate on sequences of statements from an intermediate representation (i.e., the execution traces are first translated into the Vine intermediate language).

Converting the execution trace to an intermediate representation (IR) enables more precise analysis but is expensive. Techniques that can operate on either an execution trace or an IR, e.g., dynamic program slicing [2] described in Section 2.4.2, often produce more accurate results when operating on an IR because the IR enables precise reasoning about the instruction semantics and makes instruction side-effects such as setting the processor's status flags explicit. For example, Figure 2.4 shows the list of IR statements resulting from translating the second instruction in Figure 2.3 into the Vine intermediate language. Here, one x86 instruction is translated into 10 IR statements with all side effects (i.e., AF, CF, OF, PF, SF, and ZF status flags) made explicit. However, not all analysis techniques require such precise reasoning. For example, tracking the call stack at any point in the execution provides important contextual information but there is no benefit from translating the execution trace into an IR for callstack tracking. It is more efficient to perform it directly on the execution trace. In addition, our execution traces may contain taint information. Thus, techniques that rely on taint information can operate directly on execution traces. On the other hand, formal analysis like symbolic execution greatly benefits from operating on an IR.

```
label pc_0x78061323_2:
/*cmp    $0x48,%al*/
T_8t0_283:reg8_t = cast(R_EAX_5:reg32_t)L:reg8_t;
T_32t3_286:reg32_t = cast(T_8t0_283:reg8_t)U:reg32_t;
T_0_288:reg32_t = T_32t3_286:reg32_t - 0x48:reg32_t & 0xff:reg32_t;
R_CF_10:reg1_t = T_32t3_286:reg32_t < 0x48:reg32_t;
T_1_289:reg8_t = cast(T_0_288:reg32_t)L:reg8_t;
R_PF_11:reg1_t =
  !cast(((T_1_289:reg8_t >> 7:reg32_t ^ T_1_289:reg8_t >> 6:reg32_t) ^
         (T_1_289:reg8_t >> 5:reg32_t ^ T_1_289:reg8_t >> 4:reg32_t)) ^
         ((T_1_289:reg8_t >> 3:reg32_t ^ T_1_289:reg8_t >> 2:reg32_t) ^
         (T_1_289:reg8_t >> 1:reg32_t ^ T_1_289:reg8_t)))L:reg1_t;
R_AF_12:reg1_t = 1:reg32_t ==
  (0x10:reg32_t & (T_0_288:reg32_t ^
    (T_32t3_286:reg32_t ^ 0x48:reg32_t)));
R_ZF_13:reg1_t = T_0_288:reg32_t == 0:reg32_t;
R_SF_14:reg1_t = 1:reg32_t == (1:reg32_t & T_0_288:reg32_t >> 7:reg32_t);
R_OF_15:reg1_t = 1:reg32_t ==
  (1:reg32_t & ((T_32t3_286:reg32_t ^ 0x48:reg32_t)
                 & (T_32t3_286:reg32_t ^ T_0_288:reg32_t)) >> 7:reg32_t);
```

Figure 2.4: The translation of the second instruction in Figure 2.3 into the Vine intermediate language [19]. Each variable and constant value is followed a colon and its type. The type indicates the size of the variable (e.g., reg8_t for one byte and reg32_t for four bytes).

In this thesis we present both trace-based and IR-based techniques. Our protocol reverse-engineering and binary code reuse techniques, presented in Chapter 3 and Chapter 4 respectively, are trace-based, while our model extraction techniques in Chapters 5–8 are IR-based. In the remainder of this section we provide some background information on previous dynamic analysis techniques that we leverage in this thesis.

2.4.1 Trace-Based Techniques

To enable trace-based techniques we have developed a parsing module that reads an execution trace and provides a clean API to access its information. Similarly, we have developed parsers and APIs to access the process state snapshots and the heap allocation logs that Tracecap also produces. Next, we introduce some trace-based techniques used in this thesis.

Loop detection. The loop detection module extracts the loops present in an execution trace. It supports two different detection methods: static and dynamic. The static method first extracts loop information (e.g., the addresses of the loop head and the loop exit conditions) from control flow graphs, using one of the standard loop detection techniques implemented in Vine [201]. Then, it uses the loop information in a pass over the execution trace to detect the loops present in the execution. The dynamic method does not require any static processing and extracts the loops from a single pass on the execution trace, using techniques that detect loop backedges as instructions that appear multiple times in the same function [107]. The output of both loop detection methods is a list of loops present in the execution trace. The information for each loop includes the position of the loop in the execution trace as well as information about all the iterations of a loop (i.e., loops in an execution trace are unrolled). Both methods have pros and cons. The static method is often more accurate because it can precisely identify loop entry and exit points, but it requires analyzing all the modules (i.e., program executable and dynamically link libraries it uses, including operating system libraries) used by the application, may miss loops that contain indirection, and cannot be applied if the unpacked binary is not available. On the other hand, the dynamic method cannot detect loops that do not complete an iteration, needs heuristics to identify loop exit conditions, but requires no setup and can be used on any execution trace.

Callstack tracking. The call stack tracking module iterates over the execution trace, replicating the function stack for each thread in a given process. Note that the source-level concept of a function may not be directly reflected at the binary code level as explained in Section 2.1. We use an *assembly function* abstraction at the binary level, where all code reachable from the assembly function entry point before reaching an exit point constitutes the body of the assembly function. Note that code reachable only through another entry point (i.e., through a call instruction) belongs to the body of the callee function, rather than to the caller's body. Source-level functions that are inlined by the compiler are considered part of the assembly function where they were inlined. We define exit points to be return instructions. We define entry points to be the target addresses of call instructions[2], as well as the addresses of exported functions provided in the exported function log. Using the exported function log is important to defeat a common obfuscation used by malware where an external function is not invoked using a call instruction, instead the malware program writes the return address into the stack and jumps into the entry point of the function. Note that this deobfuscation only works for external functions. An internal function that the program jumps into, without using a call instruction, will look as part of the caller function to our callstack tracking.

The call stack tracking module makes a pass on the execution trace monitoring entry and exit points. It can handle some special cases such as calls without returns and returns without calls by monitoring the expected return addresses. For example, when faced with the Unix `longjmp` function, a non-local goto used for unstructured control flow, it is able to identify to which call the return instruction that follows the `longjmp` invocation belongs to. The call stack tracking

[2]We also consider a call instruction followed by an indirect jump, an idiom commonly used in Windows binaries.

module provides an API so that a program can identify the ranges of the execution trace that belong to a function run and can execute functionality (e.g., collecting statistics) on each function run. After iterating over the execution trace it outputs a function that given an instruction counter in the execution trace, returns the function nesting at that point of the execution.

Execution trace indexing. An execution trace can only be accessed sequentially, i.e., iterating from the top. To enable random access to the execution trace we provide a tool that builds an execution trace index that stores the file pointer offset for each instruction. The execution trace index is used, among others, by techniques that iterate backwards on the trace.

2.4.2 IR-Based Techniques

In this thesis, IR-based techniques work on the Vine intermediate language representation of an execution trace, which we simply refer to as the IR. Next, we provide background information on three previously proposed IR-based techniques that we use throughout this thesis: symbolic execution, white-box exploration, and dynamic slicing.

Dynamic symbolic execution. In symbolic execution, the values of variables are replaced with symbols and the program's execution on those symbols produces formulas, rather than concrete values [106]. Dynamic symbolic execution performs both concrete and symbolic execution along an execution path [110, 78, 32]. To fix the execution path on which to perform dynamic symbolic execution, an input (e.g., test case) is given to the program and symbolic execution is performed along the concrete execution of the program on the given input. One advantage of dynamic symbolic execution is that if a symbolic formula falls outside the theory of constraints that can be solved, e.g., $A^B \leq 25$ when the solver does not support exponentiation, the formula can be replaced with the corresponding values from the concrete execution. For example, we can replace $A^B \leq 25$ with $A = 3 \wedge B = 2 \wedge (9 \leq 25)$ if A and B had values 3 and 2 respectively in the execution. This limits the generality of the expression, i.e., A and B are fixed to the values seen in the execution, but maintains the correctness of the formula.

In dynamic symbolic execution, when the program reaches a branch predicate that uses some of the symbols, a symbolic branch condition is created. The conjunction of all symbolic branch conditions forms the *path predicate*, a predicate on the symbolic program inputs that captures all the inputs that would follow the same execution path in the program.

In this thesis, we perform offline dynamic symbolic execution on an execution trace. For this, we lift the execution trace into the Vine intermediate language, transform the IR into Single Static Assignment (SSA) form, and replace the program's input (e.g., the value of the EAX register and

```
/* Execution trace */                      /* Slice for ECX at 7 */
0) 78061321:  mov    (%eax),%al            78061330:  push   $0x5
1) 78061323:  cmp    $0x48,%al             78061332:  pop    %ecx
2) 78061325:  jne    0x780a8a4c
3) 7806132b:  cmp    %ecx,-0x4(%ebp)
4) 7806132e:  je     0x7806135c
5) 78061330:  push   $0x5
6) 78061332:  pop    %ecx
7) 78061333:  cmp    %ecx,-0x4(%ebp)
8) 78061336:  jb     0x780a8a8f
```

Figure 2.5: Slicing example. On the left an extended version of the execution trace shown in Figure 2.3. On the right, the slice for the ECX register at instruction 7, which captures all instructions involved in producing the value of the ECX register at that instruction.

the contents of the buffer returned by the `recv` function) with symbols. Optionally, we can taint the variables that we want to be symbolic during program execution and then lift to the Vine intermediate language only those instructions that operate on tainted data.

White-box exploration. Dynamic analysis techniques cover a single execution path in a program. White-box exploration is a technique that leverages dynamic symbolic execution to automatically generate inputs (i.e., test cases) that execute different paths in a program [78, 32]. White-box exploration can be used to increase the coverage of dynamic analysis techniques. It works by first obtaining a path predicate for one execution. Then, it negates one of the symbolic branch conditions in the path predicate and produces a modified path predicate that includes all branch conditions in the original predicate up to the negated condition, plus the negated condition. Then, it queries a constraint solver for an input that satisfies the modified path predicate. If the solver returns an input, that input is fed to the program to generate a new execution that follows a different program path. By repeating this process, white-box exploration can automatically find inputs to explore different execution paths in a program, increasing the coverage of the analysis.

Dynamic program slicing. Dynamic program slicing takes as input an execution of a program and a variable occurrence in that execution, e.g., the value of the EAX register at the beginning of a particular instruction since EAX is used in many instructions, and extracts all statements in the execution that had some effect on the value of the variable occurrence [111, 2]. Dynamic program slicing is the dynamic counterpart of program slicing [223]. The output of slicing a variable occurrence (variable for short) is called a *slice*. Dynamic data slices contain only data dependencies while full dynamic slices contain both data and control dependencies.

```
OUT:reg1_t = (cast(INPUT_10000_0000:reg8_t)U:reg32_t +
  0xffffffb8:reg32_t & 0xff:reg32_t) == 0:reg32_t;
```

Figure 2.6: Formula for the symbolic branch condition corresponding to the conditional jump in Figure 2.3.

We have implemented a slicing algorithm proposed by Zhang et al. [239], which is precise, i.e., only statements with dependencies to the variable are included in the slice. The algorithm works bottom to top on the execution log (slicing can be performed on the execution trace or the IR) and can generate multiple slices (i.e., one per variable) in a single pass. Figure 2.5 shows an example slice. On the left it presents an extended version of the execution trace shown in Figure 2.3. On the right, it shows the slice for the ECX register at instruction 7, which captures all instructions involved in producing the value of the ECX register at that instruction.

In this thesis we use dynamic program slicing as a first step to creating small formulas for symbolic branch conditions. Given a branch condition variable in the IR, we first slice the branch condition variable over the IR, producing a slice that is often much smaller than the complete IR. Then, we create a formula by combining all the statements in a slice together and simplifying the resulting statement using different techniques such as constant folding and constant propagation. The output of this simplification is a formula (often small), which captures how the symbolic program inputs influence the branch condition. For example, Figure 2.6 shows the formula for the symbolic branch condition corresponding to the conditional jump in Figure 2.3, which in short states that INPUT == 0x48.

Part II

Protocol Reverse-Engineering

Chapter 3

Protocol Reverse-Engineering

3.1 Introduction

Protocol reverse-engineering techniques extract the specification of unknown or undocumented network protocols and file formats. Protocol reverse-engineering techniques are needed because many protocols and file formats, especially at the application layer, are closed (i.e., have no publicly available specification). For example, malware often uses undocumented network protocols such as the command-and-control (C&C) protocols used by botnets to synchronize their actions and report back on the nefarious activities. Commercial off-the-shelf applications also use a myriad of undocumented protocols and file formats. Closed network protocols include Skype's protocol [198]; protocols used by instant messaging clients such as AOL's ICQ [88], Yahoo!'s Messenger [235], and Microsoft's MSN Messenger [140]; and update protocols used by antivirus tools and browsers. Closed file formats include the DWG format used by Autodesk's AutoCAD software [7] and the PSD format used by Adobe's Photoshop software [1].

A detailed protocol specification can enable or enhance many security applications. For example, in this chapter we enable active botnet infiltration by extracting the specification of a botnet's C&C protocol and using it for deep packet inspection and rewriting of the botnet's communication. In Chapter 7 we show that a protocol specification enables generating protocol-level vulnerability-based signatures for intrusion detection systems, which are harder to evade than byte-level signatures. Protocol specifications are also the input for generic protocol parsers used in network monitoring [17, 173] and can be used to build protocol-aware fuzzers that explore deeper execution paths than random fuzzers can [176], as well as to generate accurate fingerprints required by fingerprinting tools that remotely distinguish among implementations of the same specification [162].

Currently, protocol reverse-engineering is mostly a time-consuming and error-prone manual task. Protocol reverse-engineering projects such as the ones targeting the MSN Messenger and SMB protocols from Microsoft [148, 210] [1], the Yahoo! Messenger protocol [121], or the OSCAR and ICQ protocols from AOL [72,89], have all been long term efforts lasting years. In addition, protocol reverse-engineering is not a once-and-done effort, since existing protocols are often extended to support new functionality. Thus, to successfully reverse engineer a protocol in a timely manner and keep up the effort through time, automatic protocol reverse-engineering techniques are needed.

Previous work on automatic protocol reverse-engineering proposes techniques that take as input network data [10, 52, 116]. Those techniques face the issue of limited protocol information available in network traces and cannot address encrypted protocols. To address those limitations, in this thesis we present a new approach for automatic protocol reverse-engineering, which leverages the availability of a program that implements the protocol. Our approach uses dynamic program binary analysis techniques and is based on the intuition that monitoring how the program parses and constructs protocol messages reveals a wealth of information about the message structure and its semantics.

Compared to network traces, program binaries contain richer protocol information because they represent the implementation of the protocol, which is the most detailed description of the protocol in absence of the specification. Understanding the protocol implementation can be beneficial even for protocols with a publicly available specification, because implementations often deviate from the specification. In addition, for encrypted protocols, the program binary knows the cryptographic information required to decrypt and encrypt protocol data. Thus, we can wait until the program decrypts the received network data to start our analysis and stop it before the program encrypts the network data to be sent in response, thus revealing the structure and semantics of the underlying protocol.

Our work in context. This chapter comprises work published in two conference articles. The first article appeared in the proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007). It presented a system called *Polyglot* [30], which implemented the first approach for automatic protocol reverse-engineering using dynamic binary analysis. Polyglot uses the intuition that monitoring the execution of a program that implements the protocol reveals a wealth of information about the protocol. Polyglot extracts only the message format of a received message. The second article appeared in the proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009). It presented a system called *Dispatcher* [26], which in addition to the techniques introduced in Polyglot, implemented techniques to extract the message format for a sent message. It also implemented semantics inference techniques for both sent and received messages, which we had previously introduced in a Technical Report in 2007 [28].

[1]Microsoft has since publicly released the specification of both protocols as part of their Open Specification initiative [139]

After the publication of Polyglot, other research groups published automatic protocol reverse-engineering techniques that used dynamic binary analysis for extracting the protocol grammar [229, 124, 54] and the protocol state-machine [44]. The works that focus on protocol grammar extraction use the approach we introduced in Polyglot of monitoring the execution of a program that implements the protocol. Their techniques target two issues: 1) they consider the message format to be hierarchical [229, 124, 54], rather than flat as considered in Polyglot, and 2) they extend the problem scope from extracting the message format as done in Polyglot, to extracting the protocol grammar by combining information from multiple messages [229, 54]. In Dispatcher we still focus only on message format extraction because it is a pre-requisite for both protocol grammar and state-machine extraction, but we consider the hierarchical structure of the protocol messages. In this chapter, we present a unified view of the techniques introduced in Polyglot and Dispatcher that considers the hierarchical structure of protocol messages. We also unify the protocol nomenclature used across the different protocol reverse-engineering works.

3.2 Overview & Problem Definition

In this section we introduce automatic protocol reverse-engineering and its goals, describe the scope of the problem we address, introduce common protocol elements and terminology, formally define the problem, and provide an overview of our approach.

3.2.1 Automatic Protocol Reverse-Engineering

The goal of automatic protocol reverse-engineering is given an undocumented protocol or file format to extract the *protocol grammar*, which captures the structure of all messages that comprise the protocol, and the *protocol state machine*, which captures the sequences of messages that represent valid sessions of the protocol. In this thesis we focus on reversing application layer protocols because those comprise the majority of all protocols and are more likely to be undocumented. In addition, we consider file formats a simple instance of a protocol where there are no sessions and each file corresponds to a single message.

Extracting the protocol grammar usually comprises two steps. First, given a set of input messages, extract the *message format* of each individual message. Second, combine the message format from multiple messages of the same type to identify complex message properties such as field alternation and optional fields. In this thesis we address the first step of protocol grammar extraction: extracting the message format for a given message. Extracting the message format is a pre-requisite for extracting both the protocol grammar and the protocol state-machine. The message format captures the field structure and the field semantics of the message, which we describe next.

Message format. The message format has two components: the *message field tree* and a *field attribute list* for each node in the tree. The message field tree[2] is a hierarchical tree structure where each node represents a field in the message and is tagged with a *[start:end]* range of offsets where the field appears in the message, where offset zero is the first byte in the message. A child node represents a subfield of its parent, and thus corresponds to a subrange of the parent field in the message. The children of a parent have non-overlapping ranges and are ordered using the lowest offset in their range. The root node represents the complete message, the internal nodes represent *records*[3], and the leaf nodes represent *leaf fields*[4], the smallest semantic units in a protocol. Note that leaf fields are sometimes referred to simply as fields. In this thesis, when we refer to fields in plural, we mean any node in the message field tree, which includes both records and leaf fields.

In addition to the range, a node contains a field attribute list, where each attribute captures a property of the field. Table 3.1 shows the field attributes that we consider in this thesis. The field boundary attribute captures how the recipient locates the boundary between the end of this field and the beginning of the next field in the message. For fixed-length fields the receiver can find the boundary using the constant field length value, which is known a priori. For variable-length fields the receiver can use a *delimiter*, i.e., a constant value that marks the end of the field, or a *length field*. The field dependencies attributes captures inter-field relationships such as this field being the length of another field or this field being the checksum of multiple other fields in the message. The field semantics attribute captures the type of data that a field carries. We explain the different protocol elements in more detail in the next section.

Note that the message field tree with the associated field ranges perfectly describes the structure of a given message. However, without the attribute list we cannot learn anything from this message that can be applied to other instances of the same message type (e.g., from one HTTP GET request to another).

Field semantics. One important field attribute is the field semantics, i.e, the type of data that the field contains. Typical field semantics include timestamps, hostnames, IP addresses, ports, and filenames. Field semantics are fundamental to understand what a message does and are important for both text and binary protocols. For example, an ASCII-encoded integer in a text-based protocol can represent among others a length, a port number, a sleep timer, or a checksum value. Field semantics are critical for many applications, e.g., they are needed in active botnet infiltration to identify interesting fields in a message to rewrite.

[2]Also called protocol field tree [124].
[3]Also called hierarchical fields [124, 26] and complex fields [229].
[4]Also called finest-grained fields [124].

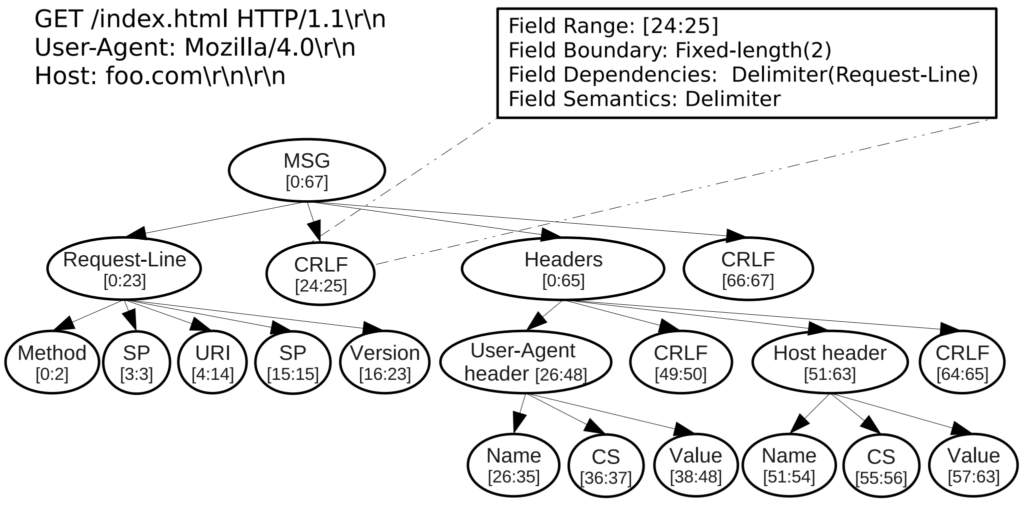


Figure 3.1: Message field tree for the HTTP request on the upper left corner. The upper right corner box shows the attribute list for one of the delimiters.

Attribute	**Value**
Field Range	Start and end offsets in message
Field Boundary	Fixed-length(l), Variable-length($Length$), Variable-length($Delimiter$)
Field Dependencies	Length(x_i), Delimiter(x_i), Checksum($x_i, \ldots, x_j$)
Field Semantics	The type of data the field carries. A value from Table 3.3

Table 3.1: Field attributes used in this thesis. Each attribute captures a property of the field.

HTTP running example. Figure 3.1, captures the message field tree for an HTTP request. The HTTP request message is shown on the upper left corner and the box on the upper right corner shows the attribute list for one of the nodes. The root node in Figure 3.1 represents the complete HTTP request, which is 68 bytes long. There are four records: the *Headers*, the *Request-Line*, the *User-Agent* header, and the *Host* header. HTTP mostly uses delimiters to mark the end of the variable-length fields. The field attribute list for the CRLF field shown in the figure, shows in the field semantics attribute that the *CRLF* field is a delimiter and in the field dependencies field that its target is the *Request-Line*. The HTTP specification is publicly available [65] and Figure 3.2 shows a partial HTTP grammar, taken from the specification, that covers most production rules related to our example HTTP request.

3.2.2 Protocol Elements

In this section we describe some elements commonly used in protocols and how they are represented in the message field tree and field attribute list.

```
HTTP-message = Request | Response
Request = generic-message
Response = generic-message
generic-message = start-line
                  *(message-header CRLF)
                  CRLF
                  [ message-body ]
start-line = Request-Line | Status-Line
message-header = field-name ":" [ field-value ]
```

Figure 3.2: Partial HTTP protocol grammar from RFC 2616 [65].

Layer	PDU
Physical Layer	bit
Data Link Layer	frame
Network Layer	packet
Transport Layer	segment
Application Layer	message

Table 3.2: Protocol Data Units (PDUs) for the different networking layers.

Message. We define a message to be the Protocol Data Unit (PDU) of the application layer, where a PDU is the information that is delivered as a unit among peer entities in a networking layer. Table 3.2 shows the PDUs for different networking layers. We separate the application layer PDU with a horizontal line because there is no defined PDU for layers above the Transport layer [203].

Fixed-length and variable-length fields. Each field, regardless if a record or a leaf field, is either fixed-length or variable-length. The length value of a fixed-length field is static, i.e., it does not change across multiple instances of the same field. The length value for a fixed-length field is part of the protocol specification and known a priori to the implementations of the protocol. In contrast, the length of a variable-length field is dynamic, i.e., it can change across multiple instances of the same field. Protocol specifications need to describe how an implementation identifies the length or the boundary of a variable-length field. The main protocol elements used for this task are length fields and delimiters, which we describe next. The field boundary attribute captures whether a field is fixed-length or variable-length and for the latter whether it uses a delimiter or a length field. In our HTTP running example, all fields except the delimiters themselves are variable-length, while in our MegaD running example only the *MSG*, *Host-Info*, and *Padding* fields are variable-length, the rest are fixed-length.

Length fields. A length field captures the size of a *target* variable-length field, which can be a record or a leaf field. A length field always precedes its target field in a message, but it does not need to be its immediate predecessor. The length field can use different units. For example, in Figure 3.7 the *Msg-Length* field encodes the total length of the message in 64-bit units, but the *Length* field encodes the length of the *Host-Info* record in bytes. The value of the length field is often the output of a formula that may use the real length of the target field plus or minus some known constant. For example, a record may have three child fields: a fixed-length type field, a fixed-length length field, and a variable-length payload. In this case the length field can capture the payload length or the record length which includes the fixed-length of both the type field and the length field itself. The field dependencies attribute captures whether a field is a length field and what the target variable-length field is. The field boundary attribute captures for a target variable-length field whether its boundary is located using a particular length field.

Delimiters. A delimiter[5] is a constant used to mark the boundary of a target variable-length field. Delimiters are fields themselves and are always the successor of the target variable-length field they delimit. Delimiters are part of the protocol specification and known to the implementations of the protocol. Delimiters can be used in binary, text or mixed protocols, e.g., delimiters are used in HTTP, which is mainly a text protocol, and also in SMB, which is mainly a binary protocol. They can be formed by one or more bytes. For example, in Figure 3.1, the Carrier-Return plus Line-Feed two-byte sequence (CRLF) is used as a delimiter to mark the end of the start-line and the different message headers [65], while SMB uses a single null byte to separate dialect strings inside a *Negotiate Protocol Request* [186]. Protocols can have multiple delimiters. For example, in Figure 3.1, in addition to the CRLF delimiter to separate headers, there is also the space (SP) delimiter that marks the end of the *Method* and *URI* fields, as well as the semicolon plus space (CS) delimiter that separates the *Name* from the *Value* in each header field. As shown in the field attribute list in Figure 3.1, the field dependencies attribute captures whether a field is a delimiter and which field is its target. The field boundary attribute captures for a target variable-length field whether its boundary is located using a particular delimiter.

Field Sequences. Field sequences, or sequences for short, are lists of consecutive fields with the same type. Sequences are used in file formats such as WMF, AVI or MPEG, and also in network protocols such as HTTP. A sequence is always variable-length, regardless if the fields that form the sequence are fixed-length or variable-length. The end of a sequence is marked using a delimiter or a special length field called a counter field. For example, in Figure 3.1, the *Headers* field is a sequence and an empty line (CRLF) delimiter is used to mark its end. Note that an array is a special case of a sequence where each field in the sequence has fixed-length. A sequence is simply a record

[5]In our early protocol reverse-engineering work [30] we referred to delimiters as separators. Since then, we have adopted the term delimiter because it has been more commonly used in follow-up work.

in the message field tree with a field semantics attribute value that indicates so. All children of a sequence are of the same type.

Keywords. Keywords are protocol constants that appear in the protocol messages. Keywords are part of the protocol specification and known a priori to the implementations. Not all protocol constants are keywords, since there are protocol constants that never appear in a message, such as the maximum length of a field. Keywords can appear in binary, text, and mixed protocols and can be strings or numbers. For example, in Figure 3.1 the “GET”, “HTTP”, “User-Agent”, and “Host” strings are all keywords, while in Figure 3.7 the version number is also a keyword. The field semantics attribute captures whether a field carries a keyword. A field can carry different keywords in different instances of the field. For example, the *Method* field in Figure 3.1 carries the “GET” keyword but in other HTTP requests it could carry the “POST” or “HEAD” keywords. Note that according to this definition delimiters are also keywords. We differentiate delimiters from other keywords because of their particular use.

Dynamic fields. Dynamic fields have been defined by previous work to be fields whose value may change across different protocol dialogs [53]. According to that definition almost any field in a protocol is dynamic. There are very few fields in protocols whose value never changes because they can encode very little information. In this thesis, we define dynamic fields to be fields that carry protocol-independent information, which means fields that never carry a protocol keyword.

3.2.3 Problem Definition

In this thesis we develop protocol reverse-engineering techniques to address two problems: *message format extraction* and *field semantics inference*. Message format extraction is the problem of extracting the message field tree for one message of the protocol. It can be applied to a message *received* by the application, as well as to a message *sent* by the application in response to a previously received message. Field semantics inference is the problem of given a message field tree, tagging each field in the tree with a field semantics attribute specifying the type of data the field carries.

The input to our message format extraction and field semantics inference techniques is execution traces taken by monitoring an application that implements the protocol, while it is involved in a network dialog using the unknown protocol. The execution traces can be obtained by monitoring a live dialog, where the application communicates with another entity somewhere on the Internet, or an offline dialog, where we replay a previously captured dialog from the unknown protocol to the application. In both cases the application runs inside the execution monitor presented in Chapter 2, which produces execution traces.

3.2.4 Approach

We design message format extraction and protocol inference techniques for both messages received and sent by an application. Thus, our approach can analyze both sides of the communication of an unknown protocol, even when an analyst has access only to the application implementing one side of the dialog. This is important because there are scenarios where access to applications that implement both sides of a dialog is difficult, such as reversing a botnet's C&C protocol where the binary of the C&C server is rarely available or reversing a proprietary instant-messaging protocol (e.g., Yahoo's YMSG or Microsoft's MSNP) where client implementations are publicly available, but server implementations are not.

To extract the format of *received* messages we use the following intuition: by monitoring how a program parses a received message we can learn the message format because in order to access the information in the leaf fields, the program first needs to find those fields by extracting the hierarchical structure of the message. By monitoring the parsing process, we can learn what the program already knows, e.g., the length of fixed-length fields and the values used as delimiters, as well as what the program has to discover, e.g., the boundaries of the variable-length fields. We present our message format extraction techniques for received messages in Section 3.3.

To extract the format of *sent* messages we use the following intuition: programs store fields in memory buffers and construct the messages to be sent by combining those buffers together. We define the *output buffer* to be the buffer that contains the message about to be sent at the time that the function that sends data over the network is invoked. As a special case, for encrypted protocols the output buffer contains the unencrypted data at the time the encryption routine is invoked. Our intuition is that the structure of the output buffer represents the inverse of the structure of the sent message. We propose *buffer deconstruction*, a technique to build the message field tree of a sent message by analyzing how the output buffer is constructed from other memory buffers in the program. We present our message format extraction techniques for sent messages in Section 3.4.

Our techniques to extract the message format differ for received and sent messages. For received messages, our techniques focus on how the program parses the message and leverage taint propagation, a data-flow technique that allows us to follow how the received message is handled throughout the parsing. For sent messages, our techniques focus on how the program builds the message from its individual fields and leverage buffer deconstruction, which analyzes how the different memory buffers are used to fill the output buffer. Note that we do not leverage taint propagation for extracting the message field tree of sent messages, because only a fraction of all possible sources of taint information during message creation (e.g., output of system calls and data sections in the program) is actually used to build the sent message.

To infer the field semantics, we use type-inference-based techniques that leverage the observation that many functions and instructions used by programs contain known semantic information that can be leveraged for field semantics inference. When a field in the received message is used to derive the arguments of those functions or instructions (i.e., semantic sinks), we can infer its semantics. When the output of those functions or instructions (i.e., semantic sources) are used to derive some field in the output buffer, we can infer its semantics. We present our field semantic inference techniques for both received and sent messages in Section 3.5.

One limitation of our message format extraction and field inference techniques is that they work at the byte level. Thus, they currently cannot handle fields smaller than 8-bit, such as the QR (query or response) bit or the 4-bit Opcode (kind of query) in a DNS request [149]. While our techniques can be extended to operate at the bit-level with some engineering effort, an end-to-end solution requires the building blocks we use e.g., taint propagation, to also operate at the bit level.

Encrypted protocols. To handle encrypted protocols such as MegaD's C&C protocol, we use the intuition that the program binary knows the cryptographic information (e.g., cryptographic routines and keys) required to decrypt and encrypt the protocol messages. Thus, we can wait until the program decrypts the received message to start our analysis and stop the analysis before the program encrypts the message to be sent in response. Compared to previous work, we propose extensions to a recently proposed technique to identify the buffers holding the unencrypted received message [221]. Our extensions generalize the technique to support implementations where there is no single boundary between decryption and protocol processing, and to identify the buffers holding the unencrypted sent message. We present our handling of encrypted protocols in Section 3.6.

Per-message execution traces. An execution trace may contain the processing of multiple messages sent and received by the application during the network dialog. To separately analyze each message we need to split the execution trace into per-message traces. This is challenging when two consecutive messages are sent on the same direction of the communication. For example, MegaD uses a TCP-based C&C protocol. In a C&C connection the bot sends a request to the C&C server and receives one or more consecutive responses with the same format. At that point the question is whether to consider the response from the server a single message in which case there is a single message field tree where the child of the root node corresponds to a sequence with two child records, or to consider the response as two messages, in which case there are two separate message field trees.

Some work defines a message to be all data received by a peer before a response is sent, i.e., before the application calls the function that writes data to the socket [44]. This makes the response from MegaD's C&C server to be a single message. In this thesis we use a different definition

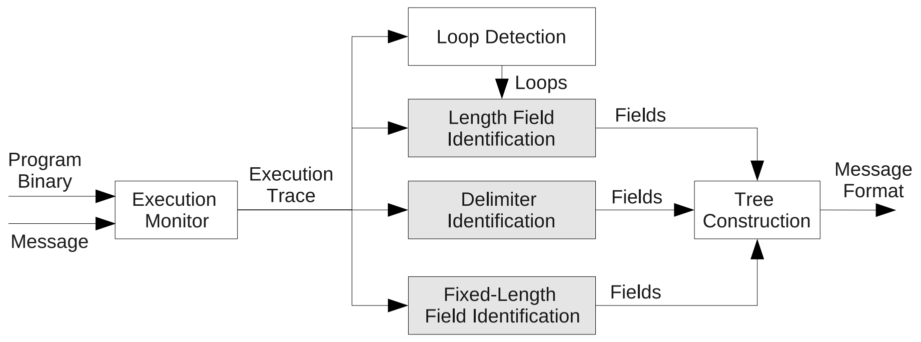


Figure 3.3: Message format extraction for received messages.

of what a message is and split the execution trace into two traces every time that the program makes a successful call to write data to a socket (e.g., *send*) and every time that the program makes a successful call to read data from a socket (e.g., *recv*), except when the argument defining the maximum number of bytes to read is tainted. In this case, the read data is considered part of the previous message and the trace is not split. This handles the case of a program reading a field conveying the length of the message payload and using this value to read the payload itself.

3.3 Message Format Extraction for a Received Message

The input for extracting the message format for a received message is an execution trace of the program while it parses the protocol message that we want to extract the format for. We have introduced execution trace logging and taint propagation in Chapter 2. Here, the execution monitor taints each byte of the received message with a different taint offset where offset zero corresponds to the first byte in the message. The execution trace contains for each instruction operand whether the operand is tainted. If tainted, it contains the taint offsets for each byte in the operand. We refer to the taint offsets as *positions* in the received message. The output of this process is the message format as a message field tree with an attribute list for each node. This message field tree has no semantics attribute. In Section 3.5 we show how to extract the values for the field semantics attribute.

Figure 3.3 illustrates the message format extraction process for a received message. It shows that the execution trace is the input to three modules: delimiter identification, length identification, and fixed-length field identification. The delimiter and length identification modules focus on variable-length fields that use delimiters and length fields to mark their boundaries. In addition to the execution trace, the length identification module also takes as input the loop information provided by the loop detection module we introduced in Chapter 2. We present the delimiter identification module in Section 3.3.1 and the length identification module in Section 3.3.2. The fixed-length identification focuses on fixed-length fields and is presented in Section 3.3.3. The fields identified by those three modules are added to the message field tree by the tree construction module.

3.3.1 Identifying Delimiters

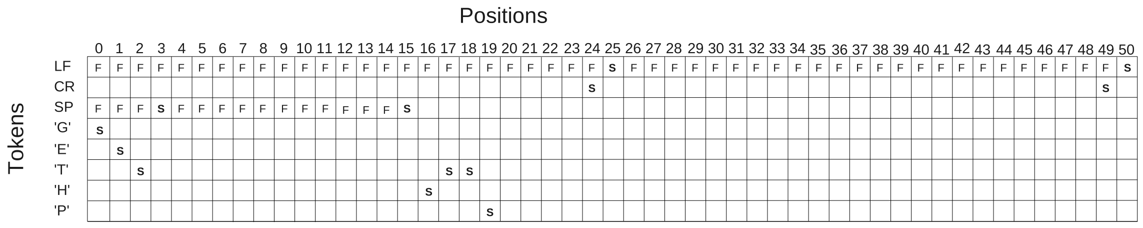
In Section 3.2.2 we defined a delimiter to be a constant used to mark the boundary of a target variable-length field. Delimiters are part of the protocol specification and known to the programs that implement the protocol. The intuition to identify delimiters is that when parsing a received message, programs search for the delimiter by comparing the different bytes in the message against the delimiter. When a successful comparison happens, the program knows it has found a delimiter and therefore the boundary of the target variable-length field that precedes the delimiter.

For example, a Web server that receives the HTTP request in Figure 3.1 knows that the Carrier-Return plus Line-Feed (CRLF) sequence is the delimiter used to mark the end of the *Request-Line* field. The server compares the bytes in the request from the beginning (position zero) until finding the CRLF value at positions [24:25]. At that point the program knows that the range of the *Request-Line* is [0:23]. Similarly, the program knows that the space character (SP) is the delimiter used to mark the end of the *Method* and *URI* fields inside the *Request-Line*. Thus, the server compares the bytes in the *Request-Line* range with the space character until it finds it at position 3. At that point, it knows that the *Method* field comprises range [0:2]. Then, it continues scanning for the next occurrence of the space character, which is found at position 15. At that point it knows that the *URI* field comprises the range [4:14] and that the remainder of the *Request-Line* (range [16:23]) has to correspond to the *Version* field.

In a nutshell, our delimiter identification technique scans the execution trace looking for comparison operations between bytes from the received message (i.e., tainted bytes) and constant values (i.e., untainted bytes). For each comparison operation found, it stores for each tainted byte involved in the comparison, the position of the tainted byte, the constant value it was compared against, and the result of the comparison (i.e., success or failed). Then, it searches for tokens (i.e., byte-long constants) that are compared against multiple consecutive positions in the input message.

The detailed process comprises 4 steps: 1) generate a *token table* that summarizes all comparisons between tainted and untainted data in the execution trace, 2) use the token table to identify byte-long delimiters, 3) extend byte-long delimiters into multi-byte delimiters, and 4) add the delimiters and their target-fields to the message field tree. We describe these steps below.

Our delimiter identification technique has two important properties. First, it makes no assumptions about the constants used as delimiters. Instead, it identifies delimiters by the way they are used. Second, it does not assume that the program searches for the delimiter in an ascending position order. All byte-comparisons between tainted and untainted data are recorded in the token table



Positions

Tokens	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
LF	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	s
CR																									s	
SP	F	F	F	s	F	F	F	F	F	F	F	F	F	F	F	s										
'G'	s																									
'E'		s																								
'T'			s															s	s							
'H'																	s									
'P'																				s						

Tokens	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
LF	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	s
CR																								s	
SP																									
'G'																									
'E'																									
'T'																									
'H'																									
'P'																									

Figure 3.4: Partial token table for the HTTP request in Figure 3.1.

in the first step, before delimiters are identified. Thus, it does not matter the order in which the comparisons are done by the program, which is important because some programs like the Apache Web server scan backwards to find delimiters.

Generating the token table. The token table summarizes all comparisons between tainted and untainted data in the execution trace. Each row in the token table represents a token (i.e., a byte-long constant value) that at some point in the execution was compared against tainted data. Thus, the token table can have at most 256 rows. Each column represents a position in the received message from zero to the message size minus one. Each entry in the table represents whether the comparison between the token and the position in the received message was successful (S) or failed (F). Figure 3.4 shows a partial token table for the comparisons that a Web server performs on the first 51 bytes of the HTTP request in Figure 3.1. For brevity, we limit the table to only 8 tokens.

To populate the token table, the delimiter identification module scans the execution trace for comparison operations that involve tainted and untainted data. It breaks each comparison operation into one-byte comparisons and for each one-byte comparison it extracts the position of the tainted byte, the token it was compared against, and the result of the comparison. It adds a new entry with this information into the token table. Only equality comparisons are added to the table and comparison operations include not only compare (`cmp`) instructions but also other operations that compilers use to compare operands such as string comparison (`scas`) instructions or `test` instructions with identical operands (used to cheaply compare if an operand has zero value).

Extracting byte-long delimiters. To extract byte-long delimiters the delimiter identification module scans each row of the token table in ascending position order to find all sequences of consecutive positions that were compared against the token. A new sequence is started every time the current position is not consecutive with the previous one and every time a successful comparison is found. The reason to break a sequence at a successful comparison is that a successful comparison marks the presence of the delimiter and thus the end of the field it delimits. Once the list of all sequences for a token has been extracted, any sequence shorter than 3 positions is removed to avoid including spurious comparisons. We call each sequence of consecutive positions a *scope* and the output of this step is a list of byte-long delimiters with the associated scopes where the delimiter was used.

For example, from the token table in Figure 3.4 this step outputs two one-byte delimiters: the Line-Feed (LF) token with scopes [0:25] and [26:50], and the space (SP) token with scopes [0:3] and [4:15]. This shows that each one-byte delimiter can have multiple scopes and that different one-byte delimiters may have overlapping scopes since the scopes for the SP token are a subrange of the scope for the LF token. Thus, the delimiter scope hierarchy captures the hierarchical relationship between the *Request-Line* and the *Method* and *URI* fields. Note that two one-byte delimiters cannot have identical scopes since we require a successful comparison to mark the end of a scope.

Extending delimiters. When a delimiter consists of multiple bytes, e.g., the CRLF delimiter, the program can use different ways to find it such as searching for the complete delimiter or only searching for one byte in the delimiter and when it finds it, checking if the remaining bytes in the delimiter are also present. For multi-byte delimiters, the previous step identifies only one byte in the delimiter or all the bytes but as independent byte-long delimiters. For example, the token table in Figure 3.4 corresponds to a Web server that scans for the LF character and once found, it checks if the CR character is present in the previous position. In this case, the previous step identifies only the LF token as a one-byte delimiter.

In this last step, we try to extend each one-byte delimiter by analyzing the comparisons at the positions before and after all occurrences of the delimiter, i.e., the comparisons at the predecessor and successor positions for the last position in each scope. If the token table shows a successful comparison with the same token for all predecessor positions, we extend the delimiter with that token. If the token table shows a successful comparison with the same token for all successor positions, we extend the delimiter with that token and increase all scopes by one. The process recurses on each delimiter that was extended, until no more delimiters are extended. At that point, any duplicate scopes for a delimiter are removed. The output of this step is a list of multi-byte delimiters with the scopes where they are used.

For example, the one-byte LF delimiter identified in the previous step has scopes [0:25] and [26:50]. This step first checks the successor positions 26 and 51, finding no successful comparisons with the same token at those positions. Then, it checks the predecessor positions 24 and 49, finding that they all have a successful comparison against the CR token. Thus, the one-byte LF delimiter is extended to be a two-byte CRLF delimiter with identical scopes. The same process for the one-byte SP delimiter produces no extensions and the output of this step is two delimiters: CRLF with scopes [0:25] and [26:50], and SP with scopes [0:3] and [4:15].

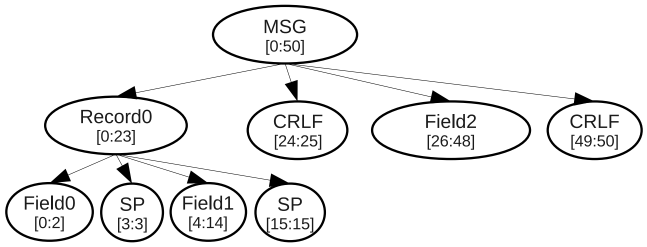


Figure 3.5: Partial message field tree generated by inserting the fields derived by identifying delimiters using the token table in Figure 3.4 into an empty tree.

Adding the delimiters and target fields to the message field tree. Once the delimiters have been identified, each scope is used to create two fields: a delimiter field with a range that covers the bytes in the delimiter and a variable-length field that covers the remainder of the scope. Both fields are added to the message field tree. For example, the [0:25] scope for the CRLF delimiter produces a delimiter field with range [24:25] and variable-length field with range [0:23]. Note that, the operation that inserts new nodes into the message field tree uses the field ranges to determine the correct position of the field in the tree. For example, Figure 3.5 shows the message field tree after inserting the fields derived by the delimiter identification process using the token table in Figure 3.4 into an empty tree. Note that the message field tree has a gap at depth 2 for the range [16:23], which corresponds to the *Version* field in Figure 3.1. Once the length and fixed-length field identification terminates, the tree construction module fills the gaps with fields.

3.3.2 Identifying Length Fields

The intuition behind our techniques for length field detection is the following. The application data is stored in a memory buffer before it is accessed (it might be moved from disk to memory first). Then a pointer is used to access the different positions in the buffer. Now when the program has located the beginning of a variable-length field, whose length is determined by a length field, it needs to use some value derived from the length field to advance the pointer to the end of the field. Thus, we identify length fields when they are *used* to increment the value of a pointer to the tainted data. For example, in Figure 3.6 we identify the length field at positions 12-13 when it is used to access positions 18-20.

We consider two possibilities to determine whether a field is being used as a length field: 1) the program computes the value of the pointer increment from the length field and adds this increment to the current value of the pointer using arithmetic operations; or 2) the program increments the pointer by one or some other constant increment using a loop, until it reaches the end of the field, indicated by a stop condition.

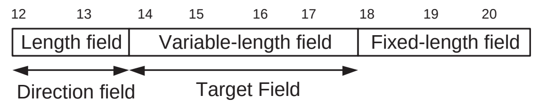


Figure 3.6: Length field example.

Incrementing the pointer using arithmetic operations. For the first case, the program performs an indirect memory access where the effective address has been computed from some tainted data. Thus, when we find an indirect memory access that: 1) accesses a tainted memory location, and 2) where the effective address has been computed from tainted data (i.e., the base or index registers used to compute the address were tainted), we mark all the consecutive positions used to compute the effective address as part of a length field. In addition, we mark the smallest position in the effective address as the end of the target field. For example, in Figure 3.6 if the instruction is accessing positions 18-20, and the address of the smallest position (i.e., 18) was calculated using taint data coming from positions 12-13, then we mark position 12 as the start of a length field with length 2, and position 18 as the end of the target field. If a length field is used to access multiple positions in the buffer, we only record the smallest position being accessed. For example, if we have already found the length field in Figure 3.6 directs to position 18, and it appears again in an indirect memory access to position 27, we still consider the end of the target field to be position 18.

Incrementing the pointer using a loop. For the second case, since the pointer increment is not tainted (i.e., it is a constant) then the previous approach does not work. In this case we assume that the stop condition for the pointer increment is calculated using a loop. The length identification module uses the loop information provided by the dynamic loop module presented in Chapter 2 to identify loops in the trace that have a tainted exit condition. After extracting the loops we check if the loop stop condition is generated from tainted data, if so we flag the loop as tainted. Every time the program uses a new position, we check if the closest loop was tainted. If so, we flag a length field. Our techniques are not complete because there are other possibilities in which a program can indirectly increment the pointer, for example using switch statements or conditionals. But, these are hardly used since the number of conditions could potentially grow very large, up to maximum value of the length field.

Variable-length fields. Length fields are used to locate the end of a variable-length target field. To determine the start of the target variable-length field, without assuming any field encoding, we use the following approach. Length fields need to appear before their target field, so they can be used to skip it. Most often, as mentioned in [53] they precede the target field in the field sequence. After we locate a length field, we consider that the sequence of bytes between the last position belonging to the length field and the end of the target field, corresponds to a variable-length field. For example, in Figure 3.6, when the length field at positions 12-13 is used to access positions 18-20, we identify everything in between (i.e., 14-17) to be a variable-length field. Thus, if a fixed-length field follows the variable-length field and is not used by the program either because the field is not needed or not supported by the program, then we will include the fixed-length field as part of the variable-length field.

Note that our approach detects length fields by looking at pointer increments and thus, it is independent of the encoding used in the length field. In contrast, previous work uses techniques for identifying length fields that assume the length is encoded using some pre-defined encoding, such as the number of bytes or words in the field [52, 53]. Thus, those techniques would miss length fields if they use other encodings, which do not belong to the set of pre-defined encodings being tested.

3.3.3 Identifying Fixed-Length Fields

In Sections 3.3.1 and 3.3.2 we have presented our techniques to identify the boundaries of variable-length fields. In this section we present our techniques to identify the boundaries of fixed-length fields. The intuition behind our fixed-length field identification technique is that fields are semantic units and programs take decisions based on the value of a field as a whole. Thus, when a field comprises multiple bytes, those bytes need to be used together in arithmetic operations, comparisons or other tasks. In addition, most fields are independent of other fields, so bytes belonging to different fields rarely are used in the same instruction. The exception to this rule are special relationships such as length fields, pointer fields or checksums.

Our approach for identifying multiple bytes belonging to the same field is the following. Initially, we consider each byte received from the network as independent. Then, for each instruction, we extract the list of positions that the taint data involved in the instruction comes from. Next, we check for special relationships among bytes, specifically in this paper we check for length fields, using the techniques explained in Section 3.3.2. If no length field is found, then we create a new fixed field that encompasses those positions. For example if an instruction uses tainted data from positions 12-14 and those positions currently do not belong to a length field, then we create a fixed field that starts at position 12 and has length 3.

If a later instruction shows a sequence of consecutive tainted positions that overlaps with a previously defined field, then we extend the previously defined field to encompass the newly found bytes. One limitation is that fixed-length fields longer than the system's word size (four bytes for 32-bit architectures, eight for 64-bit architectures) cannot be found, unless different instructions overlap on their use. Note that fields larger than 4 bytes are usually avoided for this same reason,

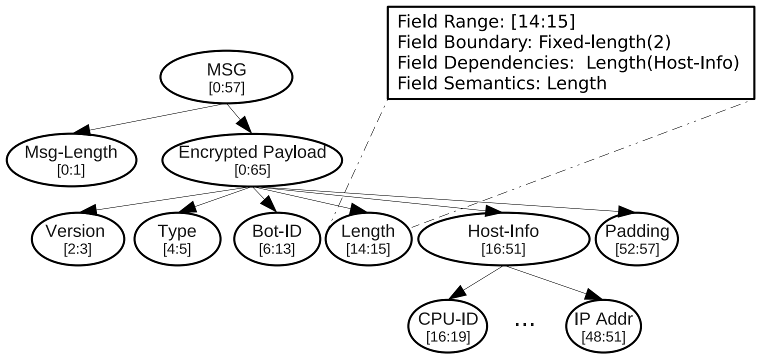


Figure 3.7: Message field tree for the MegaD Host-Information message.

since most systems have 32-bit architectures where longer fields need several instructions to be handled. For fields longer than 4 bytes, our message format truncates them into four-byte chunks. Note that this does not affect variable-length fields which are identified by finding the delimiters and the length fields.

Even with this limitation, our approach is an improvement over previous work [52], where each binary-encoded byte is considered a separate field. Using that approach, two consecutive fixed-length fields, each of length 4 bytes, would be considered to be 8 consecutive byte-long fixed-length fields.

3.4 Message Format Extraction for a Sent Message

The input for extracting the message format for a sent message is an execution trace of the program while it constructs the response to a given message. The output of this process is the message format as a message field tree with an attribute list for each node. This message field tree has no semantics attribute. In Section 3.5 we show how to extract the values for the field semantics attribute.

Our techniques to extract the message format for sent messages do not leverage taint propagation in the same way than the techniques for received messages do. For sent messages our techniques mostly work backwards (i.e., bottom-to-top) on the execution trace, while taint propagation is a forward (i.e., top-to-bottom) technique. Here, we leverage taint propagation in a different way by tainting the memory regions where the program under analysis and all dynamic libraries (DLLs) shipped with the program are loaded. Intuitively, protocol constants known to the program are stored in the data sections of those modules. Taint propagation allows us to track how those constants are used to build the sent message. This is needed to identify delimiters and keywords, which are constants as explained in Section 3.2.2.

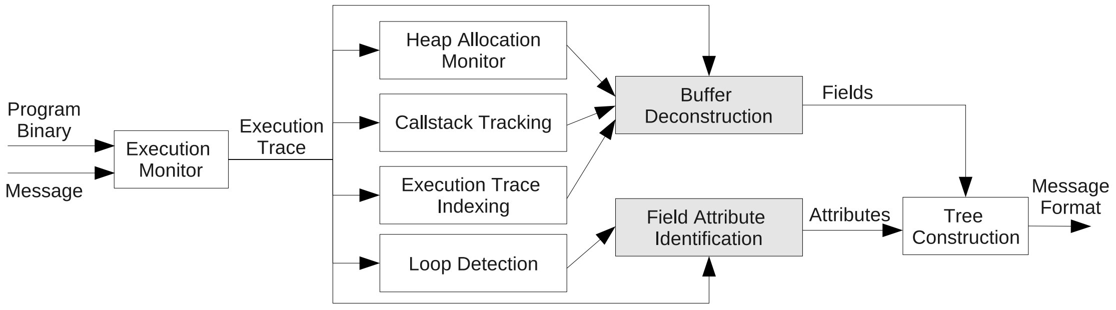


Figure 3.8: Message format extraction for sent messages.

MegaD running example. The MegaD botnet is one of the most prevalent spam botnets in use at the time of writing [130, 99]. MegaD uses an encrypted, binary (under the encryption), previously undocumented C&C protocol. Figure 3.7, corresponds to a message constructed by a MegaD bot to communicate back to the C&C server information about the bot's host. We use the message in Figure 3.7 as a running example throughout this section. The message is 58 bytes long and is partially encrypted. The *Msg-Length* field represents the total length of the message in 4-bit units and is unencrypted. The *Encrypted Payload* record corresponds to the encrypted part of the message. The other record contains the host information such as the CPU identifier and the IP address of the host.

Approach overview. The process of extracting the message format of a sent message is illustrated in Figure 3.8. It comprises three steps. The *preparation* step consists of a forward pass over the execution trace to extract information about the execution. This step uses four of the modules introduced in Chapter 2: the loop detection module, the execution trace indexing module, the heap allocation monitor, and the call stack tracking module. We present the preparation step in Section 3.4.1.

The core of the process is the *buffer deconstruction* step. The intuition behind buffer deconstruction is that the message field tree for the sent message is the inverse of the structure of the output buffer, which holds the message when is about to be sent on the network. Thus, deconstructing the output buffer into the memory buffers that were used to fill it with data reveals the message field tree of the sent message. This happens because programs store fields in memory buffers and construct the messages to be sent by combining those buffers together. Figure 3.9 shows the deconstruction of the output buffer holding the message in Figure 3.7. Note how Figure 3.9 is the upside-down version of Figure 3.7. Buffer deconstruction is implemented as a backward pass over an execution trace. It outputs a message field tree with an empty field attribute list for each node (except the field range). We present buffer deconstruction in Section 3.4.2.

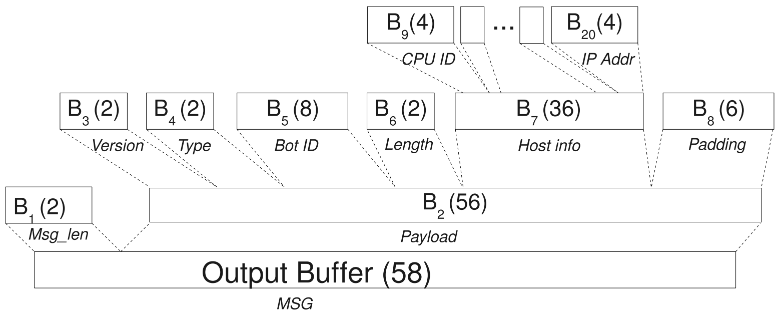


Figure 3.9: Buffer deconstruction for the MegaD message in Figure 3.7. Each box is a memory buffer starting at address B_x with the byte length in brackets. Note the similarity with the upside-down version of Figure 3.7.

Finally, the *field attribute inference* step identifies length fields, delimiters, field sequences, variable-length fields, and fixed-length fields. The information on those protocol elements is used to fill the field attributes in the message field tree. We present field attribute inference in Section 3.4.3.

3.4.1 Preparation

During preparation, a forward pass over the execution trace is made collecting information needed by buffer deconstruction and field attribute inference. Preparation uses four of the modules introduced in Chapter 2: the execution trace indexing module, the call stack tracking module, the loop detection module, and the heap allocation monitor. It uses the trace indexing module to build a trace index that enables random access to the execution trace, needed by buffer deconstruction to scan the execution trace backwards. It uses the call stack tracking module to produce a function that given a instruction in the trace returns the function nesting when the instruction was executed, also needed by buffer deconstruction. It uses the loop detection module to extract information about the loops in the execution trace, needed by field attribute inference. Buffer deconstruction also needs information on whether two different writes to the same memory address correspond to the same memory buffer, since memory locations in the stack (and occasionally in the heap) may be reused for different buffers. Buffer liveness information is gathered during preparation using the heap allocation monitor for heap buffers, and using the call stack tracking module to extract information about which memory locations in the stack are freed when the function returns.

3.4.2 Buffer Deconstruction

Buffer deconstruction is a recursive process. In each iteration it deconstructs a given memory buffer into a list of other memory buffers that were used to fill it with data. The process starts with the output buffer and recurses until there are no more buffers to deconstruct. Each memory buffer that forms the output buffer (and, recursively, the memory buffers that form them) corresponds to a field in the message field tree. At the end of each iteration, for each memory buffer used to construct the current buffer, a field is added into the message field tree. For example, the output buffer in Figure 3.9 holds the message in Figure 3.7 before it is sent over the network. Deconstructing this output buffer returns a sequence of two buffers that were used to fill it with data: a 2-byte buffer starting at offset zero in the output buffer (B_1) and a 56-byte buffer starting at offset 2 in the output buffer (B_2). Correspondingly, a field with range [0:1] and another one with range [2:57] are added to the message field tree. These two fields correspond to the *Msg-Length* and the *Encrypted Payload* fields in Figure 3.7.

Note that buffer deconstruction works at the binary level where a memory buffer is just a sequence of consecutive memory locations that were allocated in the same execution context. Thus, when any variable (e.g., an integer) is moved into memory (e.g., passed by value in a function call) it becomes a memory buffer. Buffer deconstruction has two parts. First, for each byte in the given buffer it builds a *dependency chain*. Then, using the dependency chains and the information collected in the preparation step, it deconstructs the given buffer. The input to each buffer deconstruction iteration is a buffer defined by its start address in memory, its length, and the instruction number in the trace where the buffer was last written. The start address and length of the output buffer are obtained from the arguments of the function that sends the data over the network (or the encryption function). The instruction number to start the analysis corresponds to the first instruction in the send (or encrypt) function. In the remainder of this section we introduce what program locations and dependency chains are and present how they are used to deconstruct the output buffer.

Program locations. We define a *program location* to be a one-byte-long storage unit in the program's state. We consider four types of locations: *memory locations*, *register locations*, *immediate locations*, and *constant locations*, and focus on the address of those locations, rather than on its content. Each memory byte is a memory location indexed by its address. Each byte in a register is a register location, for example, there are 4 locations (i.e., bytes) in the 32-bit EAX register: the lowest byte is EAX(0) and corresponds to the AL register, EAX(1) corresponds to the AH register, and EAX(2) and EAX(3) correspond to the higher two bytes in the register. An immediate location corresponds to a byte from an immediate in the code section of some module, indexed by the offset of the byte with respect to the beginning of the module. Constant locations represent the output of some instructions that have constant output. For example, one common instruction is to XOR one register against itself (e.g., *xor %eax, %eax*), which clears the register. Dispatcher recognizes a number of such instructions and makes each byte of its output a constant location.

Dependency chains. A dependency chain for a program location is the sequence of *write operations* that produced the value of the location at a certain point in the program. A write operation comprises the instruction number at which the write occurred, the destination location (i..e, the location that was written), the source location (i.e., the location that was read), and the offset of the written location with respect to the beginning of the output buffer. Figure 3.10 shows the dependency chains for the B_2 buffer (the one that holds the encrypted payload) in Figure 3.9. In the figure, each box represents a write operation, and each sequence of vertical boxes represents the dependency chain for one location in the buffer.

The dependency chain is computed in a backward pass starting at the given instruction number. We stop building the dependency chain at the first write operation for which the source location is: 1) an immediate location, 2) a constant location, 3) a memory location, or 4) an unknown location. We describe these four stop conditions next.

If the source location is part of an immediate or part of the output from some constant output instruction, then there are no more dependencies and the chain is complete. This is the case for the first four bytes of B_2 in Figure 3.10. The reason to stop at a source memory location is that we want to understand how a memory buffer has been constructed from other memory buffers. After deconstructing the given buffer, Dispatcher recurses on the buffers that form it. For example, in Figure 3.10 the dependency chains for locations *Mem(A*+4*)* through *Mem(A*+11*)* contains only one write operation because the source location is another memory location. Dispatcher will then create a new dependency chain for buffer *Mem(B)* through *Mem(B*+7*)*. When building the dependency chains, Dispatcher only handles a small subset of x86 instructions which simply move data around, without modifying it. This subset includes move instructions (*mov,movs*), move with zero-extend instructions (*movz*), push and pop instructions, string stores (*stos*), and instructions that are used to convert data from network to host order and vice versa such as exchange instructions (*xchg*), swap instructions (*bswap*), or right shifts that shift entire bytes (e.g., *shr $0x8,%eax*). When a write operation is performed by any other instruction, the source is considered unknown and the dependency chain stops. Often, it is enough to stop the dependency chain at such instructions, because the program is at that point performing some operation on the field (e.g., an arithmetic operation) as opposed to just moving the content around. Since programs operate on leaf fields, not on records, then at that point of the chain we have already recursed up to the corresponding leaf field in the message field tree. For example, in Figure 3.10 the dependency chains for the last two bytes stop at the same *add* instruction. Thus, both source locations are unknown. Note that those locations correspond to the length field in Figure 3.7. The fact that the program is increasing the length value indicates that the dependency chain has already reached a leaf field.

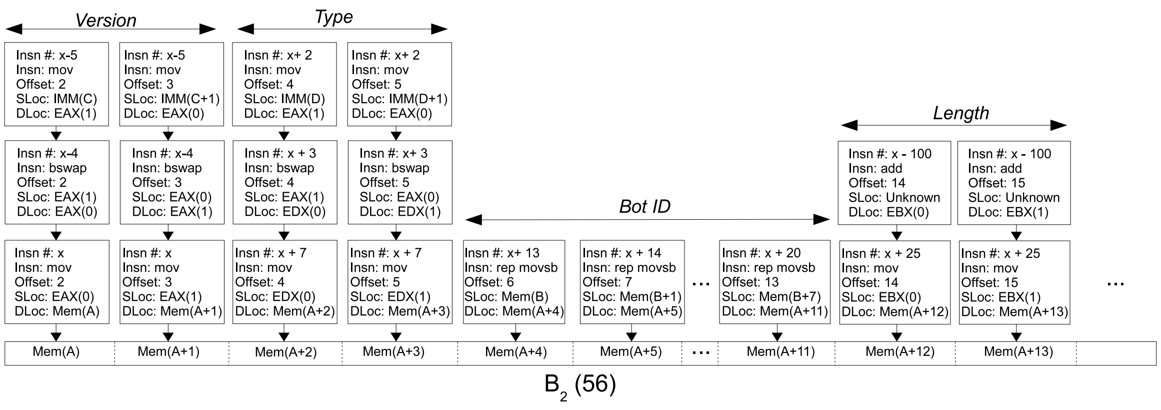

Version
Type
Length
Bot ID
Insn #: x-5 Insn: mov Offset: 2 SLoc: IMM(C) DLoc: EAX(1)
Insn #: x-5 Insn: mov Offset: 3 SLoc: IMM(C+1) DLoc: EAX(0)
Insn #: x+ 2 Insn: mov Offset: 4 SLoc: IMM(D) DLoc: EAX(1)
Insn #: x+ 2 Insn: mov Offset: 5 SLoc: IMM(D+1) DLoc: EAX(0)
Insn #: x-4 Insn: bswap Offset: 2 SLoc: EAX(1) DLoc: EAX(0)
Insn #: x-4 Insn: bswap Offset: 3 SLoc: EAX(0) DLoc: EAX(1)
Insn #: x + 3 Insn: bswap Offset: 4 SLoc: EAX(1) DLoc: EDX(0)
Insn #: x+ 3 Insn: bswap Offset: 5 SLoc: EAX(0) DLoc: EDX(1)
Insn #: x - 100 Insn: add Offset: 14 SLoc: Unknown DLoc: EBX(0)
Insn #: x - 100 Insn: add Offset: 15 SLoc: Unknown DLoc: EBX(1)
Insn #: x Insn: mov Offset: 2 SLoc: EAX(0) DLoc: Mem(A)
Insn #: x Insn: mov Offset: 3 SLoc: EAX(1) DLoc: Mem(A+1)
Insn #: x + 7 Insn: mov Offset: 4 SLoc: EDX(0) DLoc: Mem(A+2)
Insn #: x + 7 Insn: mov Offset: 5 SLoc: EDX(1) DLoc: Mem(A+3)
Insn #: x+ 13 Insn: rep movsb Offset: 6 SLoc: Mem(B) DLoc: Mem(A+4)
Insn #: x + 14 Insn: rep movsb Offset: 7 SLoc: Mem(B+1) DLoc: Mem(A+5)
...
Insn #: x + 20 Insn: rep movsb Offset: 13 SLoc: Mem(B+7) DLoc: Mem(A+11)
Insn #: x + 25 Insn: mov Offset: 14 SLoc: EBX(0) DLoc: Mem(A+12)
Insn #: x + 25 Insn: mov Offset: 15 SLoc: EBX(1) DLoc: Mem(A+13)
...
Mem(A)
Mem(A+1)
Mem(A+2)
Mem(A+3)
Mem(A+4)
Mem(A+5)
...
Mem(A+11)
Mem(A+12)
Mem(A+13)
B2 (56)

Figure 3.10: Dependency chain for B_2 in Figure 3.9. The start address of B_2 is A.

Extracting the buffer structure. We call the source location of the last element in the dependency chain of a buffer location its *source*. We say that two source locations belong to the same source buffer if they are contiguous memory locations (in either ascending or descending order) and the liveness information states that none of those locations has been freed between their corresponding write operations. If the source locations are not in memory (e.g., register, immediate, constant or unknown location), they belong to the same buffer if they were written by the same instruction (i.e, same instruction number).

To extract the structure for the given buffer Dispatcher iterates on the buffer locations from the buffer start to the buffer end. For each buffer location, Dispatcher checks whether the source of the current buffer location belongs to the same source buffer as the source of the previous buffer location. If they do not, then it has found a boundary in the structure of the buffer. The structure of the given buffer is output as a sequence of ranges that form it, where each range states whether it corresponds to a source memory buffer.

For example, in Figure 3.10 the source locations for *Mem(A*+4*)* and *Mem(A*+5*)* are contiguous locations *Mem(B)* and *Mem(B*+1*)* but the source locations for *Mem(A*+11*)* and *Mem(A*+12*)* are not contiguous. Thus, Dispatcher marks location *Mem(A*+12*)* as the beginning of a new range. Dispatcher finds 6 ranges in B_2. The first four are shown in Figure 3.10 and marked with arrows at the top of the figure. Since only the third range originates from another memory buffer, that is the only buffer that Dispatcher will recurse on to reconstruct. The last two ranges correspond to the *Host Info* and *Padding* fields in Figure 3.7 and are not shown in Figure 3.10.

Once the buffer structure has been extracted, Dispatcher uses the correspondence between buffers and fields in the analyzed message to add one field to the message field tree per range in the buffer structure using the offsets relative to the output buffer. In Figure 3.10 it adds four new fields that correspond to the *Version*, *Type*, *Bot ID*, and *Length* in Figure 3.7. Note that buffer deconstruction focuses on the source and tail of the dependency chain, ignoring the possibly multiple instructions that may move a byte of data across different registers before writing it to a memory location. There are two reasons why we ignore those internal instructions in the chain. One is that registers are only temporary storage locations, the other one is that general-purpose registers have a maximum length (i.e., 4 bytes in a 32-bit architecture) that is smaller than the size of many variable-length fields. Thus, if those intermediate instructions where accounted for, the technique would split large fields into multiple smaller fields.

3.4.3 Field Attributes Inference

The message field tree built by buffer deconstruction captures the hierarchical structure of the output message, but does not contain field attributes other than the field range. Field attributes convene information that can be generalized from this message to other messages of the same type such as if a field is fixed-length or variable-length or inter-field relationships such as if a field represents the length of another target variable-length field. Similar to the need for buffer deconstruction, new field attribute inference techniques are also needed for sent messages. Next, we propose field attribute inference techniques designed to identify different protocol elements in sent messages. These techniques differ but share common intuitions with the techniques used for received messages: both try to capture fundamental properties of the protocol elements.

Length fields. We use three different techniques to identify length fields in sent messages. The intuition behind the techniques is that length fields can be computed either by incrementing a counter as the program iterates on the field, or by subtracting pointers to the beginning and the end of the buffer. The intuition behind the first two techniques is that those arithmetic operations translate into an unknown source at the end of the dependency chains for the buffer locations corresponding to the length field. When a dependency chain ends in an unknown source, Dispatcher checks whether the instruction that performs the write is inside a known function that computes the length of a string (e.g., *strlen*) or is a subtraction of pointers to the beginning and end of the buffer. The third technique tries to identify counter increments that do not correspond to well-known string length functions. For each buffer it uses the loop information to identify if most writes to the buffer[6] belong to the same loop. If they do, then it uses the techniques in [188] to extract the loop induction variables. For each induction variable it computes the dependency chain and checks whether it intersects the dependency chains from any output buffer locations that precede the locations written in the loop

[6]Many memory move functions are optimized to move 4 bytes at a time in one loop and use separate instructions or loops to move the remaining bytes.

(since a length field always precedes its target field). Any intersecting location is part of the length field for the field processed in the loop.

Delimiters. Delimiters are constants and it is difficult to differentiate them from other constants in the sent message. The technique to identify delimiters looks for constants that appear multiple times in the same message or appear at the end of multiple messages in the same session (three appearances are required). Constants are identified using the taint information introduced by tainting the memory regions containing the program and DLLs shipped it. If the delimiters come from the data section, they can also be identified by checking whether the source address of all instances of the constant comes from the same buffer.

Variable-length fields. Fields that precede a delimiter and target fields for previously identified length fields are marked as variable-length fields. Fields derived from semantic sources that are known to be variable-length such as file data are also marked as variable-length. All other fields are marked as fixed-length. Note that some fields that a protocol specification would define as variable-length may encode always the same fixed-length data in a specific implementation. For example the *Server* header is variable-length based on the HTTP specification. However, a given HTTP server implementation may have hard-coded the *Server* string in the binary, making the field fixed-length for this implementation. Leveraging the availability of multiple implementations of the same protocol could help identify such cases.

Field sequences. The intuition behind identifying field sequences is that they are written in loops, one field at a time. The technique to identify sequences searches for loops that write multiple consecutive fields. For each loop, it adds to the message field tree one record field with the range being the combined range of all the consecutive fields written in the loop and with a *Sequence* field semantics value. It also adds one field per range of bytes written in each iteration of the loop.

3.5 Field Semantics Inference

In this section we present our techniques to identify the field semantics of both received and sent messages. The intuition behind our type-inference-based techniques is that many functions and instructions used by programs contain rich semantic information. We can leverage this information to infer field semantics by monitoring if received network data is used at a point where the semantics are known (i.e., semantics sinks), or if data to be sent to the network has been derived from data with known semantics (i.e., semantics sources). Such semantics inference is very general and can be used to identify a broad spectrum of field semantics including timestamps, filenames, hostnames, ports, IP addresses, and many others. The semantic information of those functions and instructions is publicly available in their prototypes, which describe their goal as well as the semantics of its inputs and outputs. Function prototypes can be found, for example, at the Microsoft Developer Network [138] or the standard C library documentation [93]. For instructions, one can refer to the system manufacturers' manuals.

Techniques. For *received* messages, Dispatcher uses taint propagation to monitor if a sequence of bytes from the received message is used in the arguments of some selected function calls and instructions, for which the system has been provided with the function's prototype. The sequence of bytes in the received message can then be associated with the semantics of the arguments as defined in the prototype. For example, when a program calls the *connect* function Dispatcher uses the function's prototype to check if any of the arguments on the stack is tainted. The function's prototype tells us that the first argument is the socket descriptor, the second one is an address structure that contains the IP address and port of the host to connect to, and the third one is the length of the address structure. If the memory locations that correspond to the IP address to connect to in the address structure are tainted from four bytes in the input, then Dispatcher can infer that those four bytes in the input message (identified by the offset in the taint information) form a field that contains an IP address to connect to. Similarly, if the memory locations that correspond to the port to connect to have been derived from two bytes in the input message, it can identify the position of the port field in the input message.

For *sent* messages, Dispatcher taints the output of selected functions and instructions using a unique source identifier and offset pair. For each tainted sequence of bytes in the output buffer, Dispatcher identifies from which taint source the sequence of bytes was derived. The semantics of the taint source (return values) are given by the function's or instruction's prototype, and can be associated to the sequence of bytes. For example, if a program uses the *rdtsc* instruction, we can leverage the knowledge that it takes no input and returns a 64-bit output representing the current value of the processor's time-stamp counter, which is placed in registers EDX:EAX [91]. Thus, at the time of execution when the program uses *rdtsc*, Dispatcher taints the EDX and EAX registers with a unique source identifier and offset pair. This pair uniquely labels the taint source to be from *rdtsc*, and the offsets identify each byte in the *rdtsc* stream (offsets 0 through 7 for the first use).

A special case of this technique is *cookie* inference. A cookie represents data from a received message that propagates unchanged to the output buffer (e.g., session identifiers). Thus, a cookie is simultaneously identified in the received and sent messages.

Implementation. To identify field semantics Dispatcher uses an input set of function and instruction prototypes. By default, Dispatcher includes over one hundred functions and a few instructions

Field Semantics	**Received**	**Sent**
Cookies	yes	yes
IP addresses	yes	yes
Error codes	no	yes
File data	no	yes
File information	no	yes
Filenames	yes	yes
Hash / Checksum	yes	yes
Hostnames	yes	yes
Host information	no	yes
Keyboard input	no	yes
Keywords	yes	yes
Length	yes	yes
Padding	yes	no
Ports	yes	yes
Sequences	no	yes
Registry data	no	yes
Sleep timers	yes	no
Stored data	yes	no
Timestamps	no	yes

Table 3.3: Field semantics identified by Dispatcher for both received and sent messages. Stored data represents data received over the network and *written* to the filesystem or the Windows registry, as opposed to data *read* from those sources.

for which we have already added the prototypes by searching online repositories. To identify new field semantics and their corresponding functions, we examine the external functions called by the program in the execution trace. Table 3.3 shows the field semantics that Dispatcher can infer from received and sent messages using the predefined functions.

Keywords. An important field semantic is keywords. Keywords are protocol constants that appear in network messages and are known a priori to the implementation. They are useful to create protocol signatures to detect services running on non-standard ports and mapping traffic to applications [84, 131]. Our intuition to identify keywords in received messages is that similar to delimiters, the program compares the keywords against the received application data. Dispatcher locates the keywords in the received message by analyzing the *successful comparisons* between tainted and untainted data, using comparison operations as the semantics sinks. The technique comprises two steps. The first step is identical to the first step in the delimiter identification technique presented in Section 3.3.1, that is, to populate the token table. The second step differs in that it focuses on the successful comparisons, rather than all the comparisons. It consists of scanning in ascending position order the columns in the token table. For each position, if we find a successful comparison, then we concatenate the token that was compared to the position to the current keyword. If no successful comparison is found at the current position, we store the current keyword and start a new keyword. We also break the current keyword and start a new one if the keyword crosses a field boundary as defined by the message field tree. This technique is general, in that it does not assume that the multiple bytes that form the keyword appear together in the code or that they are used sequentially. For example, using the token table shown in Figure 3.4, Dispatcher identifies two HTTP keywords: “GET” at positions [0:2] and “HTTP” at positions [16:19].

To identify keywords in sent messages, Dispatcher taints the memory region that contains the module (and DLLs shipped with the main binary) with a specific taint source, effectively tainting both immediates in the code section as well as data stored in the data section. Locations in the output buffer tainted from this source are considered keywords.

3.6 Handling Encryption

Our protocol reverse-engineering techniques work on unencrypted data. Thus, when reversing encrypted protocols we need to address two problems. First, for received messages, we need to identify the buffers holding the unencrypted data at the point that the decryption has finished since buffers may only hold the decrypted data for a brief period of time. Second, for sent messages, we need to identify the buffers holding the unencrypted data at the point that the encryption is about to begin. Once the buffers holding the unencrypted data have been identified, protocol reverse-engineering techniques can be applied on them, rather than on the messages received or about to be sent.

Recent work has looked at the problem of reverse-engineering the format of received encrypted messages [221, 128]. Since the application needs to decrypt the data before using it, those approaches monitor the application’s processing of the encrypted message and locate the buffers that contain the decrypted data at the point that the decryption has finished by identifying the output buffers of functions with a high ratio of arithmetic and bitwise instructions. Those approaches do not address the problem of finding the buffers holding the unencrypted data before it is encrypted, which is also required in our case. We have developed two different approaches to identify encoding functions. In this section we present extensions to the technique presented in ReFormat [221], which flags encoding functions as functions with a high ratio of arithmetic and bitwise instructions. Then, in Chapter 8 we present a different technique to identify encoding functions, which flags encoding functions as functions that highly mix their inputs.

Next, we describe our two extensions to the technique presented in ReFormat [221]. First, ReFormat can only handle applications where there exists a single boundary between decryption and normal protocol processing. However, multiple such boundaries may exist. As shown in Figure 3.7

MegaD messages comprise two bytes with the message length, followed by the encrypted payload. After checking the message length, a MegaD bot will decrypt 8 bytes from the encrypted payload and process them, then move to the next 8 bytes and process them, and so on. In addition, some messages in MegaD also use compression and the decryption and decompression operations are interleaved. Thus, there is no single program point where all data in a message is available unencrypted and uncompressed. Consequently, we extend the technique to identify every *instance* of encryption, hashing, compression, and obfuscation, which we generally term *encoding functions*. Second, ReFormat was not designed to identify the buffers holding the decoded (unencrypted) data before encoding (encryption). Thus, we extend the technique to also cover this case. We present the generalized technique next.

Identifying encoding functions. To identify every instance of an encoding function we have simplified the process in ReFormat by removing the cumulative ratio of arithmetic and bitwise instructions for the whole trace (since we are interested in the ratio for each function), the use of tainted data, and the concept of leaf functions. The extended technique applies the intuition in ReFormat that the decryption process contains an inordinate number of arithmetic and bitwise operations to encoding functions. It makes a forward pass over the input execution trace using the call stack tracking module. For each function instance, it computes the ratio between the number of arithmetic and bitwise operations over the total number of instructions in the function. The ratio includes only the function's own instructions. It does not include instructions belonging to any called functions. Any function instance that executes a minimum number of instructions and has a ratio larger than a pre-defined threshold is flagged as an instance of an encoding function. The minimum number of instructions is needed because the ratio is not meaningful for functions that execute few instructions. In our experiments we set the minimum number of instructions to 20. We have experimentally set the threshold to 0.55 by training with a number of known encoding functions and selecting a threshold that minimizes the number of false negatives. We evaluate the technique in Section 3.7.3.

Identifying the buffers. To identify the buffers holding the unencrypted data before encryption we compute the *read set* for the encryption routine, the set of locations read inside the encryption routine before being written. The read set for the encryption routine includes the buffers holding the unencrypted data, the encryption key, and any hard-coded tables used by the routine. We can differentiate the buffers holding the unencrypted data because their content varies between multiple instances of the same function. To identify the buffers holding the unencrypted data after decryption we compute the *write set* for the decryption routine, the set of locations written inside the decryption routine and read later in the trace. We detail the read and write set extraction as part of our interface identification technique in Chapter 4.

3.7 Evaluation

In this section we evaluate our techniques on the previously undocumented C&C protocol used by the MegaD botnet, as well as a number of open protocols. MegaD is a prevalent spamming botnet first observed in 2007 and credited at its peak with responsibility for sending a third of the world's spam [129]. We use MegaD's proprietary and encrypted C&C protocol as a real-world test of our techniques. We use the open protocols to evaluate our techniques against a known ground truth.

3.7.1 Evaluation on MegaD

MegaD uses a proprietary, encrypted, binary protocol that has not been previously analyzed. Our MegaD evaluation has two parts. We first describe the information obtained by Dispatcher on the C&C protocol used by MegaD, and then show how the information extracted by Dispatcher can be used to rewrite a C&C dialog.

MegaD C&C Protocol. The MegaD C&C protocol uses TCP for transport on either port 80 or 443[7]. It employs a proprietary encryption algorithm instead of the SSL routines for HTTPS commonly used on port 443. Some MegaD bots use port 80 and others use 443 but the encryption and protocol grammar are identical regardless of the port.

A MegaD bot communicates with four types of C&C servers: *Master Servers (MS)*, *Drop Servers (DS)*, *Template Servers (TS)*, and *SMTP Servers (SS)*. The four server types are illustrated in Figure 3.11. The botmaster uses the master servers to distribute commands to the bots. Bots locate a master server using a rendezvous algorithm, based on domain names hard-coded in the bot binaries. A bot employs pull-based communication using MegaD's C&C protocol to periodically probe the master server with a request message to which the server replies with a sequence of messages carrying authentication information and a command. The bot performs the requested action and returns the results to the master server. Drop servers distribute new binaries. A bot locates a drop server by receiving a message from its master server containing a URL specifying a file to download through HTTP. Template servers distribute the spam templates that bots use to construct spam. A bot locates a template server via a message from the master server specifying the address and port to contact. Again, communication proceeds in a pull-based fashion using MegaD's custom C&C protocol. SMTP servers play two distinct roles. First, bots check their spam-sending capabilities by sending them a test email using the standard SMTP protocol. A bot locates the SMTP server for this testing via a message from the master server specifying the server's hostname. Second, bots notify an SMTP server after downloading a new spam template and prior to commencing to spam.

[7]Malware often uses TCP ports 80 and 443 for their communication because those ports are often open in firewalls to enable Web browsing.

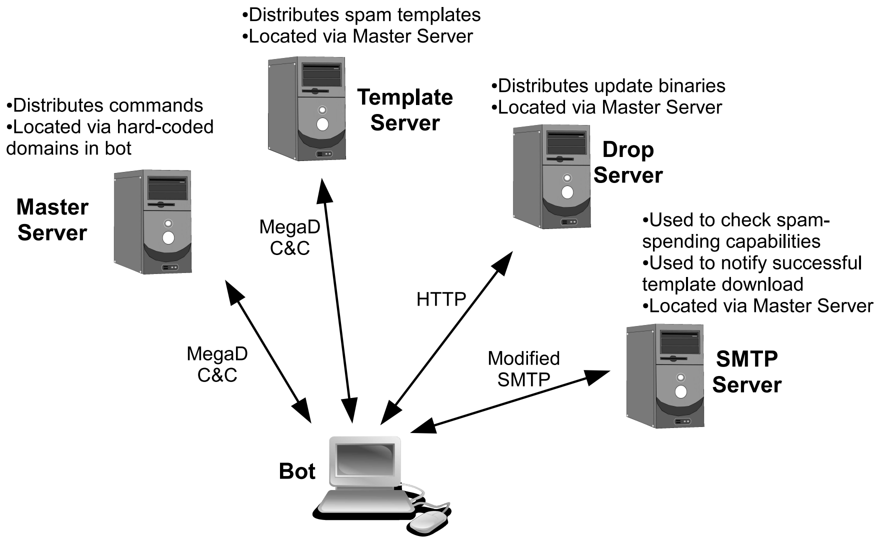

•Distributes spam templates
•Located via Master Server
Template Server
•Distributes commands
•Located via hard-coded domains in bot
Master Server
•Distributes update binaries
•Located via Master Server
Drop Server
•Used to check spam-spending capabilities
•Used to notify successful template download
•Located via Master Server
SMTP Server
MegaD C&C
MegaD C&C
HTTP
Modified SMTP
Bot

Figure 3.11: The four server types that a MegaD bot communicates with. The figure shows for each server the communication protocol used between the bot and the server, the main use of the server, and how the bot locates the server.

A bot locates the SMTP server used for template download notification via a control parameter in the spam template. The notification uses a modified SMTP protocol. Instead of sending the usual SMTP "HELO <hostname>" message, the bot sends a special "HELO 1" message and closes the connection.

Message format. We capture two MegaD C&C network traces by running the binary in a contained environment that forwards C&C traffic but blocks any other traffic from the bot (e.g., spam traffic). Our MegaD C&C traces contain 15 different messages (7 received and 8 sent by the bot). Using Dispatcher, we have extracted the message field tree for messages on both directions, as well as the associated field semantics. All 15 messages follow the structure shown in Figure 3.7 with a 2-byte message length followed by an encrypted payload. The payload, once decrypted, contains a 2-byte field that we term "version" as it is always a keyword of value 0x100 or 0x1, followed by a 2-byte message type field. The structure of the remaining payload depends on the message type. To summarize the protocol grammar we have used the output of Dispatcher to write a BinPac grammar [173] that comprises all 15 messages. Field semantics are added as comments to the grammar. Appendix A presents the MegaD protocol grammar.

To the best of our knowledge, we are the first to document the C&C protocol employed by MegaD. Thus, we lack ground truth to evaluate our grammar. To verify the grammar's accuracy, we use another execution trace that contains a different instance of one of the analyzed dialogs. We dump the content of all unencrypted messages and try to parse the messages using our grammar. For this, we employed a stand-alone version of the BinPac parser included in Bro [175]. Using our grammar, the parser successfully parses all MegaD C&C messages in the new dialog. In addition, the parser throws an error when given messages that do not follow the MegaD grammar.

Field attribute inference. The 15 MegaD messages contain no delimiters or arrays. They contain two variable-length fields that use length fields to mark their boundaries: the compressed spam-related information (i.e., template and addresses) received from the spam server, and the host information field in Figure 3.7. Both the length fields and variable-length fields are correctly detected by Dispatcher. The only attributes that Dispatcher misses are the message length fields on sent messages because they are computed using complex pointer arithmetic that Dispatcher cannot reason about. In particular, the message length is computed by subtracting the pointers to the end and beginning of the message, but then this result goes through a sequence of arithmetic and bitwise instructions that encodes the final number of bytes in value in the field.

Field semantics. Dispatcher identifies 11 different field semantics over the 15 messages: IP addresses, ports, hostnames, length, sleep timers, error codes, keywords, cookies, stored data, padding and host information. There are only two fields in the MegaD grammar for which Dispatcher does not identify their semantics. Both of them happen in received messages: one of them is the message type, which we identify by looking for fields that are compared against multiple constants in the execution and for which the message format varies depending on its value. The other corresponds to an integer whose value is checked by the program but apparently not used further. Note that we identify some fields in sent messages as keywords because they come from immediates and constants in the data section. We cannot identify exactly what they represent because we do not see how they are used by the C&C server.

Rewriting a MegaD dialog. To show how our grammar enables live rewriting, we run a live MegaD bot inside our analysis environment, which is located in a network that filters all outgoing SMTP connections for containment purposes. In a first dialog, the C&C server sends the command to the bot ordering to test for spam capability using a given Spam test server. The analysis network blocks the SMTP connection causing the bot to send an error message back to the C&C server, to communicate that it cannot send spam. No more spam-related messages are received by the bot. Then, we start a new dialog where at the time the bot calls the encrypt function to encrypt the error message, we stop the execution, rewrite the encryption buffer with the message that indicates success, and let the execution continue[8]. After the rewriting the bot keeps receiving the spam-related

[8]The size of both messages is the same once padding is accounted for, thus we can reuse the allocated buffer.

Program	**Version**	**Protocol**	**Type**	**Guest OS**
Apache [4]	2.2.1	HTTP	Server	Windows XP
BIND [14]	9.6.0	DNS	Server	Windows XP
FileZilla [67]	0.9.31	FTP	Server	Windows XP
Pidgin [177]	2.5.5	ICQ	Client	Windows XP
Sambad [186]	3.0.24	SMB	Server	Linux Fedora Core 5
TinyICQ [208]	1.2	ICQ	Client	Windows XP

Table 3.4: Different programs used in our evaluation on open protocols.

		Wireshark		**Dispatcher**		**Errors**			
Protocol	**Message Type**	$\lvert L_W \rvert$	$\lvert H_W \rvert$	$\lvert L_D \rvert$	$\lvert H_D \rvert$	$\lvert E_W^L \rvert$	$\lvert E_D^L \rvert$	$\lvert E_W^H \rvert$	$\lvert E_D^H \rvert$
HTTP	GET reply	11	1	22	0	11	1	0	1
	POST reply	11	1	22	0	11	1	0	1
DNS	A reply	27	4	28	0	1	0	0	4
FTP	Welcome0	2	1	3	1	1	0	0	0
	Welcome1	2	1	3	1	1	0	0	0
	Welcome2	2	1	3	1	1	0	0	0
	USER reply	2	1	3	1	1	1	0	0
	PASS reply	2	1	2	0	1	1	0	1
	SYST reply	2	1	2	0	1	1	0	1
ICQ	New connection	5	0	5	0	0	0	0	0
	AIM Sign-on	11	3	15	3	5	0	0	0
	AIM Logon	46	15	46	15	0	0	0	0
Total		123	30	154	22	34	5	0	8

Table 3.5: Comparison of the message field tree for sent messages extracted by Dispatcher and Wireshark 1.0.5. The ICQ client used is Pidgin. L_W and L_D are the set of leaf fields output by Wireshark and Dispatcher respectively, while H_W and H_D are the sets of record (hierarchical) fields. E_W^L and E_D^L denote the set of errors in leaf field output by Wireshark and Dispatcher, while E_W^H and E_D^H denote the set of errors in record fields.

messages, including the spam template and the addresses to spam, despite the fact that it cannot send any spam messages. Note that simply replaying the message that indicates success from a previous dialog into the new dialog does not work because the success message includes a cookie value that the C&C selects and may change between dialogs. In Chapter 4 we present our binary code reuse techniques and apply them to the cryptographic routines used to protect MegaD's C&C protocol. Using the extracted cryptographic routines in combination with MegaD's protocol grammar, we can perform this same rewriting experiment on a Network Intrusion Detection System (NIDS), rather than inside the execution monitor where the bot runs.

3.7.2 Evaluation on Open Protocols

In this section we evaluate our techniques on five open protocols: DNS, FTP, HTTP, ICQ, and SMB. To this end, we compare the output of Dispatcher with that of Wireshark [227] when processing 17 messages belonging to those 5 protocols. For each protocol we select at least one application that implements it, which we present in Table 3.4. For each protocol, we select a set of protocol messages. For HTTP we evaluate how the Apache server [4] processes a `HTTP GET` request for the file "index.html" and the reply generated by the server. For DNS we evaluate an A query to resolve the IP address of the domain "test.example.com" send to the BIND name server [14] and its corresponding reply. For FTP we analyze the sequence of messages sent by the FileZilla server [67] in response to a connection, as well as the messages sent when the username and password are received. For ICQ we analyze the messages in a login connection sent by the Pidgin client tool [177] and the responses from the server interpreted by the TinyICQ client tool [208]. For SMB, we analyze a `Negotiate Protocol Request` received by the Sambad open source server [186].

Message format. Wireshark is a network protocol analyzer containing manually written grammars (called dissectors) for a large variety of network protocols. Although Wireshark is a mature and widely-used tool, its dissectors have been manually generated and therefore are not completely error-free. Wireshark dissectors parse a message into a message field tree. The internal message field tree is not output in a visual representation by Wireshark but is accessible through the library functions. To compare the accuracy of the message format automatically extracted by Dispatcher to the manually written ones included in Wireshark, we analyze the message field tree output by both tools and manually compare them to the protocol specification. Thus, we can classify any differences between the output of both tools to be due to errors in Dispatcher, Wireshark, or both.

We denote the set of leaf fields and the set of records in the message field tree output by Wireshark as L_W (L stands for leaf) and H_W (H stands for hierarchical), respectively. L_D and H_D are the corresponding sets for Dispatcher. Table 3.5 shows the evaluation results for sent messages and Table 3.6 for received messages. For each protocol and message the tables show the number of leaf fields and records in the message field tree output by both tools as well as the result of the manual classification of its errors. Here, $|E_W^L|$ and $|E_D^L|$ represent the number of errors on leaf fields in the message field tree output by Wireshark and Dispatcher respectively. Similarly, $|E_W^H|$ and $|E_D^H|$ represent the number of errors on records.

The results show that Dispatcher outperforms Wireshark when identifying leaf fields. This result is mainly due to the inconsistencies between the different dissectors in Wireshark when identifying delimiters. Some dissectors do not add delimiters to the message field tree, some concatenate them to the variable-length field for which they mark the boundary, while others treat them as separate

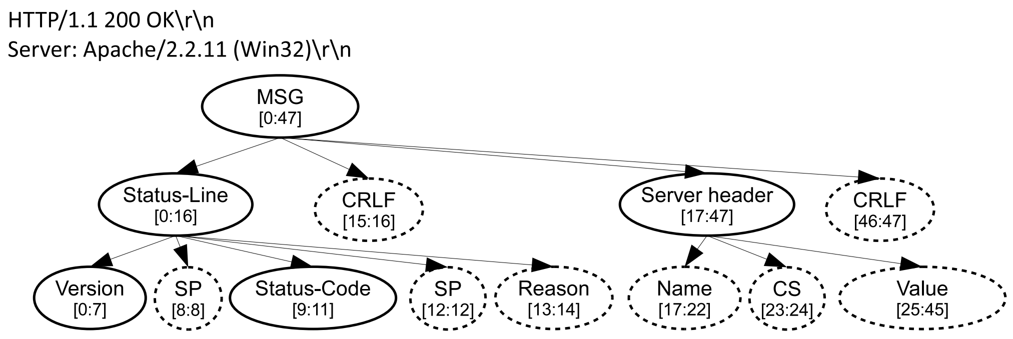


Figure 3.12: Message field tree for a simple HTTP response output by Wireshark. The dotted nodes are fields that Wireshark does not output.

		Wireshark		Dispatcher		Errors			
Protocol	**Message Type**	$\|L_W\|$	$\|H_W\|$	$\|L_D\|$	$\|H_D\|$	$\|E_W^L\|$	$\|E_D^L\|$	$\|E_W^H\|$	$\|E_D^H\|$
HTTP	GET request	13	2	40	10	27	2	8	0
DNS	A query	14	3	13	1	1	0	0	2
ICQ	New connection	38	11	36	11	0	2	0	0
	Close connection	13	3	10	3	0	3	0	0
SMB	Negotiate Protocol Request	48	16	39	11	9	6	0	5
Total		126	35	138	36	37	13	8	7

Table 3.6: Comparison of the message field tree for received messages extracted by Dispatcher and Wireshark 1.2.8. The ICQ client is TinyICQ.

fields. After checking the protocol specifications, we believe that delimiters should be treated as their own fields in all dissectors. Figure 3.12 illustrates some of the errors made by Wireshark. It shows the message field tree for a simple HTTP response output by Wireshark. The dotted nodes are missing nodes that Wireshark does not output, which include delimiters, the reason field and the children of the `Server header` field.

The results also show that Wireshark outperforms Dispatcher when identifying records. For sent messages, this is due to the program not using loops to write the arrays because the number of elements in the array is known or is small enough that the compiler has unrolled the loop. For example, if an array has only two elements, the source-level loop that processes the field iterates only twice and the compiler may decide to unroll the two iterations at the binary-level. Thus, at the binary level there is no loop that handles both records in the array and Dispatcher will flag them as separate fields rather than as two records of an array. For received messages it is often due to the loop that processes the record being missed by the detection because it executed only one iteration[9].

[9]Here we are using the dynamic loop detection method (See Section 2.4.1), which can only detect loops that complete

The two main sources of errors for Dispatcher when analyzing sent messages are: consecutive fields that are stored as a single string in the program binary and arrays that are not written using a loop. An example of consecutive fields stored as a unit by the application is the error in the *Status-Line* record of the HTTP reply message. The HTTP/1.1 specification [65] states that its format is: *Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF*, but Dispatcher considers the *Status-Code*, the delimiter, and the *Reason-Phrase* to belong to the same field because all three fields are stored as a single string in the server's data section, which is copied as a whole into the sent message. An example of a program processing an array without a loop is the *BIND* server processing separately the *Queries*, *Answers*, *Authoritative*, and *Additional* sections in the DNS reply. This introduces four errors in the results because Dispatcher cannot identify that they form an array.

The two main sources of errors for Dispatcher when analyzing received messages are: fields smaller than one byte and unused fields. An example of fields smaller than one byte are the fields that comprise the flags records in the DNS and SMB messages. Since Dispatcher works at the byte level it currently does not identify fields smaller than one byte. Unused fields are fields that the program only moves without performing any other operation on them. When two consecutive unused fixed-length fields are found, Dispatcher groups them as a single field introducing an error. For example, in the SMB *Negotiate Protocol Request* message, the *Process ID High*, *Signature*, *Reserved*, *Tree ID*, and *Process ID* fields are all grouped together by Dispatcher into a single unused field. These errors in sent and received messages highlight the fact that the message field tree extracted by Dispatcher is limited to the quality of the protocol implementation in the binary, and may differ from the specification.

Overall, Dispatcher and Wireshark achieve similar accuracy. Note that we do not claim that Dispatcher will always be as accurate as Wireshark, since we are only evaluating a limited number of protocols and messages. However, the results show that the accuracy of the message format automatically extracted by Dispatcher can rival that of Wireshark, without requiring a manually generated grammar.

Field Attribute Inference. The 17 messages contain 34 length fields, 73 delimiters, 133 variable-length fields, and 6 arrays. We have analyzed in detail the errors in the field attribute inference for sent messages. Dispatcher misses 8 length fields because their value is hard-coded in the program. Thus, their target variable-length fields are considered fixed-length. Out of the 43 delimiters in sent messages Dispatcher only misses one, which corresponds to a null byte marking the end of a cookie string that was considered part of the string. Dispatcher correctly identifies all other variable-length

a full iteration, i.e., where the backedge is taken.

Number of traces	Number of functions	False Positives	False Positive Rate
20	3,467,458 (16,852)	87 (9)	0.002%

Table 3.7: Evaluation of the detection of encoding functions. Values in parentheses represent the numbers of unique instances. False positives are computed based on manual verification.

fields in sent messages. Out of 3 arrays, Dispatcher misses one formed by the *Queries*, *Answers*, *Authoritative*, and *Additional* sections in the DNS reply, which BIND processes separately and therefore cannot be identified by Dispatcher.

Field semantics. Dispatcher correctly identifies all semantics in Table 3.3 except the 3 pointers in the DNS reply, used by the DNS compression method, which are computed using pointer arithmetic that Dispatcher cannot reason about.

3.7.3 Detecting Encoding Functions

To evaluate the detection of encoding functions presented in Section 3.6 we perform the following experiment. We obtain 25 execution traces from multiple programs that handle network data. Five of these traces process encrypted and compressed functions, four of them are from MegaD sessions and the other one is from Apache while handling an HTTPS session. MegaD uses its own encryption algorithm and the *zlib* library for compression and Apache uses SSL with AES and SHA-1[10]. The remaining 20 execution traces are from a variety of programs including three browsers processing the same plain HTTP response (Internet Explorer 7, Safari 3.1, and Google Chrome 1.0), a DNS server processing a received request (BIND), a Web server processing an HTTP GET request (AtpHttpd), the Microsoft SQL server processing a request for database information (MSSQL), as well as the RPC service embedded in Windows handling a directory listing. For all these 15 traces the inputs do not contain any checksums, encrypted or compressed data, so we believe they are free of encoding functions.

Dispatcher flags any function instances in the execution traces with at least 20 instructions and a ratio of arithmetic and bitwise instructions greater than 0.55 as encoding functions. To evaluate false negatives, we run Dispatcher on the Apache-SSL trace. Dispatcher correctly identifies all encoding functions. To evaluate false positives, we run Dispatcher on the 20 traces that do not contain encoding functions. The results are shown in Table 3.7. The 20 execution traces contain over 3.4 million functions calls from over 16,852 unique functions. Dispatcher flags 87 function instances as encoding functions, belonging to nine unique functions. Using function names and debugging information, we have been able to identify two out of those nine functions: `memchr`

[10]TLS-DHE-RSA with AES-CBC-256-SHA-1

and `comctl32.dll::TrueSaturateBits`. Based on these results, our technique correctly identifies all known encoding functions and has a false positive rate of 0.002%.

Next, we run Dispatcher on the four MegaD execution traces. Four unique encoding functions are identified. Three of them appear in all four execution traces: the decryption routine, the encryption routine, and a key generation routine that generates the encryption and decryption keys from a seed value in the binary before calling the encryption or decryption routines. In addition, in one execution trace Dispatcher flags a fourth function that corresponds to the *inflate* function in the *zlib* library, which is statically linked into the MegaD binary.

3.8 Related Work

Protocol reverse-engineering projects that enable interoperability of open solutions with proprietary protocols have existed for a long time. Those projects relied on manual techniques, which are slow and costly [148, 210, 121, 72, 89]. Automatic protocol reverse-engineering techniques can be used, among other applications, to reduce the cost and time associated with these projects.

Protocol format. Early work on automatic protocol format reverse-engineering takes as input network traffic. Among these approaches, the Protocol Informatics project pioneers the use of sequence alignment algorithms [10] and Discoverer proposes a related, but improved, technique that first tokenizes messages into a sequence of binary and text tokens, then clusters similar token sequences and finally merges similar sequences using type-based sequence alignment [52]. Approaches based on network traffic are useful when a program that implements the protocol is not available. However, they cannot reverse encrypted protocols and are limited by the lack of protocol information in network traffic. Leveraging a program that implements the protocol significantly improves the quality of the reversed-engineered format.

Lim et al. [123] use static analysis on program binaries to extract the format from files and application data output by a program. Their approach requires the user to input the prototype of the functions that write data to the output buffer. This information is often not available, e.g., when the functions used to write data are not exported by the program. Their static analysis approach requires sophisticated analysis to deal with indirection and cannot handle packed binaries such as MegaD. Also, they do not extract the format of received messages or infer field semantics.

In Polyglot, we propose a dynamic binary analysis approach for extracting the message format of received messages that does not require any a priori knowledge about the program or the protocol and can effectively deal with indirection and packed binaries [30]. Dynamic binary analysis techniques are also used in follow-up work that extracts the hierarchical message format [229, 124, 54, 125]. We detail those works next.

In Autoformat, the authors propose techniques to extract the message field tree of received messages and to identify field sequences [124]. Their technique groups together consecutive positions in the input message that are processed by the same function. However, a function may parse multiple fields, e.g., when parsing two consecutive fixed-length fields in a binary protocol. Their output message field tree captures the hierarchical structure of the message but contains no field attributes. Thus, it cannot be used to generalize across multiple instances of the same message. To identify field sequences their technique looks for input positions that with similar execution history, i.e., that have been processed by the same functions.

Wondracek et al. propose techniques to extract the message format of received messages and use hierarchical sequence alignment to identify optional fields, alternation, and sequences of identically-structured records [229]. Their message format captures the hierarchical field structure and contains field attributes such as length fields and delimiters, which can be used to generalize across messages. When identifying leaf fields, they break an input chunk that is not a delimiter, length field, or variable-length field, into individual bytes and thus may miss identifying multi-byte fixed-length fields.

In Tupni the authors propose techniques to identify field sequences and to aggregate information from multiple messages [54]. Their field sequence identification technique groups together input positions that are handled in the same loop iteration. Tupni identifies fields with the same type across different messages by comparing the set of instructions that operate on the fields. Then, it aligns fields based on their types using the technique in [52].

Lin and Zhang develop techniques to extract the input syntax tree of inputs with top-down or bottom-up grammars [125]. Their technique for top-down grammars has the same scope as the works above and assumes that program control dependence follows program parsing. However, many programs may not satisfy this assumption, e.g., they may backtrack to previously scanned fields or they may perform error checks that do not reveal input structure but modify the program's control dependence. Their input syntax tree represents the hierarchical structure of the input but does not allow to generalize to other inputs, similar to a message field tree with no field attributes. They also propose a technique for inputs with bottom-up grammars, commonly used in programming languages.

In Dispatcher, we propose message format extraction techniques for sent messages and field semantics inference techniques for both received and sent messages. Compared to the above approaches, Dispatcher is able to extract the message format for received and sent messages from the same binary. This is important in scenarios where only the program that implements one side of the dialog is available such as when analyzing the C&C protocols used by botnets and instant messaging protocols. In addition, Dispatcher extracts fine-grained field semantics, which are important to understand what a message does, as well as for identifying fields with the same type across multiple messages.

State-machine. In addition to extracting the protocol grammar, protocol reverse-engineering also includes inferring the protocol's state-machine. ScriptGen infers the protocol state-machine from network data [117]. Due to the lack of protocol information in network data it is difficult for ScriptGen to determine whether two network messages are two instances of the same message type. This is needed when converting messages into alphabet symbols. ScriptGen outputs a state-machine that captures only previously-seen sessions, without generalization. Prospex uses execution trace similarity metrics to cluster messages of the same type so they can be assigned the same symbol from the alphabet [44]. Then, it extracts a tree that captures previously-seen sessions, labels the tree nodes using heuristics, and applies an algorithm to infer the minimal consistent DFA. Techniques to extract the message format like the ones presented in this chapter are a prerequisite for techniques that extract the protocol state-machine.

Protocol specification languages. Previous work has proposed languages to describe protocol specifications [17, 50, 173]. Such languages are useful to store the results from protocol reverse-engineering and enable the construction of generic protocol parsers. In this thesis, we use the BinPac language to represent our MegaD C&C protocol specification and the generic BinPac parser to analyze MegaD messages given that specification [173].

Other related work. Previous work has targeted protocol reverse-engineering for specific applications like protocol replay or inferring connections that belong to the same application session. RolePlayer [53] and ScriptGen [117, 116] address the problem of replaying previously captured network sessions. Such systems perform limited protocol reverse-engineering from network traffic only to the extent necessary for replay. Their focus is to identify the dynamic fields, i.e., fields that change value between sessions, such as cookies, length fields or IP addresses. Our field semantics inference techniques leverage the richer semantics available in protocol implementations compared to network traffic, accurately extracting a wide range of field semantics for dynamic fields. Replayer uses dynamic binary analysis to replay complete program executions that correspond to network dialogs [158]. Previous work also addresses the related problem of identifying multiple connections that belong to the same application session from network traffic [102].

3.9 Conclusion

In this chapter, we have proposed a new approach for automatic protocol reverse-engineering that uses dynamic program binary analysis. Our approach takes as input execution traces obtained by running a program that implements the protocol, while it processes a received message and builds the corresponding response. Compared to previous approaches that take as input network traces, our approach infers more complete protocol information and can analyze encrypted protocols.

We have develop techniques to extract the message format and the field semantics of messages on both directions of the communication, even when only one endpoint's implementation of the protocol is available. Our message format extraction techniques identify the field structure of a message as well as hard-to-find protocol elements in the message such as length fields, delimiters, variable-length fields, and multiple consecutive fixed-length fields. Our field semantics inference techniques identify a wealth of field semantics including filenames, IP addresses, timestamps, ports, and error codes. In addition, we have shown how to apply our techniques to encrypted protocols by identifying the buffers that hold the unencrypted received message after decryption and the unencrypted message to be sent before encryption.

We have implemented our techniques in a tool called Dispatcher and have used it to extract the grammar of the previously undocumented, encrypted, C&C protocol of MegaD, a prevalent spam botnet. We have shown how the protocol grammar enables active botnet infiltration by rewriting a message that the bot sends to the C&C server. Furthermore, we have evaluated our techniques on a variety of open protocols and compared Dispatcher's output with the output of Wireshark, a state-of-the-art protocol analyzer. Dispatcher achieves similar accuracy as Wireshark, without requiring access to the protocol grammar.

Part III

Binary Code Reuse

Chapter 4

Binary Code Reuse

4.1 Introduction

Often a security analyst wishes to reuse a code fragment that is available in a program's binary, what we call *binary code reuse*. For example, a piece of malware may use proprietary compression and encryption algorithms to encode the data that it sends over the network and a security analyst may be interested in reusing those functions to decode the network messages. Further, the analyst may be interested in building a network proxy that can monitor and modify the malware's compressed and encrypted protocol on the network. Also, for dialog replay if some field of a network protocol is changed, other dependent fields such as length or checksum fields may need to be updated [53]. If those fields use proprietary or complex encodings, the encoding functions can be extracted and deployed in the network proxy so that the rewritten message is correctly formatted. Another application is the creation of static unpacking tools for a class of malware samples [212]. Currently, creating a static unpacker is a slow, manual process. Frameworks have emerged to speed the process [209], but a faster approach would be to extract the unpacking function from the malware sample and reuse it as a static unpacker.

At the core of these and other security applications is binary code reuse, an important problem for which current solutions are highly manual [60, 122, 194]. In this thesis we conduct the first systematic study of *automatic binary code reuse*, which can be defined as the process of automatically identifying the interface and extracting the instructions and data dependencies of a code fragment from an executable program, so that it is self-contained and can be reused by external code. Reusing binary code is useful because for many programs, such as COTS applications and malware, source code is not available. It is also challenging because binary code is not designed to be reusable even if the source code it has been generated from is.

Binary code reuse encompasses two main challenges: *interface identification*, i.e., inferring the prototype of the code fragment so that other source code can interface with it, and *code extraction*, i.e., extracting the instructions and data dependencies of the code fragment so that it is self-contained and independent of the rest of the program's functionality. Interface identification is challenging because the code fragment may not have a prototype available, e.g., it was intended only for internal use. A prototype for the binary code fragment enables reuse of the code by generating and passing appropriate inputs. Interface identification is challenging because the *parameters* that comprise the prototype are not explicitly defined in the binary code and also because they need to be expressed using variables and types, which do not exist in the binary code. Our approach uses dynamic analysis to extract a parameter abstraction at the binary level (an *assembly parameter*) and then translate the assembly parameters into the formal parameters in the prototype. Interface identification takes as input a set of execution traces. It identifies the inputs and outputs for each function run present in the execution traces, splits them into assembly parameters, identifies important attributes such as the parameter type (input, output, input-output) and the parameter location (register, stack, table), and finally combines this information across the multiple function runs.

In addition to the prototype, we want to extract the code fragment itself, i.e., its instructions and data dependencies, so that it is self-contained and can be reused by other code, independently of the rest of the functionality in the program. The self-contained code fragment can easily be shared with other users and can be statically instrumented or rewritten, e.g., for profiling or to enforce a safety policy on its memory accesses if it is untrusted. Our code extraction technique uses the observation that for reusing a binary code fragment a user often has no need to understand its inner workings. For example, a security analyst may want to reuse the proprietary cipher used by some malware, together with the session keys, to decrypt some data, without worrying about how the proprietary cipher works. For these applications, complex reverse-engineering or decompilation methods are not necessary to recover the source code. We can leverage the support of current C compilers for inline assembly [75, 142] and generate a function with a C prototype but an inline assembly body. Thus, the output of our approach is a source code function even if the binary code fragment to reuse does not correspond to an assembly function. Code extraction uses a combination of static and dynamic analysis that includes hybrid disassembly [156], symbolic execution [106], and jump table identification [39] techniques.

Not all binary code can be reused. To reuse a binary code fragment, the fragment should have a clean interface and be designed to perform a specific well-contained task, mostly independent of the remaining code in the program. In this paper we mostly focus on reusing binary code fragments that correspond to functions at the source code level, what we call *assembly functions*, because in structured programming a function is the base unit of source code reuse. Functions are usually designed to perform an independent, well-contained task and have a well-defined interface, the *function prototype*. In addition, we show that a code fragment that does not correspond to a complete assembly function, but has a clean interface and performs a well-contained task, can also be reused. Reusing an arbitrary assembly function can be extremely challenging because the function interface can be convoluted and the function can have complex side effects. Our approach handles common side effects such as an assembly function modifying one of its parameters or accessing a global variable, and also handles calls to internal and standard library functions. But we exclude functions with a variable-length argument list or functions that are passed recursive structures such as trees. We refer the reader to Section 4.2.3 for a more detailed description of the problem's scope.

In this chapter, we design and implement BCR, a tool that extracts code fragments from program binaries and wraps them in a C prototype, so they can be reused by other C code. We use BCR to extract the encryption and decryption routines used by two spam botnets: MegaD and Kraken. We show that these routines, together with appropriate session keys, can be reused by a network proxy to decrypt encrypted traffic on the network. Further, we show that the network proxy can also rewrite the malware's encrypted traffic by combining the extracted encryption and decryption functions with the session keys and the protocol grammar. To show that we can reuse code fragments that do not correspond to complete assembly functions we also extract the unpacking functions from two samples of Zbot, a trojan, and use an unpacking fragment from one sample as part of the routine to unpack the other sample.

Other applications. In addition to the applications that we examine in this thesis, binary code reuse is useful for many other applications. For example, it can be used to automatically describe the interface of undocumented functions. It often happens that malware uses undocumented functions from the Windows API, which are not described in the public documentation [138]. Projects to manually document such functions [163] could benefit from our approach to automatically identify the interface of a binary code fragment. Extracted functions could also be useful in the development of programs that interoperate with other proprietary interfaces or file formats, by allowing the mixture of code extracted from previous implementations with re-implemented replacements and new functionality. Another application is to determine whether two pieces of binary code are functionally equivalent, i.e., given the same inputs they produce the same outputs even if the instructions in both pieces of binary code may differ. Recent work has addressed this issue at the source code level by fuzzing both pieces of source code and comparing the input-output pairs [97], but how to interface with a binary code fragment to perform such fuzzing is an open problem. Finally, a security analyst may want to fuzz a well-contained, security-sensitive function independently of the program state in which it is used.

4.2 Overview & Problem Definition

In this section we give an overview of binary code reuse, formally define it, outline the scope of our solution, and present an overview of our approach.

4.2.1 Overview

Binary code reuse comprises two tasks: 1) *interface identification*, i.e., identifying the interface of the code fragment and formatting it as a prototype that can be invoked from other source code; and 2) *code extraction*, i.e., extracting the instructions and data dependencies of the code fragment so that it is self-contained and can be reused independently of the rest of the program's functionality.

The interface of the code fragment specifies its inputs and outputs. Interface identification is challenging because binary code has memory and registers rather than named parameters, and has limited type and semantic information, which must be converted into a source level prototype. It is also challenging because the code fragment might have been created by any compiler or written by hand, thus few assumptions can be made about its calling convention. In addition, the extracted code fragment needs to be self-contained, so we need a recursive process that extracts any function called from inside the code fragment that we want to extract (and from inside those callees) and we need to account for the possible side effects from the code fragment and its callees. For example, we need to identify and extract the data dependencies such as global variables and tables that the code fragment uses.

Previous work on binary code reuse is highly manual [60, 122, 194]. As far as we know we are the first ones to systematically study automatic binary code reuse. Our goal is to automate the whole process, with a focus on automatically identifying the code fragment's interface. There are two different representations for the extracted binary code: decompiled source code [60, 122] and assembly instructions [60, 194]. In this work we use inline assembly with a C prototype because inline assembly is the most accurate representation of the code (it represents what gets executed) and because decompilation is not needed for binary code reuse. The use of inline assembly limits portability to the x86 architecture, and requires compiler support, but the x86 architecture is still by far the most important architecture in security applications, and commonly used compilers include rich support for inline assembly [75, 142].

To reuse a binary code fragment, the code fragment should have a clean interface and be designed to perform a well-contained task, relatively independent of the remaining code in the program. Otherwise, if the fragment's interface is not clean or the code performs several intertwined tasks and the user is only interested in one of them, it becomes difficult to separate the relevant code and interface with it. In structured programming, the above characteristics are usually associated with functions, which are the basic unit of (source) code reuse in a program and reduce the development and maintenance costs of a program by making the code modular. Thus, it makes sense to focus on reusing binary code that corresponds to a function at the source level since that code was designed to be modular in the first place. However, the source-level concept of a function may not be directly reflected at the binary level, since functions at the source level can be inlined, split into non-contiguous binary code fragments, or can exit using jumps instead of return instructions (e.g., due to tail-call optimizations). Despite this blurring, it is possible to define an *assembly function* abstraction at the binary level for which an inferred prototype gives a clean interface when the underlying functionality is well modularized. We describe our function abstraction next.

4.2.2 Problem Definition

To reuse functions from program binaries, we first need a function abstraction that captures our definition of what a function is in binary code.

Function abstraction. A *basic block* is a sequence of instructions that has one entry point, one exit point, and contains no instructions that perform control-flow transfer[1]. Basic blocks are disjoint and partition the code in an executable. We define an *assembly function* to be a collection of basic blocks with a single function entry point, which is the target of the instruction that transfers control from the external code into the assembly function code, and one or more function exit points, which are instructions that transfer control to external code not belonging to the function. All code reachable from the function entry point before reaching a function exit point constitutes the body of the assembly function, except code reachable only through call instructions before corresponding return instructions, which is instead part of the called function. In other words, the body of a function is assumed to continue with the next instruction after a call instruction. A function exit point can be a return or interrupt instruction. Our definition does not include assembly functions with multiple entry points, which we treat as multiple (partially overlapping) assembly functions, each including all code reachable from one function entry point to any function exit point.

If one assembly function jumps to another, i.e., without using a call function as mentioned in Section 2.4.1, this definition considers the blocks following the jump target to be part of the assembly function to extract. We can further extend our definition of a function exit point to include jumps to the entry point of any other assembly function in the program's binary or in an external dynamic linked library (DLL). For this we need a list of entry points for other assembly functions such as the one provided by the exported function log produced by the execution monitor.

[1]Call instructions are a special case that is often not considered to end a basic block because the callee often returns to the instruction following the call instruction.

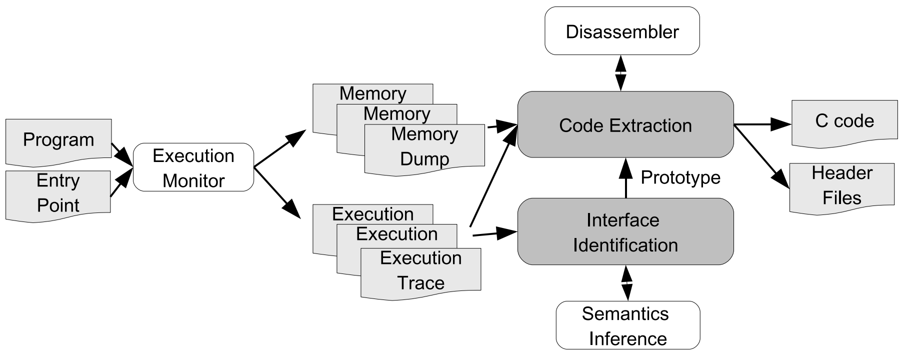


Figure 4.1: BCR architecture. The core of BCR are the interface identification and code extraction modules in gray. The execution module and the disassembler are previously-available building blocks described in Chapter 2. The semantics inference module was detailed in Chapter 3.

Problem definition. The problem of binary code reuse is defined as: given the binary of a program and the entry point of an assembly function in the binary, identify the interface and extract the instructions and data dependencies that belong to the assembly function so that it is self-contained and can be reused by external C code. The extracted function consists of both an inline assembly function with a C prototype and a header file containing the function's data dependencies. The problem definition when the code fragment does not correspond to a complete assembly function, i.e., for an arbitrary code fragment, is similar except that it requires the exit points to be given, because a return instruction can no longer be considered an exit point.

4.2.3 Scope

To reuse a binary code fragment, the fragment should have a clean interface and be designed to perform a specific well-contained task, mostly independent of the remaining code in the program. Our binary code reuse approach focuses on reusing assembly functions because they correspond to functions at the source code level, which are the base unit of source code reuse in structured programming. However, we also show that a code fragment that does not correspond to a complete assembly function, but has a clean interface and performs a well-contained task, can also be reused.

Reusing an arbitrary assembly function can be extremely challenging because the function interface can be convoluted and the function can have complex side effects. Our approach handles common side effects such as an assembly function modifying one of its parameters or accessing a global variable, and also handles calls to internal and standard library functions, but it excludes other complex cases such as functions with a variable-length argument list or functions that are passed recursive structures such as trees. An important class of functions that we extract in this thesis are *encoding functions*, which we introduced in Section 3.6 and include encryption and decryption, compression and decompression, code packing and unpacking, checksums, and generally any function that encodes data. Encoding functions are usually well-contained, have clean interfaces, limited side effects, and are interesting for many security applications. Next, we detail the assumptions we make to limit the scope of the problem:

- The function entry point is known. For encoding functions, we can identify the entry point using the techniques in Section 3.6 that flag functions with a high ratio of arithmetic and bitwise operations, as well as the techniques in Section 8.3 that flag functions that highly mix their inputs.
- Since our approach uses dynamic analysis, we assume that we can execute the function at least once. If some specific input is needed to reach the function, we assume we are provided with such input.
- The function has a fixed parameter list. Thus, we exclude functions with variable-length list of arguments such as `printf`.
- The function is not passed complex recursive structures such as lists or trees (pointers to single-level structures are supported).
- The function does not call system calls directly (e.g., through interrupt or `sysenter` instructions) but instead uses system calls only through well-known functions that are available in the target system where the function is reused (e.g., the standard C library, or the Windows API if the target system is Windows-based).
- The function contains no code that explicitly uses its own location. For example, the code should not check if it is loaded at a specific address or offset. This restriction excludes most self-modifying code. However, the function may still reference global addresses through standard position-independent-code and dynamic linking: relocatable and non-relocatable code are both supported.

4.2.4 Approach

Figure 4.1 illustrates our binary code reuse approach implemented in our BCR tool. First, the program is run inside the *execution monitor*, which was described in Chapter 2. The execution monitor outputs an execution trace of the run, as well as a process state snapshot when the exit point of the binary code fragment is reached. To produce the process state snapshot, the execution monitor tracks when execution reaches the given entry point of the code fragment and when it leaves the code fragment via an exit point (e.g., a return instruction). When the execution reaches an exit point, the execution monitor produces a process state snapshot.

In this chapter we refer to the process state snapshot as a memory dump since we do not use the register and taint information that the snapshot may also contain. It is important to take the memory dump at the end of the function's execution to maximize the code pages present in the dump, as pages may not be loaded into memory till they are used. The program may be run multiple times to produce a collection of execution traces and memory dumps.

The interface identification module takes as input the execution traces generated during program execution and outputs the prototype, which captures how external source code can interface with the code fragment. Interface identification comprises three steps. First, it identifies the inputs and outputs for each function run present in the execution traces. Then, it splits the inputs and outputs for each function run into assembly parameters and identifies important attributes such as the parameter type (input, output, input-output) and the parameter location (register, stack, table). It also identifies the parameter semantics using the semantics inference techniques presented in Chapter 3. Finally, it combines the assembly parameters found in the multiple function runs into the formal parameters for the function prototype.

The code extraction module extracts the instructions and data dependencies of the code fragment. It takes as input the execution traces, the memory dumps, and the prototype output by the interface identification module. It produces as output C code corresponding to the body of the code fragment, and header files with its data dependencies (e.g., with the tables and constants used by the code fragment). Code extraction comprises three steps. First, it recovers the instructions that form the body of the code fragment using hybrid disassembly [156]. Hybrid disassembly combines the best of static and dynamic disassembly. It addresses the problem of resolving indirection in static disassembly using information from the execution traces, and it can disassemble instructions that may

```
char enc_tbl[256] = { 0x53, … ,0x9c };
int encode(char*src, char*dst, int len) {
  int i;
  if (!src || !dst) return -1;
  memset(dst, 0, len);
  for (i=0; i<len; ++i)
    dst[i] = enc_tbl[src[i]];
  return len;
}
static int func_00401000(
    void* buf0,  /* IN; STACK(0); FixLen(4); PTR */
    void* buf1,  /* IN-OUT; STACK(1); FixLen(4); PTR */
    data32_t buf0_len,  /*IN;STACK(2); FixLen(4); LEN */
) {
data32_t       retval_EAX;
  __asm__ __volatile__ (
"push    %[val0]\n\t"
"push    %[buf1]\n\t"
"push    %[buf0]\n\t"
"call     lbl00401000\n\t"
"jmp      lbl_func_00401000_ret\n\t"
"lbl00401000:\n\t"
"push   %%ebp\n\t"
"mov    %%esp,%%ebp\n\t"
"push   %%ecx\n\t"
"cmpl   $0x0,0x8(%%ebp)\n\t"
"je   lbl00401010\n\t"
 ...
"lbl00401015:\n\t"
"mov    0x10(%%ebp),%%eax\n\t"
"push   %%eax\n\t"
"push   $0x0\n\t"
"mov    0xc(%%ebp),%%ecx\n\t"
"push   %%ecx\n\t"
"call   memset\n\t"
"add    $0xc,%%esp\n\t"
"movl   $0x0,-0x4(%%ebp)\n\t"
"jmp    lbl00401039\n\t"
 ...
"lbl00401041:\n\t"
"mov    0x8(%%ebp),%%ecx\n\t"
"add    -0x4(%%ebp),%%ecx\n\t"
"movsbl (%%ecx),%%edx\n\t"
"mov    0xc(%%ebp),%%eax\n\t"
"add    -0x4(%%ebp),%%eax\n\t"
"mov    0x3018+tbl_00400000(%%edx),%%cl\n\t"
"mov    %%cl,(%%eax)\n\t"
"jmp    lbl00401030\n\t"
"lbl0040105a:\n\t"
"mov    0x10(%%ebp),%%eax\n\t"
"mov    %%ebp,%%esp\n\t"
"pop    %%ebp\n\t"
"ret    \n\t"
"lbl_func_00401000_ret:\n\t"
  : /* outputs */ "=a" (retval_EAX)
  : /* inputs */ [buf0] "mem" (buf0), [buf1] "mem" (buf1),
    [val0] "mem" (val0), [buf2] "c" (buf2)
  : /* clobber list */ "memory"
  );
  return  retval_EAX;
}
```

Figure 4.2: Running example. At the top, the source for the `encode` function. Below, the extracted assembly function. The boxes indicate changes to the assembly code.

not have been executed during the program runs. Hybrid disassembly outputs the disassembled instructions belonging to the code fragment. Then, the code extraction module makes the assembly code relocatable by arranging the disassembled instructions into basic blocks and rewriting table accesses, as well as the target addresses of call and jump instructions, to use labels. Finally, it encodes the body as an inline-assembly block inside a function definition that uses the C prototype output by the interface identification module. In addition, it creates a C header file containing the memory dump as an array.

Isolation. Because the extracted code runs in the same address space as the program that uses it, the same security concerns apply to it as to an untrusted third-party library: a malicious extracted function might attempt to call other functions in memory or overwrite the application's data. An isolation mechanism is needed to limit what the extracted code can do. In this work we process the extracted code with a software-based fault isolation (SFI) tool to insert runtime checks that prevent the extracted code fragment from writing or jumping outside designated memory regions (separate from the rest of the program). We use PittSFIeld [134], an SFI implementation for x86 assembly code that enforces jump alignment to avoid overlapping instructions and includes a separate safety verifier, which can be used by a third party to verify that the code has been correctly rewritten.

Running example. Figure 4.2 shows our running example. At the top is the source code for the `encode` function, which reads `len` characters from buffer `src`, transforms them using the static table `enc_tbl`, and writes them to the `dst` buffer. Below it is the assembly function corresponding to the `encode` function, extracted by BCR from the program binary. The large boxes in the figure show the C prototype produced by the interface identification module, and the prologue and epilogue introduced by the code generation module. The smaller boxes show the additional elements in the body of the function that have been rewritten or modified to make the function stand-alone. The rest are the unmodified assembly instructions extracted by the body extraction module. Also produced, but omitted from the figure, is a header file that contains a memory dump of the original module. This header file contains the module's data that the extracted code may access, e.g., the contents of the table `tbl_004000000` needed by the boxed instruction that accesses the table.

Next, we detail the interface identification module in Section 4.3 and the code extraction module in Section 4.4. For simplicity, we focus the discussion on the case where the binary code fragment to reuse corresponds to an assembly function. Later in Section 4.5 we show an example of applying BCR to a code fragment that does not correspond to an assembly function.

4.3 Interface Identification

The goal of the interface identification module is to build a C function prototype for the assembly function so that it can be reused from other C code. The C prototype comprises the function's name and a list of its *formal parameters*. However, formal parameters do not directly appear at the binary code level, so BCR works with a binary-level abstraction, which we term an *assembly parameter* and describe next. At the same time, we collect some additional information, such as the parameter length or its semantics. This information does not directly appear in the prototype, but it is needed for interfacing with the extracted code.

The interface identification module identifies the assembly parameters using a dynamic analysis that takes as input the execution traces produced by the execution monitor. Thus, it can only extract parameters that have been used by the function in the executions captured in the given execution traces. To increase the coverage inside the function we can use the white-box exploration techniques we describe in Chapters 6–8. In our experiments, we achieve no false negatives in interface identification with less than 8 runs for any function and have not needed such exploration.

In the remainder of this section we describe how to identify the prototype of an assembly function. The process for identifying the prototype of an arbitrary binary code fragment is analogous.

Parameter abstraction. An assembly parameter plays a role for an assembly function analogous to a formal parameter for a C function, specifying a location representing an input or output value. But, instead of being referred to by a human-written name, assembly parameters are identified with a location in the machine state. To be specific, we define assembly parameters with five attributes:

1. The *parameter type* captures whether it is only an input to the function (`IN`), only an output from the function (`OUT`) or both (`IN-OUT`). An example of an `IN-OUT` parameter is a character buffer that the assembly function converts in-place to uppercase.
2. The *parameter location* describes how the code finds the parameter in the program's state. A parameter can be found on the stack, in a register, or at another location in memory. For stack parameters, the location records the fixed offset from the value of the stack pointer at the entry point; for a register, it specifies which register. Memory locations can be accessed using a fixed address or pointed by another pointer parameter, perhaps with an additional offset. BCR also specially classifies globals that are accessed as tables via indexing from a fixed starting address, recording the starting address and the offset.
3. The *parameter length* can be either fixed or variable. A variable length could be determined by the value of another length parameter, or the presence of a known delimiter (like a null character for a C-style string).

4. The *parameter semantics* indicates how its value is used. Parameters have pointer or length semantics if they are used to identify the location and size of other parameters, as previously described. Our parameter abstraction supports a number of semantic types related to system operations, such as IP addresses, timestamps, and filenames. An “unknown” type represents a parameter whose semantics have not been determined.
5. The *parameter value list* gives the values BCR has observed the parameter to take over all assembly function executions. This is especially useful if the parameter’s semantics are otherwise unknown: a user can just supply a value that has been used frequently in the past.

Overview. The interface identification module performs three steps. For each assembly function execution, it identifies a list of assembly parameters used by the assembly function in that run (Section 4.3.1). Next, it combines the assembly parameters from multiple runs to identify missing parameters and generalizes the parameter attributes (Section 4.3.2). Finally, it identifies additional semantics by running the assembly function again in the execution monitor using the parameter information and a taint tracking analysis (Section 4.3.3). Later, in Section 4.4.2, we will explain how the code generation module translates the assembly parameters produced by the interface identification module into the formal parameters and outputs the C function prototype.

4.3.1 Identifying the Assembly Parameters from a Function Run

For each function run in the execution traces the interface identification module identifies the run’s assembly parameters. Because there are no variables at the binary level (only registers and memory), this module introduces abstract variables (sometimes called A-locs [8]) as an abstraction over the machine-level view to represent concepts such as buffers and stack parameters. These variables must be sufficiently general to allow for rewriting: for instance, the addresses of global variables must be identified if the variable is to be relocated. A final challenge is that because the code being extracted might have been created by any compiler or written by hand, BCR must make as few assumptions as possible about its calling conventions.

In outline, our approach is that the interface identification module first identifies all the bytes in the program’s state (in registers or memory) that are either an input or an output of the assembly function, which we call *input locations* and *output locations*, respectively. It then generalizes over those locations to recognize abstract locations and assembly parameters. To get the best combination of precision and efficiency, we use a combination of local detection of instruction idioms, and whole-program dataflow analysis using tainting and symbolic execution. In the remainder of this section we refer to an assembly parameter simply as a parameter for brevity, and use the term formal parameter to refer to the parameters in the C function prototype. Next, we define what input and output locations are. Note that, as introduced in Section 3.4.2, a program location is a one-byte-long storage unit in the program's state (memory, register, immediate or constant).

Input locations. We define an input location to be a register or memory location that is read by the function in the given run before it is written. Identifying the input locations from an execution trace is a dynamic dataflow-based counterpart to static live-variables dataflow analysis [151], where input locations correspond to variables live at function entry. Like the static analysis, the dynamic analysis conceptually proceeds backwards, marking locations as inputs if they are read, but marking the previous value of a location as dead if it is overwritten. (Since we are interested only in liveness at function entrance, we can use a forward implementation.) The dynamic analysis is also simpler because only one execution path is considered, and the addresses in the trace can be used directly instead of conservative alias analysis. This basic determination of input locations is independent of the semantics of the location, but as we will explain later not all input locations will be treated as parameters (for instance, a function's return address will be excluded).

Output locations. We define an output location to be a register, memory, or constant location that is written by the extracted function and read by the code that executes after the function returns. Extending the analogy with compiler-style static analysis, this corresponds to the intersection of the reaching definitions of the function's code with the locations that are live in the subsequent code. Like static reaching definitions [151], it is computed in a single forward pass through the trace.

Our choice of requiring that values be read later is motivated by minimizing false positives (a false positive output location translates into an extra parameter in the C function prototype). This requirement can produce false negatives on a single run, if an output value is not used under some circumstances. However, our experience is that such false negatives can be well addressed by combining multiple function runs, so using a strict definition in this phase gives the best overall precision.

Approach. The input and output locations contain all locations belonging to the assembly parameters and globals used by the assembly function, without regard to calling conventions. In addition to identifying them, the interface identification module needs to classify the input and output locations into higher level abstractions representing parameters. Also, it needs to identify whether a parameter corresponds to a stack location, to a global, or is accessed using a pointer. The overall parameter identification process from one function run is summarized in Table 4.1 and described next.

For efficiency, the basic identification of parameters is a single forward pass that performs only local analysis of instructions in the trace. It starts at the entry point of one execution of a func-

Step	Description
1	Identify stack and table accesses
2	Identify input and output locations
3	Remove unnecessary locations (e.g., saved registers, ESP, return address)
4	Identify input and input-output pointers by value
5	Split input locations into parameter instances using pointer, stack and table access information
6	Identify input parameter pointers by dataflow
7	Split output locations into parameter instances using pointer information
8	Identify output parameter pointers by dataflow

Table 4.1: Summary of parameter identification process for a function run.

tion, and uses one mode to analyze both the function and the functions it calls, without discerning between them (for instance, a location is counted as an input even if it is only read in a called function), and another mode to analyze the remainder of the trace after the function finishes. For each instruction, it identifies the locations the instruction reads and writes. For each location, it identifies the first and last times the location is read and written within the function, as well as the first time it is read or written after the function. Based on this information, a location is classified as an input location if it is read inside the function before being written inside the function, and as an output location if it is written in the function and then read outside the function before being written outside the function; observe that a location can be both an input and an output.

At the same time, the analysis identifies stack and table accesses by a local matching of machine code idioms. The ESP register is always considered to point to the stack. The EBP register is only considered to point to the stack if the difference between its value and that of ESP at function entrance is a small constant, to support both code that uses it as a frame pointer and code that uses it as a general-purpose register. Then, a memory access is a stack access if it uses a stack register as a starting address and has a constant offset. On the other hand, a memory access is classified as a table access if its starting address is a constant and the offset is a non-stack register. The starting address and offset values in stack and table accesses are recorded for future use.

Excluding unnecessary input locations. The input locations given by the simple liveness-style definition above include several kinds of locations with bookkeeping roles in function calls which should not be considered parameters, so we next discuss how to exclude them. To exclude the return address, the interface identification module ignores any memory locations written by a call instruction or read by a return instruction during the function execution. To exclude the stack pointer, it ignores any access to ESP. When code calls functions in a dynamically linked library, it fetches the real entry point of the function from an export table, but we exclude such loads.

Most complex is the treatment of saved registers. For instance, we define a stack location to be used for saving the register EBX if the contents of EBX are first saved in that location with a push instruction, and later restored to EBX with a pop instruction. But the location is not a saved register location if the value is popped to a different register than it was pushed from, if the stack value is accessed before the pop, or if after the pop, the stack value is read before being overwritten. Conventionally, the stack is used to save certain registers designated by the calling convention if a called function modifies them, but our analysis is independent of the calling convention's designation: it simply excludes any location used only for saving a register.

Identifying pointers. A final building block in identifying parameters is to identify locations that hold pointers. The interface identification module uses a combination of two approaches for this task: an inexpensive value-based method that can be applied on all locations, and a more expensive dataflow-based method that works by creating a symbolic formula and is applied selectively. To detect a pointer by value, BCR simply checks each sequence of four consecutive input locations (pointers are four bytes on our 32-bit architecture) to see if their value forms an address of another input or output location. However, this simple approach can fail to detect some pointers (for instance, the address of a buffer that was only accessed with non-zero indexes), so we also implement a more sophisticated approach.

To identify more pointers, the interface identification module uses a symbolic execution approach using our Vine system [19] to analyze an indirect memory access. The input locations to the function are marked as symbolic variables, and the module computes a formula for the value of the effective address of the access in terms of them, using dynamic slicing and simplification as explained in Section 2.4.2. It then checks whether the resulting formula has the form of a 32-bit symbolic input plus a constant. If so, the input locations are considered a pointer, and the constant an offset within the region the pointer points to. (The reverse situation of a constant starting address and a variable offset does not occur, because it would already have been classified as a global table.) Though precise, this symbolic execution is relatively expensive, so the interface identification module uses it only when needed, as we will describe next.

Identifying assembly parameters from input and output locations. Once the input and output locations have been identified and unnecessary locations removed, the interface identification module identifies input and input-output pointers by value as explained above. Then it uses the pointers, stack, and table accesses to classify the input and output locations into assembly parameters. Each parameter is a contiguous region in memory (or a register), but two distinct parameters may be adjacent in memory, so the key task is separating a contiguous region into parameters. The module considers a location to be the start of a new parameter if it is the start of a pointer, the address after the end of a pointer, or the location of a pointer, stack, or table access. With the information found so far, the interface identification module determines the parameter type, location, and value, and if the parameter has pointer semantics. The parameter length is provisionally set to the length seen on this run.

Then, the interface identification module attempts to further classify any parameters that are in memory but are not on the stack and are not known globals by applying the dataflow-based pointer identification analysis. Specifically, it checks whether the access to the starting location of the parameter was a pointer access; if so, it updates the type of the pointed-to parameter and the semantics of the pointer parameter accordingly. After classifying the input locations and pointers in this way, the module classifies the output locations similarly, and then identifies and classifies other pointers that point to them.

4.3.2 Combining Assembly Parameters from Multiple Function Runs

The set of assembly parameters identified from a single run may be incomplete, for instance if an input pointer is first checked not to be null and if null, the function returns without further processing. This is the case with the `src` and `dst` parameters of the `encode` function in Figure 4.2. Therefore the interface identification module further improves its results by combining the information about parameters identified on multiple runs.

The final set of parameters identified is the union of the parameters identified over all runs, where parameters are considered the same if they have the same parameter location. When parameters with the same location differ in other attributes between runs, those attributes are merged as follows:

- The parameter type generalizes to input-output if it was input in some runs and output in others.
- The parameter length generalizes to variable-length if it was fixed-length in some runs and variable-length in others, or if it had differing lengths across runs.
- The parameter semantics generalizes to any non-unknown value if it was a known value in some runs and unknown in others (e.g., a parameter is considered a pointer if it was identified to be a pointer at least once, even if it was considered unknown on runs when it was NULL). On the other hand, the semantics are replaced with unknown if they had conflicting non-unknown values on different runs.
- The parameter value list is the union of all the observed values.

4.3.3 Identifying Parameter Semantics

In addition to the declared type of a parameter included in the C prototype, it is also common (e.g., in MSDN documentation [138]) to provide additional information in text or a comment that explains how the parameter is used; what we refer to as its *semantics*. For instance, one integer parameter

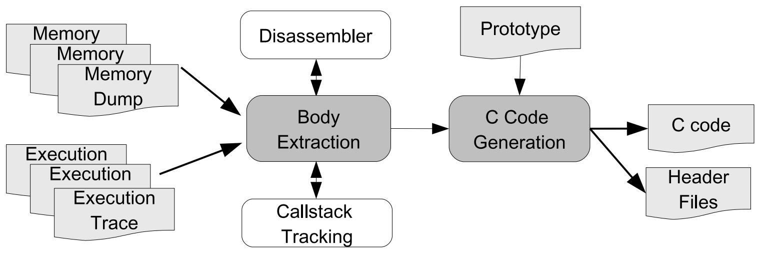


Figure 4.3: Architecture of the code extraction module.

might hold the length of a buffer, while another is an IP address. We next describe the techniques that the interface identification module uses to identify parameter semantics.

Two kinds of semantics that occur frequently in encoding functions as part of specifying other input and output parameters are pointers and lengths. As described above, the parameter identification process finds pointer parameters at the same time it identifies the parameters they point to. To identify length parameters, their targets, as well as variable-length parameters that use a delimiter to mark their end (e.g., null-terminated strings), we leverage the message format extraction techniques introduced in Chapter 3.

The interface identification module uses the semantics inference techniques presented in Chapter 3 to detect semantics related to system operations such as IP addresses, timestamps, ports, and filenames. In a nutshell, certain well-known functions take inputs or produce outputs of a particular type, so BCR uses taint tracking to propagate these types to the target function (the one being extracted) if an output of a well-known function is used as an input to the target function, or an output of the target function is an input to a well-known function. For instance, the argument to the `inet_ntoa` function is an IP address, so an output parameter that is used to derive that argument must itself be an IP address. Conversely, if an input parameter is based on the return value of `RtlGetLastWin32Error`, it must be an error code. Currently, BCR supports the semantics defined in Table 3.3 plus "pointer" and "unknown". A similar approach can be used at the instruction level to select a more specific C type (e.g., `float` rather than `int`) [82].

Semantics inference leverages the execution monitor's support for function hooks, which we introduced in Section 2.3.2. Hooks added after the execution of well-known functions and the target function taint their outputs, and hooks before their execution check if their inputs are tainted. Because a function hook can only be added to the target function after its parameters have been identified, semantics inference requires an extra run of the function in the execution monitor.

4.4 Code Extraction

In this section we describe the code extraction module. The main challenges for code extraction are identifying all instructions from the assembly function to reuse and producing a properly formatted C function that can be reused by external code. Figure 4.3 shows the architecture of the code extraction module, which comprises two sub-modules: *body extraction* and *C code generation*. First, the body extraction module uses hybrid disassembly to disassemble all instructions that comprise the body of the assembly function. In addition, it makes the assembly code relocatable by rewriting the assembly instructions to use labels for the targets of call and jump instructions, as well as for absolute memory addresses. Then, the C code generation module formats the assembly function as an inline-assembly block, creates a C function prototype from the prototype output by the interface identification module, and adjusts the function's prologue and epilogue so that it conforms to the calling convention expected by the C code. For brevity, in this section we use “C function” to refer to a function with a C prototype and an inline-assembly body.

4.4.1 Body Extraction

Extracting the body of an assembly function to reuse is a recursive process that starts by extracting the body of the given assembly function and then recursively extracts the body of each of the assembly functions that are descendants of this function in the function call graph. The body extraction module classifies descendant functions into two categories: *well-known functions* that may be available in the system where the C function is going to be recompiled, e.g., functions in the standard C library or in the Windows Native API, and the rest, which we term *internal functions*. The body extraction module extracts the body of the given function and all internal descendant functions. As an optimization, it avoids extracting well-known functions. This increases portability: for example if a function from a Windows executable uses `strcpy` from the standard C library, it can be recompiled in a Linux system making a call to the local `strcpy` function. In other cases, portability is not possible because the function may not have a direct replacement in the target OS (e.g., there is no direct replacement in Linux for `NtReadFile`), so this optimization is not performed and we use instead a compatibility layer [226]. For instance, in our running example, shown in Figure 4.2, the `encode` function calls `memset`; since it is part of the C library and thus likely available in the target system, `memset` is not extracted.

Hybrid disassembly. The body extraction module uses *hybrid disassembly*, a technique that combines static and dynamic disassembly [156]. The body extraction module supports three disassembly modes of operation: purely static, purely dynamic, and hybrid disassembly. In purely static disassembly, the body extraction module statically disassembles the code starting at the given entry point, using the IDA Pro commercial disassembler [90]. If the program binary is not packed, then disassembly is performed directly on the executable. For packed binaries disassembly is performed on the memory dump taken by the execution monitor at the code's exit point. Purely static disassembly provides good code coverage but may not be able to disassemble code reachable through indirect jumps or calls, as well as code intertwined with data.

In purely dynamic disassembly, the body extraction module disassembles only instructions belonging to the function and its descendants that appear in the given execution traces. For this, it uses the call stack tracking module presented in Chapter 2 to identify which instructions belong to each function. Purely dynamic disassembly has low code coverage but has no trouble dealing with packed executables, or indirect jumps or calls.

In hybrid disassembly, the body extraction module combines both static disassembly with dynamic information from the execution traces to obtain the best of both modes. We have found that hybrid disassembly works best and have set it to be the default mode of operation. For hybrid disassembly, the body extraction module first uses static disassembly starting at the given function entry point. In the presence of indirection, the static disassembler may miss instructions because it can not resolve the instructions' targets. Thus, the body extraction module collects the targets of all indirect jumps and calls seen in the execution traces and directs the static disassembler to continue disassembling at those addresses. For example, in Figure 4.2, the call to the `memset` function was originally a direct call to a stub that used an indirect jump into `memset`'s entry point in a dynamic library. The body extraction module resolves the target of the jump and uses the information in the exported functions log, output by the execution monitor, to determine that the function is the standard `memset`. In addition, the body extraction module uses a dataflow-based approach to statically identify the targets of jump tables, another class of indirect jumps often used to implement switch statements [39]. This technique can find additional targets of jump tables, not seen during the execution. In the presence of code intertwined with data, the static disassembler may also miss instructions. Hybrid disassembly collects the addresses of all executed instructions and directs the static disassembler to continue disassembling at those addresses. Using this approach, hybrid disassembly is able to identify more instructions that belong to the function body but that do not appear in the execution traces.

There exist some situations where static disassembly may not be possible even from a memory dump, for instance if a program re-packs or deletes instructions right after executing them: the code may be re-packed by the time the memory dump is taken. In such a situation hybrid disassembly smoothly falls back to be equivalent to purely dynamic disassembly. To summarize, hybrid disassembly uses static disassembly when possible and incorporates additional dynamic information to further extend disassembly. For each function, hybrid disassembly stores the disassembled basic blocks, and recovers the control flow graph.

Rewriting call/jumps to use labels. Once the C function is recompiled it will almost certainly be placed at a different address, so the body extraction module needs to make the code relocatable. To enable this, it inserts a label at the beginning of each basic block. Then, it rewrites the targets of jump and call instructions to use these labels. If the target of a jump instruction has not been recovered by the hybrid disassembly, it is rewritten to use a unique missing block label that exits the function with a special error condition. Figure 4.2 uses small boxes to highlight the inserted block labels and the rewritten call/jump instructions. Rewriting the call/jump instructions to use labels also enables a user or a subsequent tool (like the SFI tool discussed in Section 4.5.5) to instrument the function or alter its behavior by inserting new instructions in the body.

Rewriting global and table accesses. The extracted C function is composed of a C file with the assembly function and a header file. The header file contains a memory dump of the module containing the function to extract, taken at the function's exit point on a given run. The body extraction module rewrites instructions that access global variables or tables to point to the corresponding offsets in the memory dump array. This way the extracted function can access table offsets that have not been seen in the execution traces. In our running example, the header file is not shown for brevity, but the array with the contents from the memory dump is called `tbl_004000000` and the instruction that accesses `enc_tbl` has been rewritten to use the label `0x3018+tbl_00400000` which is the first byte of `enc_tbl` in the memory dump. The memory dump is taken at the function's exit point, but if the interface identification module discovers any input parameters that are accessed using a fixed address and modified inside the function, e.g., a global table that is updated by the function, it ensures that the parameter values on function entry are copied into the dump, so that they are correct when the function is invoked again.

An alternative approach would be to create separate C arrays and variables for each global parameter, which would reduce the space requirements for the extracted function. Though this would work well for scalar global variables, it would be difficult to infer the correct size for tables, since the binary does not contain bounds for individual variables, and code compiled from C often does not even have bounds checks. (An intermediate approach would be to estimate the size of a table by multiplying the largest observed offset by a safety factor; this would be appropriate if it could be assumed that testing covered at least a uniform fraction of the entries in each table.)

4.4.2 C Code Generation

The code generation module writes the output C files using the information provided by the interface identification module and the body extraction module. To encode the function body, the code generation module uses GCC's inline assembly feature [75]. It wraps the function body in an assembly block and then puts the assembly block inside a function definition with a C function prototype, as shown in Figure 4.2. In addition it creates a C header file containing the memory dump as an array. Though our current implementation is just for GCC, the inline assembly features of Visual C/C++ [142] would also be sufficient for our purposes. In fact, some of the Visual C/C++ features, such as "naked" inline assembly functions, for which the compiler does not generate a prologue or epilogue, could simplify our processing.

The assembly block contains the assembly instructions and the list of inputs, outputs, and clobbered registers. These are filled using the parameter information provided by the interface identification module. When GCC compiles the function, it will add prologue and epilogue code that affects the stack layout, so even if the extracted function originally used a standard calling convention, it would not find the stack parameters where it expects. To overcome this problem, the code generation module inserts wrapper code at the beginning of the function that reads the parameters from the C prototype (as inputs to the assembly block), puts them in the stack or register locations expected by the extracted function, and calls the extracted entry point. After the call instruction it inserts a jump to the end of the function so that the epilogue inserted by GCC is executed. The second box in Figure 4.2 shows this wrapper.

The C prototype comprises the function name and the formal parameters of the function. The function name is based on its entry point (`func_00401000` in the running example), and each parameter's C type is based on its size and whether it is a pointer. Input and input-output parameters located in the stack or registers appear first, with stack parameters appearing in order of increasing offset (this means that if the extracted function used the most common C calling convention, their order will match the original source). For each output parameter returned using a register, the code generation module adds an additional pointer formal parameter at the end of the C prototype and uses the outputs list in the assembly block to let GCC know that the register needs to be copied to the pointed-to location. Additionally, for output global or table parameters the code generation module adds a C variable corresponding to the start address of the global or table in the memory dump. This makes the function's side effects available to other C code.

Each formal parameter is also annotated with a comment that gives information about the attribute values for the corresponding assembly parameter such as the parameter type and its semantics. These are useful for a user who wants to reuse the function. In addition, it prints the most common value seen for each parameter during the multiple executions along with the percentage of executions where the parameter showed that value. This allows the user to select a value for the parameter when the parameter semantics are unknown. The function prototype is shown in the first box in Figure 4.2.

4.5 Evaluation

This section describes the experiments we have performed to demonstrate that our binary code reuse approach and implementation is effective for security applications such as rewriting encrypted malware network traffic and static unpacking, that non-function fragments can be extracted to give useful functions, and that extracted functions can be used safely even though they come from an untrusted source.

4.5.1 Rewriting MegaD's C&C Protocol

In Chapter 3 we introduced MegaD, a prevalent spamming botnet first observed in 2007 and credited at its peak with responsibility for sending a third of the world's spam [129]. In that chapter we reverse-engineered MegaD's proprietary, encrypted, C&C protocol and demonstrated how to rewrite C&C messages on the host by modifying a buffer before encryption. In this section we show that our binary code reuse approach enables the same C&C rewriting on a network proxy, by extracting the bot's key generation and encryption functions.

Function extraction. MegaD's C&C protocol is protected using a proprietary encryption algorithm, and the bot contains functions for block encryption, block decryption, and a common key generator. We identify the entry points of the three functions using the techniques presented in Section 3.6 that flag functions with a high ratio of arithmetic and bitwise operations.

First, we use BCR to automatically extract the key generation function. The identified prototype shows that the function has two parameters and uses two global tables. The first parameter points to an output buffer where the function writes the generated key. The second parameter is a pointer to an 8 byte buffer containing the seed from which the key is generated. Thus, the function generates the encryption key from the given seed and the two tables in the binary. Other attributes show that all calls to the key generation function use the same "abcdefgh" seed, and that the two tables are not modified by the function.

Although the entry points for the block encryption and decryption functions are different, the first instruction in the block decryption function jumps to the entry point of the block encryption function, so here we describe just the encryption function. The prototype extracted by BCR has 3 parameters and uses 6 global tables. The first parameter points to an input buffer containing a key (as produced by the key generation function). The other two parameters are pointers to the same 8 byte input-output buffer that on entry contains the unencrypted data and on exit contains the encrypted data.

The technique to automatically detect encoding functions identifies the functions with highest ratio of arithmetic and bitwise operations, which for block ciphers is usually the functions that process a single block. To encrypt or decrypt an arbitrary message, we would like a function that encrypts or decrypts arbitrary length data. Thus, when using this technique, after BCR extracts the detected encoding functions, we instruct it to extract their parent functions as well. Then, we compare the prototype of each detected function with the one of the parent. If the parent's prototype is similar but accepts variable-length data, e.g., it has a length parameter, then we keep the parent function, otherwise we manually write a wrapper for the block function. For MegaD, the parent of the block encryption function has additional parameters, because it performs other tasks such as setting up the network and parsing the message. It contains no single loop that performs decryption of a variable-length buffer; instead, decryption is interleaved with parsing. Since we are not interested in the parent function's other functionality, we write our own wrapper for the block encryption function.

Note that in the process of extracting the encoding functions we also identify the keys that each function invocation uses. We do this by looking for input parameters to the encoding functions that are fixed-length, are not derived from the input, and have no other semantics (i.e., are not a pointer or a length). For MegaD all invocations of the encryption/decryption function use the same key, which corresponds to the output of the key generation function. As mentioned earlier, the key generation function takes as input a hard-coded seed value "abcdefgh", thus generating the same key in each invocation.

To verify that the extracted encryption/decryption function works correctly, we augment the grammar for the unencrypted MegaD C&C protocol, presented in Appendix A, to use the extracted decryption function. This augmented grammar serves as input to the BinPac parser shipped with the Bro intrusion detection system [173]. Using the augmented grammar, Bro successfully parses all the encrypted MegaD C&C messages found in our network traces.

Network-based C&C rewriting. To perform network rewriting we must deploy the encryption/decryption function, as well as the session keys, in a network proxy. Such a proxy will only be effective if the functions and keys match those in the bots, so to estimate the rate at which they change we repeat our analysis with other MegaD samples. In total we have analyzed four MegaD samples, who were first seen in the wild between February 2008 and February 2010. Although there are differences between the samples, such as some samples using TCP port 80 instead of 443 for its C&C, the parser, using the decryption function and keys extracted from the December 2008 sample, is able to successfully parse the C&C messages from all other samples. In addition, we extract the key generation and encryption functions from the oldest sample (February 2008) and compare them with the ones from the December 2008 sample. Although there are syntactic differences, i.e., short sequences of instructions have been replaced with equivalent ones possibly by recompiling with different options or by obfuscating the code, the versions are functionally equivalent, producing the same outputs on more than a billion randomly generated inputs. Thus we conclude that the relevant algorithms and keys, including the session key, have been unchanged during the two year time span of our samples.

To show how our binary code reuse approach enables live rewriting on the network, we build a network proxy that is able to decrypt, parse, modify and re-encrypt MegaD C&C messages that it sees on the network. To test the proxy we reproduce the experiment in Section 3.7.1, but perform rewriting on the network rather than on the host. The experiment proceeds as follows.

We run a live MegaD bot inside a network that filters all outgoing SMTP connections for containment purposes, but allows the C&C protocol through. To start, suppose that no proxy is in use. The bot probes the C&C server for a command and the C&C server sends in response a message that orders the bot to test its ability to send spam by connecting to a test mail server. Because the firewall at the border of the network blocks SMTP, the connection to the test mail server fails and the bot sends a reply to the C&C server indicating that it cannot send spam, and afterward no more spam-related messages are received.

Next, we repeat the experiment adding a network proxy that acts as a man-in-the-middle on traffic between the C&C server and the bot. For each message sent by the bot, the proxy decrypts it and checks if it is a message that it needs to rewrite. When the bot sends the message indicating that it has no SMTP capability, the proxy, instead of relaying it to the C&C server, creates a different message indicating that the SMTP test was successful, encrypts it, and sends it to the C&C server instead. Note that the fail and success messages were identified as part of our reverse engineering of MegaD's C&C protocol grammar, presented in Chapter 3, by observing the differences in the output message that is sent right after the SMTP connection succeeded or failed. Also, it would not suffice for the proxy to replay a previously captured success message, because the message also includes a nonce value selected by the C&C server at the beginning of each dialog. With the proxy in place, the bot keeps receiving spam-related C&C messages, even if it is unable to actually send spam. The spam-related C&C messages include a command to download a spam template, which contains all the information about the spam operations such as the format of the messages, the spam URLs, and the list of addresses to spam.

4.5.2 Rewriting Kraken's C&C Protocol

Kraken is a spam botnet that was discovered on April 2008 and has been thoroughly analyzed [178, 122, 194]. Previous analysis uncovered that Kraken (versions 315 and 316) uses a proprietary cipher to encrypt its C&C protocol and that the encryption keys are randomly generated by each bot and prepended to the encrypted message sent over the network [178, 122]. Researchers have manually reverse-engineered the decryption function used by Kraken and have provided code to replicate it [122]. In this thesis, we extract Kraken's decryption function using our automatic approach and verify that our extracted function is functionally equivalent to the one manually extracted in previous work. Specifically, when testing the manually and automatically extracted function on millions of random inputs, we find their outputs are always the same. In addition, we extract the corresponding encryption function and a checksum function, used by the bot to verify the integrity of the network messages.

Similarly to the MegaD experiment described in Section 4.5.1, we build a network proxy that uses the extracted encryption, decryption, and checksum functions, as well as the protocol grammar, and use it to rewrite a C&C message to falsify the result of an SMTP capability check. Unfortunately (for the purpose of this experiment), none of our Kraken samples connects to a live C&C server on the Internet. Thus, to verify that the message rewriting works we use a previously published Kraken parser [122]. The rewritten message parses correctly and has the STMP flag correctly modified.

4.5.3 Reusing Binary Code that is not an Assembly Function

Next, we show that our approach enables reusing a binary code fragment that does not correspond to a complete assembly function, but has a clean interface and performs an independent task. We extract unpacking code from two versions of a trojan horse program *Zbot* used primarily to steal banking and financial information [202]. Zbot uses two nested layers of packing. The samples, provided to us by an external researcher, represent a typical task in the course of malware analysis: they have already had one layer of packing removed, and we have been provided the entry points for a second, more complex, unpacking routine.

The function prototype extracted by BCR is identical for both functions. It contains two pointer parameters: the ESI register points to an input-output buffer containing packed data as input and a count of the number of bytes unpacked as output, while the EDI register points to an output buffer for unpacked data. Since ESI and EDI are not used for parameter passing in any of the standard x86 calling conventions, this suggests these functions were originally written in assembly code.

Although the prototypes are the same, the unpacking functions are not functionally equivalent; they both consist of two distinct loops, and we find that extracting these loops separately captures more natural functional units. Examining the extracted function bodies, we find that both consist of

	General		Code Extraction			Parameter Identification		
Function	**Runs**	**Runtime (sec)**	**# Insn.**	**# Missed blocks**	**# Indirect call/jump**	**# Param.**	**FP**	**FN**
MegaD keygen	4	3	320	0	0	3	0	0
MegaD encrypt	6	257	732	0	0	4	0	0
Kraken encrypt	2	16	66	0	0	7	1	0
Kraken decrypt	1	2	66	0	0	6	0	0
Kraken checksum	1	179	39	0	0	4	1	0
Zbot v1151	2	15	98	0	0	2	0	0
Zbot v1652	2	17	93	0	0	2	0	0
MD5_Init	6	2	10	0	0	1	0	0
MD5_Update	6	38	110	0	1	3	0	0
MD5_Final	7	31	67	0	3	2	0	0
SHA1_Init	1	8	11	0	0	1	0	0
SHA1_Update	1	36	110	0	1	3	0	0
SHA1_Final	2	36	76	0	3	2	0	0

Table 4.2: Evaluation results. At the top are the functions extracted during the end-to-end applications and at the bottom some additional functions extracted from the OpenSSL library.

two loops that are separated by `pusha` and `popa` instructions that save and restore processor state. Each loop makes its own pass over the packed data, with the first pass applying a simpler deciphering by subtracting a hard-coded key, and the second pass performing a more complex instruction-by-instruction unpacking. After extracting the two loops into separate functions, we verify that the differences between the versions are only in the first loop: the extracted version of the second loop can be reused across the sample versions. This highlights the fact that as long as a binary code fragment has a clean interface and performs a well-separated task, it can be reused even if it does not correspond to a complete function in the original machine code.

4.5.4 Quantitative Summary of Function Extraction

Table 4.2 summarizes the extraction results for all functions in Section 4.5.1 through Section 4.5.3 and the MD5 and SHA1 hash functions that we extract from the OpenSSL library for evaluation purposes. Note that, in OpenSSL obtaining the hash of a value requires calling three separate functions: Init, Update, and Final. The *General* section of the table shows the number of function runs in the execution traces used as input to the function extraction step, and the total time needed to extract the function. The *Code Extraction* section has the number of instructions in each extracted function, the number of missed blocks and the number of indirect call and jump instructions. We approximate the number of missed blocks by counting the number of conditional jumps for which we have not seen the code that follows each of its branches. Note that when the function has indirect calls or jumps this method is not enough. However, in this case only the OpenSSL functions, for which we have the source code, present indirect calls or jumps. The *Parameter Identification* section shows the number of parameters in the C function prototype and the number of false positives (e.g., unnecessary parameters in the prototype) and false negatives (e.g., missing parameters in the prototype). For the OpenSSL functions, the false positives and negatives are measured by comparison with the original C source code. For the malware samples, no source is available, so we compare with our manual analysis and (for Kraken) with other reported results.

The results show that a small number of executions is enough to extract the complete function without missing blocks or parameters. For samples without indirect jumps or calls, static disassembly recovers all basic blocks. For the samples with indirection, the dynamic information resolves the indirection and enables the static disassembler to find all the instructions in the function body. The Kraken checksum and MegaD encrypt samples are significantly slower to extract than the other samples. This is because they have larger number of invocations of the dataflow-based pointer analysis technique, which dominates the running time. The parameter identification results show that no parameters are missed. Some runs do not identify all parameters, e.g., when the function first checks if a pointer has NULL value and if NULL it exist without further processing of the other parameters. However, combining multiple executions (Section 4.3.2) gives complete results. For the functions from OpenSSL, the parameters include fields in a context structure that is passed to the functions via a pointer. There are two false positives in the Kraken functions (i.e., extra parameters are identified), both of which are output parameters reported as returned in the ECX register. These are caused by a compiler optimization (performed by the Microsoft compiler, for instance) that replaces the instruction `sub $4,%esp` to reserve a location on the stack with the more compact instruction `push %ecx`, which has the same effect on the stack pointer and also copies a value from ECX that will later be overwritten. When this idiom occurs in the code following an extracted function that uses ECX internally, the interface identification module incorrectly identifies ECX as a function output. Such common idioms could be specially handled during interface identification or identified through bottom-to-top live analysis since the ECX value is dead. More generally, false positive parameters are not a serious problem for usability: extra outputs can simply be ignored, and extra inputs do not change the extracted function's execution.

4.5.5 Software-based Fault Isolation

If the extracted functions are to be used in a security-sensitive application, there is a danger that a malicious extracted function could try to hijack or interfere with the operation of the application that calls it. To prevent this, we use software-based fault isolation (SFI) [217] as a lightweight mechanism to prevent the extracted code from writing to or calling locations in the rest of the application. SFI creates separate "sandbox" data and code regions for the extracted function, so that it can only write to its data region and it can only jump within its code region. SFI works by adding checks just before each store or jump instruction, but the extracted code still runs in the same address space, so calls from the application are still simple and efficient.

Specifically, we post-process our extracted malware functions using PittSFIeld, an implementation of SFI for x86 assembly code [134]. PittSFIeld adds new instructions for checks, and to enforce additional alignment constraints to avoid overlapping instructions. Thus, BCR's translation of jumps to use labels is necessary for it to work. PittSFIeld was previously implemented for use with the assembly code generated by GCC, so in order to work with assembly code that could be generated by other compilers or hand-written, we generalize it to save and restore the temporary register used in sandboxed operations, and to not assume that EBP is always a pointer to the stack. We also make corresponding changes to PittSFIeld's separate verification tool, so a user can check the safety of an extracted function without trusting the person who extracted it.

4.6 Related Work

This section compares our approach with the manual process it aims to replace, other automatic approaches, techniques for related problems in other domains, and some other tasks that require similar algorithms.

Manual code extraction. Code extraction is a common manual activity in malware analysis [60, 122, 194]. While this process can give the analyst a deep understanding of the malicious functionality, it is also very time-consuming. Simple tool support can make some of the repetitive tasks more convenient, but existing approaches still require specialized skills. Our approach allows this task to be automated, when all that is needed is to be able to execute the functionality in another context.

Other binary code reuse approaches Kolbitsch et al. have since proposed a different approach for binary code reuse and implemented it in a tool called Inspector Gadget [109]. Their approach is based in identifying a sink in an execution where interesting behavior happens (e.g., a call to a function that writes to disk) and then apply dynamic program slicing to identify the instructions and data dependencies related to that behavior in the execution. The slice is the basis of a gadget: the binary code to be reused. There are significant differences between BCR and Inspector Gadget.

First, BCR requires as input the entry point of the code to reuse while Inspector Gadget requires as input a point in the execution where the desired behavior manifests, e.g., a call to a function like `gethostbyname` or `WriteFile`. Second, Inspector Gadget has been designed to extract behaviors that may include multiple steps (e.g., download a file, decrypt it, execute it), while BCR has been designed to extract individual functionalities (e.g., the decryption function). While Inspector

Gadget can extract full behaviors at once, full behaviors limit the possibilities in which an analyst can reuse the code: an analyst can only interface with the gadget by changing the values returned by external functions that the gadget invokes. In contrast, BCR performs interface identification and is able to generate a valid C function that comprises only some specific functionality (e.g., an algorithm) that any other C program can invoke. For example, it is not clear how to use Inspector Gadget to enable a NIDS to decrypt C&C messages that are seen on the network, as presented in Section 4.5, or how to build a signature that identifies botnet traffic by first decrypting the traffic and then checking for some pattern in the decrypted code. Those are not behaviors that the bot implements but an analyst may still want to reuse the decryption function and implement its own functionality on top. While the decryption function may be present in a gadget extracted from the bot, it may not be easy to interface with it.

Third, gadgets only contain instructions that appear during the single execution trace used as input to Inspector Gadget. This is an important limitation because branches that were not taken during execution may need to be executed during code reuse, i.e., when the input changes the program may take a different path. In such a situation, a gadget would crash. To address this issue Inspector Gadget modifies the slice to include additional instructions that force the branch to take the path that was observed during the execution. This solution is problematic since the gadget no longer behaves as the original program and the results may not be meaningful. In contrast, BCR addresses this issue using hybrid disassembly, which combines static and dynamic disassembly to increase the number of relevant instructions extracted. Hybrid disassembly includes all instructions from the code fragment that appear in the (possibly multiple) input execution traces, and in addition, it is often able to disassemble many more instructions that were not executed (i.e., do not appear in the execution traces).

Finally, it is difficult to obtain a slice that matches exactly the desired functionality. Dynamic slicing tends to be conservative and that results in slices that are larger than needed. One example is when slicing a variable (e.g., the content of the EAX register), that was popped from the stack. The stack pointer (i.e., the ESP register) is a dependency of the pop instruction. If the stack pointer is included in the slice, then all previous instructions that use the stack, regardless if related to the behavior under analysis, will also need to be included in the slice. This is the conservative approach and the resulting slice includes much unrelated functionality. On the other hand, if the stack pointer is not included in the slice, then the slice is not complete and rerunning the slice would produce a crash if the stack layout is not correct. These slicing details are not mentioned in [109]. However, in previous work [108] the authors decide not to include the stack pointer in the slices and propose heuristics to *fix the stack*, which are not guaranteed to work and require knowledge about compiler-specific mechanisms for handling procedures and stack frames.

Input-output relationship extraction. A variant on the extraction problem is extracting the relationship between some given inputs and outputs of a computation. To extract such relationships, previous work has used symbolic execution [28, 108] or dynamic binary slicing [108, 114]. When the functionality to be extracted is sufficiently simple, it can be represented by a single input-output symbolic formula. For instance, such input-output formulas can be used for protocol dialog replay [28], or as a malware signature [108]. However, a single formula is not a practical representation for more complex functionality that includes loops or other variant control-flow paths, or uses complex data structures.

Another alternative representation is a dynamic binary slice that captures the instructions needed to produce the output from the inputs in a given execution. Dynamic binary slices are usually generated by applying modified versions of dynamic program slicing techniques [2] on execution traces. For instance, Lanzi et al. [114] produce dynamic binary slices using a combination of backwards and forward slicing, and use them to analyze kernel malware. When it cannot extract an exact input-output symbolic formula, the malware modeling tool of Kolbitsch et al. [108] combines dynamic binary slicing with tainted scopes to capture control dependencies. There are two main differences between extracting input-output symbolic formulas or dynamic binary slices and binary code reuse. First, our problem is more difficult because the inputs and outputs must be inferred. Second, by using a combination of dynamic and static analysis to extract the body of the code fragment we achieve better coverage than purely dynamic techniques.

Other applications of interface extraction. Jiang and Su [97] investigate the problem of automatic interface extraction in C source code, to allow automated random testing for fragments with equivalent behavior. Their task of determining which variables constitute inputs and outputs of a fragment is related to the one we tackle in Section 4.3, but made easier by the availability of type information. Extracting the code itself is also easier because in their scenario code fragments are restricted to contiguous statements.

Lin et al. [126] extract an interface to functionality in a benign program in order to add malicious functionality: for instance, to turn an email client into a spam-sending trojan horse. Because the functionality runs in its original context, their interface need not cover all inputs and outputs of the code, only those relevant to a particular use. Using techniques similar to our output inference, they perform a side-effect analysis to determine whether a function's memory effects can be reverted.

Liveness analysis. The analysis that our tool performs to identify input and output variables are the dynamic analogues of static data-flow analysis performed by compilers, such as live variable and reaching definitions analysis [151]. Some of the same challenges we face have also been addressed in purely static tools that, like our tool, must operate on binary code. For instance, link-time optimizers [81, 189] must also exclude saves of callee-saved registers from the results of naive liveness analysis.

Binary rewriting. Many of the techniques required for binary code reuse are used in binary rewriting and instrumentation applications. For instance, purely static disassembly provides insufficient coverage for even benign applications on Windows/x86 platforms, so state-of-the art rewriting tools require a hybrid of static and dynamic disassembly [156] much as we do. Cifuentes and Van Emmerik [39] introduced the technique we adopt for locating the jump table statements used to implement switch statements.

4.7 Conclusion

This chapter performs the first systematic study of automatic binary code reuse, which we define as the process of automatically identifying the interface and extracting the instructions and data dependencies of a code fragment from an executable program, so that it is self-contained and can be reused by external code.

We have proposed a novel interface identification technique to extract the prototype of an undocumented code fragment directly from the program's binary, without access to its source code. We have designed a code extraction approach to automatically extract a code fragment from a program binary so that it is self-contained. The extracted code fragment can be run independently of the rest of the program's functionality in an external C program, and can be easily tested, instrumented, or shared with other users.

We have implemented BCR, a tool that uses our approach to automatically extract an assembly function from a program binary. We have used BCR to reuse the cryptographic routines used by two spam botnets in a network proxy that can rewrite the malware's C&C encrypted traffic. In addition, we have extracted an unpacking function from a trojan horse program, and have shown that a code fragment belonging to that function can be reused by the unpacking function for a different sample from the same family. Finally, we have applied software-based fault isolation techniques [134] to the extracted functions to ensure they can be used safely even though they come from an untrusted source.

Part IV

Model Extraction

Chapter 5

Deviation Detection

5.1 Introduction

Many different implementations usually exist for the same specification. Due to the abundance of coding errors and specification ambiguities, these implementations usually contain *deviations*, i.e., differences in how they check and process some of their inputs. As a result, the same inputs can cause different implementations to behave differently. For example, an implementation may not perform sufficient input checking to verify if an input is well-formed as indicated in the specification. Thus, for some inputs, it might exhibit a deviation from another implementation, which follows the specification and performs the correct input checking.

Automatically finding deviations between implementations of the same specification, what we call *deviation detection*, is important for several applications. For example, deviation detection is important for testing implementations of network protocols. The Internet Standards process requires that two independent implementations of a protocol from different code bases have been developed and tested for interoperability before advancing a protocol to Draft Standard [18]. Deviation detection could be used to enhance interoperability testing since experience shows that even after current manual testing, differences still exist on how different protocol implementations handle some of the protocol inputs. In this thesis we show how to automatically find deviations between two implementations of the same protocol specification and how to apply the discovered deviations to two particular applications: *error detection* and *fingerprint generation*.

First, deviations are important for error detection because a deviation often indicates that at least one of the two implementations has an error or that the specification is underspecified. Finding such errors is important to guarantee that the specification is correctly implemented, to ensure proper interoperability with other implementations, to make certain that the specification is unambiguous, and to enhance system security since errors often represent vulnerabilities that can be exploited.

Enabling error detection by automatically finding deviations between two different implementations is particularly attractive because it does not require a manually written model of the specification. These models are usually complex, tedious, and error-prone to generate. Note that deviations do not necessarily flag an error in one of the two implementations or the specification. For example, it could happen that the specification allows for optional functionality that is provided by only one of the implementations. However, deviation detection is a good way to automatically find candidate implementation errors and to detect ambiguities in the specification.

Second, such deviations naturally give rise to *fingerprints*, which are inputs that, when given to two different implementations, will result in different output states. Fingerprints can be used to distinguish between the different implementations and we call the discovery of such inputs *fingerprint generation*. Fingerprinting has been in use for more than a decade [43] and is an important tool in network security for remotely identifying which implementation of an application or operating system a remote host is running. Fingerprinting tools [162, 181, 6] need fingerprints to operate and constantly require new fingerprints as new implementations, or new versions of existing implementations, become available. Thus, the process of automatically finding these fingerprints, i.e., the fingerprint generation, is crucial for these tools.

Deviation detection is a challenging task— deviations usually happen in corner cases, and discovering deviations is often like finding needles in a haystack. Previous work in related areas is largely insufficient. For example, the most commonly used technique is a variant of fuzz testing where random or semi-random inputs are generated and set to different implementations to observe if they trigger a difference in outputs [29, 63, 145]. The obvious drawback of this approach is that it may take many such random inputs before finding a deviation.

In this thesis we propose a novel approach to automatically discover deviations in input checking and processing between different implementations of the same protocol specification. We are given two programs in binary form P_1 and P_2, which implement the same protocol specification and wish to find inputs such that the same input, when sent to the two implementations, will cause each implementation to result in a different output state. At a high level, we build two models, M_1 and M_2, which capture how each implementation processes a single input. Then, we check whether the predicate $(M_1 \wedge \neg M_2) \vee (\neg M_1 \wedge M_2)$ is satisfiable, using a solver such as a decision procedure. If the predicate is satisfiable, it means that we can find an input, which will satisfy M_1 but not M_2 or vice versa, what we call a deviation because such input leads the two program executions to different output states. Note that such inputs are only deviations for certain if the models have perfect accuracy and coverage. Otherwise, the returned inputs are only good candidates to trigger a deviation. Since our models may not include all possible execution paths, we verify such candidate inputs by sending them to the two programs and monitoring their output states. If the two programs end up in two different output states, then we have successfully found a deviation between the two implementations, and the corresponding input that triggers the deviation.

We have designed and implemented model extraction techniques that produce models covering a single execution path in the program, and found that such models are surprisingly effective at finding deviations between different implementations of the same functionality. We have evaluated our approach using 3 HTTP server implementations and 2 NTP server implementations. Our approach successfully identifies deviations between the different server implementations for the same protocol and automatically generates inputs that trigger different server behaviors. These deviations include errors and differences in the interpretation of the protocol specification. For example it automatically finds an HTTP request that is accepted by the MiniWeb Web server with a "HTTP/1.1 200 OK" response, while it is rejected by the Apache Web server with a "HTTP/1.1 400 Bad Request" response. Such deviation is due to an error in the MiniWeb server that fails to verify the value of the first byte in the URL. The evaluation shows that our approach is accurate: in one case, the relevant part of the input that triggers the deviation is only three bits. Our approach is also efficient: we find deviations using a single request in about one minute.

The remainder of this chapter is organized as follows. Section 5.2 introduces the problem and presents an overview of our approach. Section 5.3 describes our model extraction technique. Section 5.4 presents how to generate candidate deviation inputs and how to validate that they truly trigger a deviation. Our evaluation results are presented in Section 5.5. We discuss enhancements to our approach in Section 5.6. Finally, we present the related work in Section 5.7 and conclude in Section 5.8.

5.2 Problem Definition & Approach Overview

In this section, we first introduce the intuition behind our model extraction techniques. Then, we formally define the deviation detection problem with respect to the extracted models. Finally, we provide an overview of our deviation detection approach.

5.2.1 Model Extraction

Our model extraction techniques use the intuition that a program can be seen as a mapping function $P : I \rightarrow S$ from the input space I to the output space S. A program accepts an input $x \in I$ and then processes the input resulting in a particular *output state* $s \in S$, i.e., $P(x) = s$. Note that we say an output state rather than an output because we are interested in programs that process highly structured inputs (e.g., files or network traffic) and many different inputs may produce the same output state. For example, consider a network protocol that includes a timestamp in every message such as the `Date` header in an HTTP response. Every HTTP request received by a Web server may produce a different HTTP response because the `Date` value in the response constantly changes, but many of those HTTP responses will produce an equivalent output state, i.e., they will return "HTTP 200 OK" and serve back a file hosted in the Web server.

Our model extraction techniques consider output states that are disjoint and extract a *model* for each different output state. Our models are boolean predicates that capture all inputs to the program $x \in I$ that cause the program to reach a particular output state s: $M_P^s(x) = true \iff P(x) = s$. Thus, the program can be seen as a conjunction of the models for each output state. Such models could be generated in different ways (e.g., statically or dynamically, from the program's source code or from its binary). In this thesis, we describe dynamic approaches for model extraction that take as input the program in binary form.

Two important properties of models are correctness and completeness. A model M_P^s is correct if it returns true only for inputs that reach the output state s. It is complete if returns true for all inputs that can possibly reach the output state s. Our models are correct by construction because they only contain paths that have been executed and have reached the output state. To be complete a model needs to cover all paths that can possibly reach the output state. Thus, a critical challenge for model extraction techniques is to build high-coverage models that capture many execution paths in the code. In this thesis we have evolved our model extraction techniques to progressively increase their coverage. In this chapter we show that even models that cover a single execution path are useful to find deviations. Then, in Chapter 6 we present techniques for extracting multi-path models that are the disjunction of multiple execution paths that reach an output state. Finally, in Chapter 7 we show how to refine those multi-path models by merging common parts of the execution, so that the resulting model has higher coverage and is smaller.

An output state is a pair that comprises a program point and a boolean predicate on the program state that needs to be satisfied at that program point. It can similarly comprise a set of such pairs. Thus, the model can be seen as a conjunction of a *reachability predicate*, which captures the inputs that make the program execution reach the program point in the output state definition, and the boolean predicate that needs to hold on the program state when that point is reached.

The semantics of an output state, i.e., what the program point and the predicate in the output state represent, are application-dependant. For example, when finding deviations between Web servers, an output state for a Web server could be that an input HTTP request causes a successful delivery of a webpage in an output HTTP response. Here, the output state could be defined with a pair where the program point is the call site that invokes the `send` function, and the boolean predicate states whether the message to be sent has a 200 status code, which indicates success. If there are multiple call sites for the `send` function, a set of pairs could be used instead.

5.2.2 Problem Definition

Deviation detection is the problem of automatically finding program inputs that cause two implementations of the same specification to reach different output states. We call those program inputs *deviations* and focus on finding deviations between two implementations of the same protocol specification.

We are given two implementations of the same protocol specification in binary form P_1 and P_2. As explained in Section 5.2.1 each implementation at a high level can be viewed as a mapping function $P_1, P_2 : I \rightarrow S$ from the protocol input space I to the protocol output space S. Our goal is to find inputs $x \in I$ such that $P_1(x) \neq P_2(x)$. Finding such inputs through random testing is usually hard. However, in general it is easy to find inputs $x \in I$ such that $P_1(x) = P_2(x) = s \in S$, i.e., most inputs will result in the same output state s for different implementations of the same specification.

For example, given two implementations of a Web server, e.g., Apache [4] and MiniWeb [147], implementing the same HTTP protocol specification [65], it is easy to find inputs (e.g., HTTP requests) for which both servers, if configured similarly, produce the same output state (e.g., an "HTTP 200 OK" response). However, it is not so easy to find deviations, inputs for which both servers produce different output states such as one server accepting the request with a "HTTP/1.1 200 OK" response, while the other one rejecting it with a "HTTP/1.1 400 Bad Request" response. Our approach automatically finds such deviations.

Output states. The output states for deviation detection need to be externally observable. We use two methods to observe such states: (a) monitoring the network traffic output by the program, and (b) supervising its environment, which allows us to detect unexpected states such as program halt, reboot, crash, or resource starvation. However, we cannot simply compare the complete output from both implementations, since the output may be different but equivalent. For example, many protocols contain sequence numbers, and we would expect the output from two different implementations to contain two different sequence numbers. However, the output messages may still be equivalent. Thus, we may use some domain knowledge about the specific protocol being analyzed to determine when two output states are different. For example, many protocols such as HTTP include a status code in the response to provide feedback about the status of the request. We use this information to determine if two output states are equivalent or not. In other cases, we observe the effect of a particular query in the program, such as program crash or reboot. Clearly these cases are different from a response being emitted by the program.

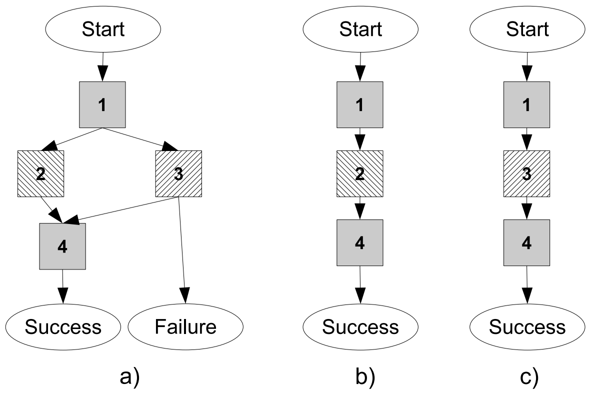


Figure 5.1: On the left, the control-flow graph of a program. On the right, two different execution paths in the program that end up in the same output state. Since two different paths can end up in the same output state, the validation phase checks whether the new execution path truly ends up in a different state.

5.2.3 Approach

The intuition behind our deviation detection approach is that if we have a model for the output state s for each of the two implementations of the same specification: $M^s_{P_1}$, $M^s_{P_2}$, then a deviation is just an input that satisfies the following predicate: $(M^s_{P_1} \wedge \neg M^s_{P_2}) \vee (\neg M^s_{P_1} \wedge M^s_{P_2})$. Such an input is a deviation because it produces an output state s for one of the implementations and another output state $t \neq s$ for the other implementation.

Given the above intuition, the main challenge is creating a model M^s_P that captures all the program inputs that reach the output state s. Furthermore, we observe that the above method can still be used even if our model does not consider the entire program and only considers a *single* execution path. In that case, the model M^s_P represents the subset of protocol inputs that follow the same execution path and still reach the output state s. Thus, $M^s_P(x) = true \Rightarrow P(x) = s$, since if an input satisfies M^s_P then by definition it will make program P go to state s, but the converse is not necessarily true—an input which makes P go to state s may not satisfy M^s_P. For example, Figure 5.1a shows the control flow graph of a program P with two paths that reach the `Success` state. If the post-condition for the `Success` state is always true and the model for the `Success` state contains only the path in 5.1b, then there exists an input, i.e., the one that produces the execution in 5.1c, which makes P go to the `Success` state but does not satisfy the model for 5.1b.

In our problem, this means that the difference between $M^s_{P_1}$ and $M^s_{P_2}$ may not necessarily result in a true deviation. Instead, the difference between $M^s_{P_1}$ and $M^s_{P_2}$ is a good candidate, which we can then test to validate whether it is a true deviation. We discuss models with multiple execution paths in Section 5.6.

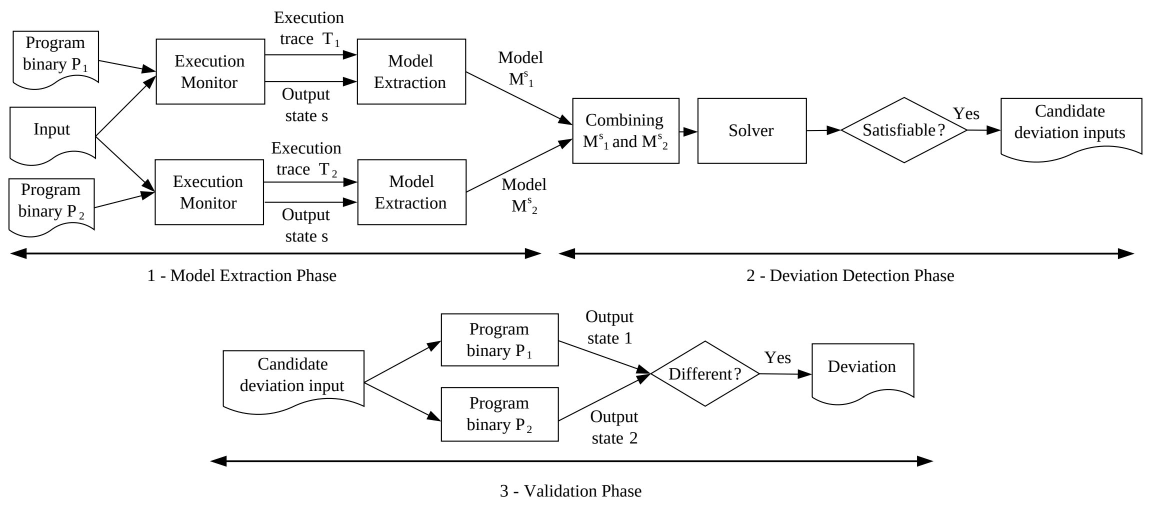


Figure 5.2: Overview of our deviation detection approach.

Our approach is an iterative process, and each iteration consists of three phases, as shown in Figure 5.2. First, in the *model extraction* phase, we are given two binaries P_1 and P_2 implementing the same protocol specification, such as HTTP, and an input x, such as an HTTP GET request. For each implementation, we log an execution trace of the binary as it processes the input, and record what output state s it reaches, such as halting or sending a reply. For this, we use the execution monitor introduced in Chapter 2. We start monitoring the execution before sending a message to the program and stop the trace when we observe a response from the program. We use a no-response timer to stop the trace if no answer is observed from the server after a configurable amount of time. We assume that the execution from both binaries reaches equivalent output states; otherwise we have already found a deviation! For each implementation P_1 and P_2, we then use this information to produce a model (a symbolic predicate) over the input, M_1^s and M_2^s respectively, each of which is satisfied for inputs that cause each binary to follow the same path than the original input did in each program and still reach the same output state s as the original input did.

Next, in the *deviation detection* phase, we use a solver (such as a decision procedure) to find differences between the two models M_1^s and M_2^s. In particular, we ask the solver if $(M_1^s \wedge \neg M_2^s) \vee (M_2^s \wedge \neg M_1^s)$ is satisfiable. When satisfiable the solver will return an example satisfying input. We call these inputs the *candidate deviation inputs*.

Finally, in the *validation* phase we evaluate the candidate deviation inputs obtained in the model extraction phase on both implementations and check whether the implementations do in fact reach different output states. This phase is necessary because the symbolic predicate might not include all possible execution paths, then an input that satisfies M_1^s is guaranteed to make P_1 reach the same equivalent output state as the original input x but an input that does not satisfy M_1^s may also make P_1 reach a equivalent output state. Hence, the generated candidate deviation inputs may actually still cause both implementations to reach equivalent output states.

If the implementations *do* reach different output states, then we have found a deviation triggered by that input. This deviation is useful for two things: (1) it may represent an implementation error in at least one of the implementations, which can then be checked against the protocol specification to verify whether it is truly an error; (2) it can be used as a *fingerprint* to distinguish between the two implementations.

Iteration. We can iterate this entire process to examine *other* input types. Continuing with the HTTP example, we can compare how the two implementations process other types of HTTP requests, such as HEAD and POST, by repeating the process on those types of requests. In Section 5.6 we discuss how to use white-box exploration techniques to automate this iteration, so that many different inputs types can be tested and rarely used paths can be covered.

5.3 Extracting Single-Path Models

In this section, we describe the model extraction phase. The goal of the model extraction phase is that given an input x such that $P_1(x) = P_2(x) = s$, where s is the output state when executing input x with the two given programs, we would like to compute two models, M_1^s and M_2^s, such that, $M_1^s = true \Rightarrow P_1(x) = s$ and $M_2^s = true \Rightarrow P_2(x) = s$. Each model is a boolean predicate that captures the set of inputs that would reach the output state s by following the same execution path that the program followed during the execution. Each model is the conjunction of the symbolic path predicate obtained by executing the program on symbols, rather than concrete inputs, and the boolean predicate Q_s that defines the output state.

5.3.1 Building the Symbolic Path Predicate

In our design, we build the symbolic predicate in two distinct steps. We first execute the program on the original input, while recording a trace of the (concrete) execution, using the execution monitor introduced in Chapter 2. We then use this execution trace as input to the dynamic symbolic execution process that builds the symbolic predicate.

In this section we explain how to generate the symbolic path predicate from an execution trace. In dynamic symbolic execution, the input to the program (e.g., the network message received by the program) is converted into a sequence of symbols (one symbol per input byte) and the program is run on a combination of symbols and concrete values. When the program reaches a branch predicate that uses some of the input symbols, a symbolic branch condition is created. The conjunction of all symbolic branch conditions forms the path predicate, a predicate on the symbolic program inputs that captures all the inputs that would follow the same execution path in the program than the original input. The symbolic path predicate can be generated using different methods like forward symbolic execution or weakest pre-condition [78, 33, 20, 21].

In this thesis we compute the symbolic predicate using the technique of *weakest pre-condition* [23, 61]. To compute the weakest pre-condition we use the Vine platform [19], which was introduced in Chapter 2. The weakest pre-condition, denoted $wp(P, Q)$, is a boolean predicate f over the input space I of program P such that if $f(x) =$ *true*, then $P(x)$ will terminate in a state satisfying Q. In our setting, the post-condition is the predicate Q_s that defines the output state s. Thus, the weakest pre-condition computed over the execution trace with the post-condition Q_s is exactly the model that captures the set of inputs that would reach the output state s by following the same execution path that the program followed during the execution. In an nutshell, to generate the symbolic predicate, we perform the following steps:

1. Record the execution trace of the program on the original input, which sets the program path.
2. Translate the execution trace into a program B, written in the intermediate representation (IR) offered by the Vine intermediate language. As an optimization the taint information from the execution trace can be used to include in the IR program only instructions that operate on data derived from the input.
3. Translate B into a single assignment (SSA) form.
4. Calculate the weakest pre-condition $wp(B, Q_s)$ using the algorithm proposed by Brumley et al. [21]. The weakest pre-condition on a single-path program starts with the post-condition and further constrains it to follow the same program path taken in the trace, by adding assertions for the symbolic branches encountered during the execution.

5.3.2 Memory Reads and Writes using Symbolic Addresses

If an instruction accesses memory using an address that is derived from the input, then in the path predicate the address will be symbolic. A symbolic address could access different memory locations and we must analyze the set of possible locations it may access. To create a sound path predicate, we add an assertion to the path predicate to only consider executions that would calculate an address within this set. The size of this set with respect to the set of all possible locations that may be accessed influences the generality of the path predicate. The larger the set, the more general the path predicate is at the cost of more analysis.

Memory reads. When reading from memory using a symbolic address, the path predicate must include initialization statements for the set of memory locations that could be read. In some cases, we achieve good results considering only the address that was actually used in the logged execution trace and adding the corresponding constraints to the path predicate to preserve soundness. To increase the generality of the path predicate we use two other techniques. We use range analysis to estimate the range of symbolic memory addresses that could be accessed [8]. Range analysis is conservative but costly. In addition, we add special handling for the common case of static tables, which are present in the data section of the program and are read, but not written, by the program. For this, we extract the constraints leading to the table access and query the solver for possible offset values in the table. If the solver returns an input, we add it (negated) to the path predicate, e.g., if it returns INPUT = 5, then we add to the path predicate the constraint INPUT != 5. This forces the solver to return a different answer the next time we query it. We repeat this process until the solver fails to return an input. This method identifies the different locations in the table that may be accessed. Then, our system extracts the contents of those locations from a memory dump and adds concrete initializers to the path predicate.

Memory writes. We need not transform writes to memory locations that use a symbolic address. Instead we record the set of possibly accessed addresses, and add the corresponding constraint to the path predicate to preserve soundness. These constraints force the solver to reason about any potential alias relationships. As part of the weakest pre-condition calculation, subsequent memory reads that could use one of the addresses being considered are transformed to a conditional statement handling these potential aliasing relationships. As with memory reads, we often achieve good results by only considering the address that was actually used in the logged execution trace. This reduces the coverage of the path predicate but maintains its accuracy. Again, we could generalize the predicate to consider more values, by selecting a larger set of addresses to consider.

5.4 Deviation Detection & Validation

In this section we present the deviation detection and validation phases. The deviation detection phase takes as input the models for each implementation produced by the model extraction phase, presented in Section 5.3, and outputs candidate inputs to trigger a deviation. Then, the validation phase verifies whether those candidate deviation inputs truly trigger a deviation.

5.4.1 Deviation Detection

The deviation detection phase uses a solver to find candidate inputs, which may cause deviations. This phase takes as input the models M_1^s and M_2^s generated for the programs P_1 and P_2 in the model extraction phase. We rewrite the variables in each predicate so that they refer to the same input, but each to their own internal states. We then query the solver whether the combined predicate $(M_1^s \wedge \neg M_2^s) \vee (\neg M_1^s \wedge M_2^s)$ is satisfiable, and if so, to provide an example that satisfies the combined predicate. If the solver returns an example, then we have found an input that satisfies one program's model, but not the other. If we had perfectly and fully modeled each program, and perfectly specified the post-condition to be that "the input results in an equivalent output state", then this input would be guaranteed to produce an equivalent output state in one program, but not the other.

However, since the models extracted in Section 5.3 only consider one execution path, then, as illustrated in Figure 5.1, it is possible that while an input does not satisfy the symbolic predicate generated for a server, it actually does result in an identical or equivalent output state, thus not triggering a deviation. This means that the input returned by the solver is only a *candidate deviation input* and we need an additional validation phase to check whether those candidate deviation inputs trigger a deviation. Note that we can query the solver for multiple candidate deviation inputs, each time requiring the new candidate input to be different than the previous ones.

5.4.2 Validation

The validation phase checks each candidate deviation input to determine whether it actually drives both implementations to different output states. To check whether a deviation has been found, each candidate deviation input is sent to the implementations being examined, and the outputs of the execution are compared to determine whether they result in equivalent or different output states. Determining if two output states are equivalent may require some domain knowledge about the protocol implemented by the programs. We use two methods to compare output states: monitoring the network traffic output by the program, and supervising its environment to detect unexpected states such as program crash.

For protocols that contain some type of status code in the response, such as HTTP in the Status-Line [65], each different value of the status code represents a different output state for the server. For those protocols that do not contain a status code in the response, such as NTP [146], we define a generic *valid state* and consider the server to have reached that state, as a consequence of an input, if it sends any well-formed response to the input, independently of the values of the fields in the response.

In addition, we define three special output states: a *fatal state* that includes any behavior that is likely to cause the server to stop processing future queries such as a crash, reboot, halt or resource

Server	Version	Type	Binary Size
Apache [4]	2.2.4	HTTP server	4,344 kB
MiniWeb [147]	0.8.1	HTTP server	528 kB
Savant [187]	3.1	HTTP server	280 kB
NetTime [157]	2.0 beta 7	NTP server	3,702 kB
Ntpd [164]	4.1.72	NTP server	192 kB

Table 5.1: Different server implementations used in our evaluation.

starvation, a *no-response state* that indicates that the server is not in the fatal state but still did not respond before a configurable timer expired, and a *malformed state* that includes any response from the server that is missing mandatory fields. This last state is needed because servers might send messages back to the client that do not follow the guidelines in the corresponding specification. For example several HTTP servers, such as Apache or Savant, might respond to an incorrect request with a raw message written into the socket, such as the string "IOError" without including the expected Status-Line such as "HTTP/1.1 400 Bad Request".

5.5 Evaluation

We have evaluated our deviation detection approach on two different protocols: HTTP and NTP. We selected these two protocols as representatives of two large families of protocols: text protocols (HTTP) and binary protocols (NTP). In particular, we use three HTTP server implementations and two NTP server implementations, as shown in Table 5.1. All the implementations are Windows binaries.

The original inputs, which we send to the servers during the model extraction phase were obtained by capturing a network trace from one of our workstations and selecting all the HTTP and NTP requests that it contained. For each HTTP request in the network trace, we send it to each of the HTTP servers, running inside the execution monitor, and output an execution trace that captures how the server processed the request. The execution monitor also records the output state observed at the end of the execution. If an input produces the same output state for all HTTP servers then we use the execution traces for each server as input to the model extraction phase. Otherwise we have already found a deviation! We proceed similarly for each NTP request. In Section 5.5.1, we show the deviations we discovered in the Web servers, and in Section 5.5.2, the deviations we discovered in the NTP servers.

Original request:
```
0000:   47 45 54 20  2F 69 6E 64  65 78 2E 68  74 6D 6C 20  GET /index.html
0010:   48 54 54 50  2F 31 2E 31  0D 0A 48 6F  73 74 3A 20  HTTP/1.1..Host:
0020:   31 30 2E 30  2E 30 2E 32  31 0D 0A 0D  0A           10.0.0.21....
```

Figure 5.3: One of the original HTTP requests we used to generate execution traces from all HTTP servers, during the model extraction phase.

5.5.1 Deviations in Web Servers

This section shows the deviations we found among three Web server implementations: Apache, MiniWeb, and Savant. We show results for a specific HTTP query, which we find to be specially important because it discovered deviations between different server pairs. Figure 5.3 shows this query, which is an HTTP GET request for the file `/index.html`. The post-condition used to identify the output state is based on the Status-Code in the HTTP reply and some additional special states as explained in Section 5.4.2. The output state for all three servers from the original input is: *Status-Code == 200*, which means that the Web Server returned the requested webpage.

Deviations detected. For each server we first extract the model from the execution trace, which represents how the server handled the original HTTP request shown in Figure 5.3. We call these models: M_A^{200}, M_S^{200}, M_M^{200} for Apache, Savant and MiniWeb respectively. For simplicity, we remove the output state and use M_A, M_S, M_M to identify the models. Then, for each of the three possible server pairs: Apache-MiniWeb, Apache-Savant and Savant-MiniWeb, we calculate the combined predicate as explained in Section 5.4.1. For example, for the Apache-MiniWeb pair, the combined predicate is $(M_A \wedge \neg M_M) \vee (M_M \wedge \neg M_A)$. To obtain more detailed information, we break the combined predicate into two separates queries to the solver, one representing each side of the disjunction. For example, for the Apache-MiniWeb pair, we query the solver twice: one for $(M_A \wedge \neg M_M)$ and another time for $(M_M \wedge \neg M_A)$.

Table 5.2 summarizes the deviations found for the three Web servers. Each cell of the table represents a different query to the solver, that is, half of the combined predicate for each server pair. Thus, the table has six possible cells. For example, the combined predicate for the Apache-MiniWeb pair, is shown as the disjunction of Cases 1 and 3. Out of the six possible cases, the solver returned unsatisfiable for three of them (Cases 1, 5, and 6). For the remaining cases, where the solver was able to generate at least one candidate deviation input, we show two numbers in the format X/Y. The X value represents the number of different candidate deviation inputs we obtained from the solver, and the Y value represents the number of these candidate deviation inputs that actually generated different output states when sent to the servers in the validation phase. Thus, the Y value represents the number of inputs that triggered a deviation.

	$\neg M_A$	$\neg M_M$	$\neg M_S$
M_A	N/A	Case 1: unsatisfiable	Case 2: 5/0
M_M	Case 3: 5/5	N/A	Case 4: 5/5
M_S	Case 5: unsatisfiable	Case 6: unsatisfiable	N/A

Table 5.2: Summary of deviations found for the HTTP servers, including the number of candidate input queries requested to the solver and the number of deviations found. Each cell represents the results from one query to the solver and each query to the solver handles half of the combined predicate for each server pair. For example Case 3 shows the results when querying the solver for $(M_M \wedge \neg M_A)$ and the combined predicate for the Apache-MiniWeb pair is the disjunction of Cases 1 and 3.

Candidate deviation input:
```
0000:   47 45 54 20  E8 69 6E 64  65 78 2E 68  74 6D 6C 20  GET .index.html
0010:   B4 12 02 12  90 04 02 04  0D 0A 48 6F  A6 4C 08 20  ..........Ho.L.
0020:   28 D0 82 91  12 E0 84 0C  35 0D 0A 0D  0A           (.......5....
```

Miniweb response:
```
HTTP/1.1 200 OK
Server: Miniweb
Cache-control: no-cache
[...]
```

Apache response:
```
HTTP/1.1 400 Bad Request
Date: Sat, 03 Feb 2007 05:33:55 GMT
Server: Apache/2.2.4 (Win32)
[...]
```

Figure 5.4: Example deviation found for Case 3, where MiniWeb's predicate is satisfied while Apache's isn't. The figure includes the candidate deviation input being sent and the responses obtained from the servers, which show two different output states.

In Case 2, none of the five candidate deviation inputs returned by the solver were able to generate different output states when sent to the servers, that is, no deviations were found. For Cases 3 and 4, all candidate deviation inputs triggered a deviation when sent to the servers during the validation phase. In both cases, the MiniWeb server accepted some input that was rejected by the other server. We analyze these cases in more detail next.

Applications to error detection and fingerprint generation. Figure 5.4 shows one of the deviations found for the Apache-MiniWeb pair. It presents one of the candidate deviation inputs obtained from the solver in Case 3, and the responses received from both Apache and MiniWeb when that candidate input was sent to them during the validation phase. The key difference is on the fifth byte of the candidate deviation input, whose original ASCII value represented a slash, indicating an absolute path. In the generated candidate deviation input, the byte has value 0xE8. We have confirmed that MiniWeb does indeed accept any value on this byte. So, this deviation reflects an error by MiniWeb: it ignores the first character of the requested URI and assumes it to be a slash, which is a deviation from the URI specification [12].

Candidate deviation input:
```
0000:   47 45 54 20  08 69 6E 64  65 78 2E 68  74 6D 6C 20  GET .index.html
0010:   09 09 09 09  09 09 09 09  0D 0A 48 6F  FF FF FF 20  ..........Ho...
0020:   09 09 09 09  09 09 09 09  09 0D 0A 0D  0A           .............
```

Miniweb response:
```
HTTP/1.1 200 OK
Server: Miniweb
Cache-control: no-cache
[...]
```

Savant response:
```
File not found
```

Figure 5.5: Example deviation found for Case 4, where MiniWeb's predicate is satisfied while Savant's isn't. The output states show that MiniWeb accepts the input but Savant rejects it with a malformed response.

Candidate deviation input:
```
0000:   47 45 54 20  2F 69 6E 64  65 78 2E 68  74 6D 6C 20  GET /index.html
0010:   48 54 54 50  2F 08 2E 31  0D 0A 48 6F  FF FF FF 20  HTTP/..1..Ho...
0020:   09 09 09 09  09 09 09 09  09 0D 0A 0D  0A           .............
```

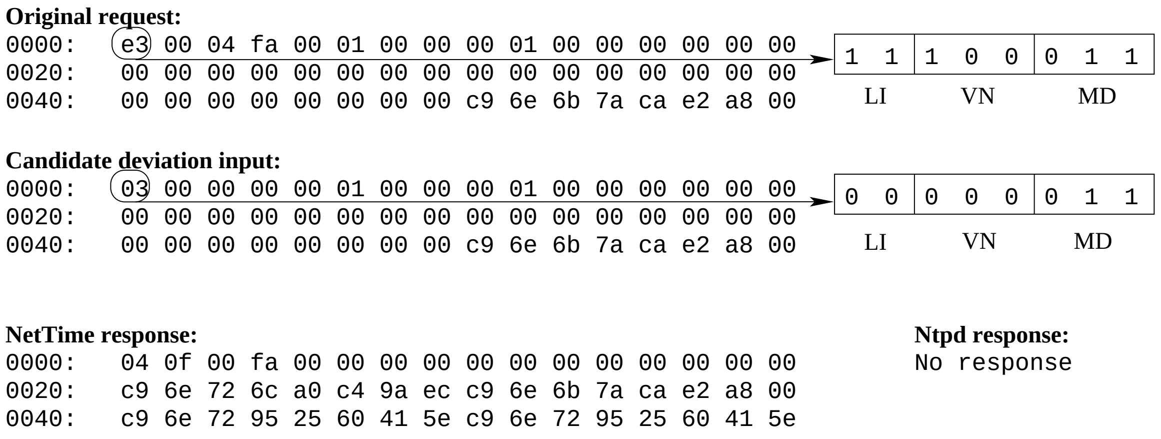
Miniweb response:
```
HTTP/1.1 200 OK
Server: Miniweb
Cache-control: no-cache
[...]
```

Savant response:
```
HTTP/1.1 400 Only 0.9 and 1.X requests supported
Server: Savant/3.1
Content-Type: text/html
[...]
```

Figure 5.6: Another example deviation for Case 4, between MiniWeb and Savant. The main different is on byte 21, which is part of the Version string. In this case MiniWeb accepts the request but Savant rejects it.

Figure 5.5 shows one of the deviations found for the Savant-MiniWeb pair. It presents one of the candidate deviation inputs obtained from the solver in Case 4, including the responses received from both Savant and MiniWeb when the candidate deviation input was sent to them during the validation phase. Again, the candidate deviation input has a different value on the fifth byte, but in this case the response from Savant is only a raw "File not found" string. Note that this string does not include the HTTP Status-Line, the first line in the response that includes the response code, as required by the HTTP specification and can be considered malformed [65]. Thus, this deviation identifies an error though in this case both servers (i.e. MiniWeb and Savant) are deviating from the HTTP specification.

Figure 5.6 shows another deviation found in Case 4 for the Savant-MiniWeb pair. The HTTP specification mandates that the first line of an HTTP request must include a protocol version string. There are 3 possible valid values for this version string: "HTTP/1.1", "HTTP/1.0", and "HTTP/0.9", corresponding to different versions of the HTTP protocol. However, we see that the candidate



Original request:

```
0000:   e3 00 04 fa 00 01 00 00 00 01 00 00 00 00 00 00    1 1 | 1 0 0 | 0 1 1
0020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     LI     VN     MD
0040:   00 00 00 00 00 00 00 00 c9 6e 6b 7a ca e2 a8 00
```

Candidate deviation input:

```
0000:   03 00 00 00 00 01 00 00 00 01 00 00 00 00 00 00    0 0 | 0 0 0 | 0 1 1
0020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     LI     VN     MD
0040:   00 00 00 00 00 00 00 00 c9 6e 6b 7a ca e2 a8 00
```

NetTime response:

```
0000:   04 0f 00 fa 00 00 00 00 00 00 00 00 00 00 00 00
0020:   c9 6e 72 6c a0 c4 9a ec c9 6e 6b 7a ca e2 a8 00
0040:   c9 6e 72 95 25 60 41 5e c9 6e 72 95 25 60 41 5e
```

Ntpd response:

```
No response
```

Figure 5.7: Example deviation obtained for the NTP servers. It includes the original request sent in the model extraction phase, the candidate deviation input output by the solver, and the responses received from the servers, when replaying the candidate deviation input. Note that the output states are different since NetTime does send a response, while Ntpd does not.

deviation input produced by the solver uses instead a different version string, ”HTTP/\b.1”. Since MiniWeb accepts this answer, it indicates that MiniWeb is not properly verifying the values received on this field. On the other hand, Savant is sending an error to the client indicating an invalid HTTP version, which indicates that it is properly checking the value it received in the version field. This deviation shows another error in MiniWeb’s implementation.

To summarize, in this section we have shown that our approach is able to discover deviations between multiple real-world HTTP Web servers. We have presented detailed analysis of three of them, and confirmed the deviations they trigger as errors. Out of the three deviations analyzed in detail, two of them can be attributed to be MiniWeb’s implementation errors, while the other one was an implementation error by both MiniWeb and Savant. The discovered inputs that trigger deviations can potentially be used as fingerprints to differentiate among these implementations.

5.5.2 Deviations in Time Servers

In this section we show the deviations found on the two NTP servers analyzed: NetTime [157] and Ntpd [164]. Again, for simplicity, we focus on a single request that we show in Figure 5.7. This request represents a simple query for time synchronization from a client. The request uses the Simple Network Time Protocol (SNTP) Version 4 protocol, which is a subset of NTP [146]. The output state for both servers on the original input shown in Figure 5.7 is the valid state that we presented in Section 5.4.2, which represents that the server sends a well-formed response to the input, independently of the values of the fields in the response.

Deviations detected. First, we generate the models for both servers: M_T and M_N for NetTime and Ntpd respectively from the execution traces captured using the original request shown in Figure 5.7. Since we have one server pair, we need to query the solver twice. In Case 7, we query the solver for $(M_N \wedge \neg M_T)$ and in Case 8 we query it for $(M_T \wedge \neg M_N)$. The solver returns unsatisfiable for Case 7. For Case 8, the solver returns 4 candidate deviation inputs. One of these candidate inputs triggers a deviation during the validation phase and is shown in Figure 5.7. It presents the candidate deviation input returned by the solver, and the response obtained from both NTP servers when that candidate deviation input was sent to them during the validation phase.

Applications to error detection and fingerprint generation. The results in Figure 5.7 show that the candidate deviation input returned by the solver in Case 8 has different values at bytes 0, 2 and 3. First, bytes 2 and 3 have been zeroed out in the candidate deviation input. This is not relevant since these bytes represent the “Poll” and “Precision” fields and are only significant in messages sent by servers, not in the queries sent by the clients, and thus are ignored by the servers.

The important difference is on byte 0, which is presented in detail on the right hand side of Figure 5.7. Byte 0 contains three fields: “Leap Indicator” (LI), “Version” (VN) and “Mode” (MD) fields. The difference with the original request is in the Version field. The candidate deviation input has a decimal value of 0 for this field (note that the field length is 3 bits), instead of the original decimal value of 4. When this candidate deviation input was sent to both servers, Ntpd ignored it, choosing not to respond, while NetTime responded with a version number with value 0. Thus, this candidate deviation input leads the two servers into different output states.

We check the specification for this case to find out that a zero value for the Version field is reserved, and according to the latest specification should no longer be supported by current and future NTP/SNTP servers [146]. However, the previous specification states that the server should copy the version number received from the client in the request, into the response, without dictating any special handling for the zero value. Since both implementations seem to be following different versions of the specification, we cannot definitely assign this error to one of the specifications. Instead, this example shows that we can identify inconsistencies or ambiguity in protocol specifications. In addition, we can use this query as a fingerprint to differentiate between the two implementations.

5.5.3 Performance

In this section, we measure the execution time and the output size at different steps in our approach. The results from the model extraction phase and the deviation detection phase are shown in Table 5.3 and Table 5.4, respectively. In Table 5.3, the column “Trace-to-IR time” shows the time spent in converting an execution trace into our IR program. The values show that the time spent to convert

Program	Trace-to-IR time	% of Symbolic Inst.	IR-to-predicate time	Model Size
Apache	7.6s	3.9%	31.9s	49,786
MiniWeb	5.6s	1.0%	14.9s	25,628
Savant	6.3s	2.2%	15.2s	24,789
Ntpd	0.073s	0.1%	5.3s	1,695
NetTime	0.75s	0.1%	4.3s	5,059

Table 5.3: Execution time and predicate size obtained during the model extraction phase.

	Input Calculation Time
Apache - MiniWeb	21.3s
Apache - Savant	11.8s
Savant - MiniWeb	9.0s
NetTime - Ntpd	0.56s

Table 5.4: Execution time needed to calculate a candidate deviation input for each server pair.

the execution trace is significantly larger for the Web servers, when compared to the time spent on the NTP servers. This is likely due to a larger complexity of the HTTP protocol, specifically a larger number of conditions affecting the input. This is shown in the second column as the percentage of all instructions that operate on symbolic data, i.e., on data derived from the input. The “IR-to-predicate time” column shows the time spent in generating a symbolic path predicate from the IR program. Finally, the “Model Size” column shows the size of the generated models, measured by the number of expressions that they contain. The model size shows again the larger complexity in the HTTP implementations, when compared to the NTP implementations.

In Table 5.4, we show the time used by the solver in the deviation detection phase to produce a candidate deviation input from the combined symbolic predicate. The results show that our approach is very efficient in discovering deviations. In many cases, we can discover deviations between two implementations in approximately one minute. Fuzz testing approaches are likely to take much longer, since they usually need to test many more examples.

5.6 Discussion

In this section we discuss extensions to the work presented in this chapter, as well as the relationship with the work that we present in the subsequent chapters of this thesis.

Covering rarely used paths. Some errors are hidden in rarely used program paths and finding them can take multiple iterations in our approach. For each iteration, we need an input that drives both implementations to equivalent output states. In this chapter, these protocol inputs were obtained from a network trace. Thus, the more different inputs contained in the network trace, the more paths we could potentially cover. In Chapter 6, we present white-box exploration techniques that enable automatically generating inputs that explore different execution paths in the program, starting from a single input seed. We could use those input generation techniques on both implementations to automatically find inputs that drive both implementations to the same output state. Those inputs could then be used as input to the deviation detection process.

Creating models that include multiple paths. In this chapter, we have presented techniques to generate models that contain a single execution path. Although such models have proved to be effective at finding deviations between protocol implementations, we expect higher coverage models to find a larger number of deviations. In Chapter 6, we present a model extraction technique based on white-box exploration that can produce models covering multiple execution paths. Then, in Chapter 7, we further refine our model extraction techniques to merge execution paths that share common constraints. High-coverage models that merge paths, when used for deviation detection, can identify a larger number of deviations and reduce the number of candidate inputs that do not pass the validation phase.

Addressing other protocol interactions. In this chapter we have evaluated our deviation detection approach over server implementations for protocols that use request/response interactions (e.g. HTTP, NTP), where we examine the request being received by a server program. Our approach applies to other scenarios as well. For example, with clients programs we could analyze the response being received by the client. In protocol interactions involving multiple steps, we could consider the output state to be the state of the program after the last step is finished.

5.7 Related Work

Symbolic execution & weakest pre-condition. Symbolic execution [106] has been used for a wide variety of problems including generating vulnerability signatures [21], automatic test case generation [78, 190], proving the viability of evasion techniques [113], and finding bugs in programs [33, 236]. Weakest pre-condition was originally proposed for developing correct programs from the ground up [61]. It has been used for different applications including finding bugs in programs [68] and for sound replay of application dialog [158].

Model checking. Chen et al. [36] manually identify rules representing ordered sequences of security-relevant operations, and use model checking techniques to detect violations of those rules in software. There is also a line of research using model checking to find errors in protocol implementations. Udrea et al. [211] use static source code analysis to check if a C implementation of a protocol matches a manually specified rule-based specification of its behavior. Musuvathi et.al. [154, 153] use a model checker that operates directly on C and C++ code and use it to check for errors in TCP/IP and AODV implementations. Chaki et al. [35] build models from implementations and check them against a specification model. Compared to our approach, these approaches need reference models to detect errors. Although these techniques are useful, our approach is quite different. Instead of comparing an implementation to a manually defined model, we compare implementations against each other. Another significant difference is that our approach works directly on binaries, and does not require access to the source code.

Protocol error detection. There has been considerable work on testing network protocol implementations, with heavy emphasis on tools for automatically detecting errors in network protocols using fuzz testing [92, 94, 132, 176, 200, 228, 100]. Fuzz testing [145] is a technique in which random or semi-random inputs are generated and fed to the program under study, while monitoring for unexpected program output, usually an unexpected final state such as program crash or reboot. Compared to fuzz testing, our approach is more efficient for discovering deviations since it requires testing far fewer inputs. It can detect deviations by comparing how two implementations process the same input, even if this input leads both implementation to equivalent states.

Protocol fingerprinting. There has also been previous research on protocol fingerprinting [43, 174] but available fingerprinting tools [162, 181, 6] use manually extracted fingerprints. More recently, automatic fingerprint generation techniques, working only on network input and output, have been proposed [29]. Our approach is different in that we use binary analysis to automatically generate the candidate inputs.

5.8 Conclusion

In this chapter we have presented a novel approach for deviation detection, the process of automatically finding deviations in the way that two different implementations of the same specification process their input. Our approach can automatically find an input that when sent to both implementations it drives them to different output states. It can be iterated to find multiple such inputs.

Our deviation detection approach enables and automates two important applications: error detection and fingerprint generation. It has several advantages over current solutions. First, it automatically builds models from the programs that implement the specification and finds deviations by comparing those models, without requiring access to a manually written model of the specification.

Second, it works directly on program binaries, without access to the source code of the implementations. Finally, because our models capture the internal processing of the implementations, our approach can find the needle (deviation) in the haystack (input space) without having to check each straw (input) individually. Thus, it can find deviations significantly faster than techniques that focus on random or semi-random probing.

We have built a prototype system to evaluate our techniques, and have used it to automatically discover deviations in multiple implementations of two different protocols: HTTP and NTP. Our results show that our approach successfully finds deviations after a few minutes. Those deviations include errors in at least one of the implementations and differences in the interpretation of the specification. Those deviations produce different, externally observable, output states in each implementation and can thus be used as fingerprints.

Chapter 6

Filtering-Failure Attack Generation

6.1 Introduction

There exists a broad class of security issues where a filter, intended to block malicious inputs destined for an application, incorrectly models how the application interprets those inputs. A *filtering-failure attack* is an evasion attack where the attacker takes advantage of those differences between the filter's and the application's interpretation of the same input to bypass the filter and still compromise the application.

One important class of filtering-failure attacks are *content-sniffing XSS attacks*. Content-sniffing XSS attacks are a class of cross-site scripting (XSS) attacks in which the attacker uploads some malicious content to a benign web site (e.g., a picture uploaded to Wikipedia, or a paper uploaded to a conference management system). The malicious content is accessed by a user of the web site, and is interpreted as *text/html* by the user's browser. Thus, the attacker can run JavaScript, embedded in the malicious content, in the user's browser in the context of the site that accepted the content. Such attacks are possible because the web site's upload filter has a different view than the user's browser about which content should be considered *text/html*. This discrepancy often occurs due to a lack of information or understanding by the web site's developers about the *content-sniffing algorithm* that runs in the browser and decides what MIME type to associate to some given content. For instance, some content that the web site's upload filter accepts because it interprets it as a PostScript document might be interpreted as HTML by the browser of the user downloading the content. There are other examples of filtering-failure attacks. For example, an Intrusion Detection System (IDS) may deploy a vulnerability signature to protect some unpatched application in the internal network. If the signature incorrectly models which inputs exploit the vulnerability in the application, then an attacker can potentially construct an input that is not matched by the IDS' signature but still exploits the application.

An important security problem is how to automatically find filtering-failure attacks and obtain inputs that demonstrate the attacks. Armed with such an attack input, one can demonstrate to the filter's developer the necessity of improving the filter so that it accurately resembles the application's behavior. In this chapter we propose an approach to automatically generate filtering-failure attacks. Our approach compares a model of the filter with a model of the application's functionality that the filter is designed to protect and automatically finds inputs that the filter considers benign but can still compromise the application.

To extract the models of the filter and the application, we can use model extraction techniques like the ones we introduced in Chapter 5. However, the single-path models that we used to find deviations in that chapter have limited coverage. In this chapter we propose a technique to extract a model of a fragment of binary code that captures multiple execution paths inside the code fragment. These multi-path models have significantly higher coverage than the single-path models that we used in Chapter 5. To extract multi-path models, we design *string-enhanced white-box exploration*. String-enhanced white-box exploration is similar in spirit to previous white-box exploration techniques used for automatic test case generation [33, 78, 79]. Unlike previous work, our technique incrementally builds a model from the explored paths and reasons directly about string operations, which provides a significant performance boost for programs that heavily use string operations.

An important characteristic of many security applications, such as an IDS signature matching engine and a content-sniffing algorithm in a browser, is that they rely heavily on string operations. Current white-box exploration techniques [78, 33, 79] are not efficient at dealing with such applications because they contain a large number of loops (potentially unbounded if they depend on the input). The intuition behind our string-enhanced white-box exploration technique is that we can enhance the exploration of programs that use strings, by reasoning directly about string operations, rather than reasoning about the individual byte-level operations that comprise those string operations. Reasoning directly about string operations significantly increases the coverage that the exploration achieves per unit of time.

In this chapter we demonstrate our approach to construct filtering-failure attacks by finding inputs that trigger content-sniffing XSS attacks. We use our string-enhanced white-box exploration technique to obtain a model for the closed-source content-sniffing algorithms of two different browsers: Internet Explorer 7 and Safari 3.1[1]. Then, we compare those models with the model of a web site's upload filter to automatically find content-sniffing XSS attacks. We use two different web site's upload filters: the one used by MediaWiki [135], an open-source wiki application used by many sites including the Wikipedia encyclopedia [224], and the one used by the HotCRP [87], a popular conference management Web application.

[1]Though much of Safari is open-source as part of the WebKit project [222], the content-sniffing algorithm in Safari is part of the closed-source *CFNetwork* library.

The MediaWiki filter is based on the MIME detection functions provided in PHP, which other sites may also use. Our approach finds 6 MIME types that an attacker can use to build content-sniffing XSS attacks against sites that use MediaWiki when the user accesses the site using Internet Explorer 7, and a different set of 6 MIME types that the attacker can use against users that employ Safari 3.1. For HotCRP, it finds that an attacker can use PostScript and PDF chameleon documents to launch content-sniffing XSS attacks against users that employ Internet Explorer 7.

The remainder of this chapter is organized as follows. In Section 6.2 we provide an overview of content-sniffing XSS attacks. Then, in Section 6.3 we formally define the problem of generating filtering-failure attacks and present an overview of our approach. Next, in Section 6.4 we detail our string-enhanced white-box exploration technique. We evaluate our approach in Section 6.5 and describe the related work in Section 6.6. Finally, we conclude in Section 6.7.

6.2 Content-Sniffing XSS attacks

For compatibility, every Web browser employs a *content-sniffing algorithm* that inspects the contents of HTTP responses and occasionally overrides the MIME type provided by the server. For example, these algorithms let browsers render the approximately 1% of HTTP responses that lack a *Content-Type* header. In a competitive browser market, a browser that guesses the "correct" MIME type is more appealing to users than a browser that fails to render these sites. Once one browser vendor implements content sniffing, the other browser vendors are forced to follow suit or risk losing market share.

If not carefully designed for security, a content-sniffing algorithm can be leveraged by an attacker to launch *content-sniffing XSS attacks*, a type of cross-site scripting (XSS) attacks. We illustrate content-sniffing XSS attacks by describing an attack against the HotCRP conference management system. Suppose a malicious author uploads a paper to HotCRP in PostScript format. By carefully crafting the paper, the author can create a *chameleon* document that both is valid PostScript and contains HTML (see Figure 1.4). HotCRP accepts the chameleon document as PostScript, but when a reviewer attempts to read the paper using Internet Explorer 7, the browser's content-sniffing algorithm treats the chameleon as HTML, letting the attacker run a malicious script in HotCRP's security origin. The attacker's script can perform actions on behalf of the reviewer, such as giving the paper a glowing review and a high score.

6.2.1 Background

In this section, we provide background information about how servers identify the type of content included in an HTTP response. We do this in the context of a Web site that allows its users to upload content that can later be downloaded by other users, such as in a photograph sharing or a conference management site.

Content-Type. HTTP identifies the type of content in uploads or downloads using the *Content-Type* header. This header contains a *MIME type*[2] such as *text/plain* or *application/postscript*. When a user uploads a file using HTTP, the server typically stores both the file itself and a MIME type. Later, when another user requests the file, the Web server sends the stored MIME type in the *Content-Type* header. The browser uses this MIME type to determine how to present the file to the user or to select an appropriate plug-in.

Some Web servers (e.g., old versions of Apache [5]) send the wrong MIME type in the *Content-Type* header. For example, a server might send a GIF image with a *Content-Type* of *text/html* or *text/-plain*. Some HTTP responses lack a *Content-Type* header entirely or contain an invalid MIME type, such as */* or *unknown/unknown*. To render these Web sites correctly, browsers use *content-sniffing* algorithms that guess the "correct" MIME type by inspecting the contents of HTTP responses.

Upload filters. When a user uploads a file to a Web site, the site has three options for assigning a MIME type to the content: (1) the Web site can use the MIME type received in the *Content-Type* header; (2) the Web site can infer the MIME type from the file's extension; (3) the Web site can examine the contents of the file. In practice, the MIME type in the *Content-Type* header or inferred from the extension is often incorrect. Moreover, if the user is malicious, neither option (1) nor option (2) is reliable. For these reasons, many sites choose option (3).

6.2.2 Content-Sniffing XSS Attacks

When a Web site's upload filter differs from a browser's content-sniffing algorithm, an attacker can often mount a *content-sniffing XSS attack*. In a content-sniffing XSS attack, the attacker uploads a seemingly benign file to an honest Web site. Many Web sites accept user uploads. For example, photograph sharing sites accept user-uploaded images and conference management sites accepts user-uploaded research papers. After the attacker uploads a malicious file, the attacker directs the user to view the file. Instead of treating the file as an image or a research paper, the user's browser treats the file as HTML because the browser's content-sniffing algorithm overrides the server's MIME type. The browser then renders the attacker's HTML in the honest site's security origin, letting the attacker steal the user's credentials for the site or transact with the site on behalf of the user.

[2]Multipurpose Internet Mail Extensions (MIME) is an Internet standard [70,71,150] originally developed to let email include non-text attachments, text using non-ASCII encodings, and multiple pieces of content in the same message. MIME defines MIME types, which are used by a number of protocols, including HTTP.

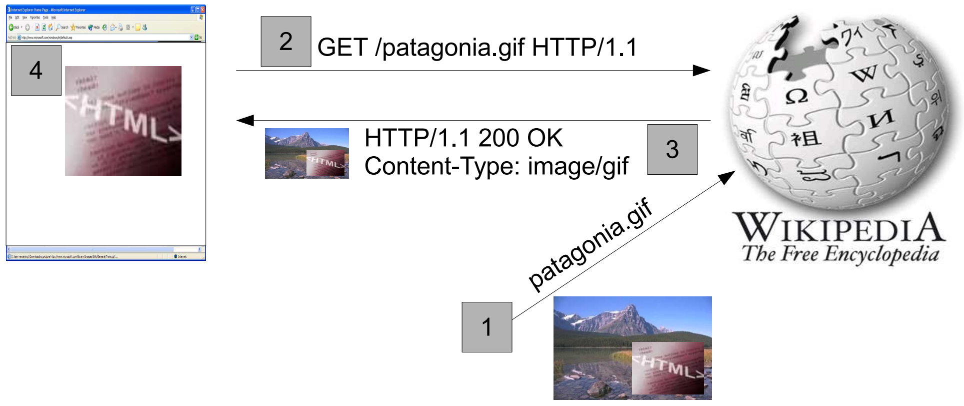


Figure 6.1: An example content-sniffing XSS attack on Wikipedia and a user of Internet Explorer 7. The numbered boxes show the sequence of events: 1) the attacker uploads a GIF/HTML chameleon to Wikipedia, 2) the user request the file, 3) the Web server delivers the content, and 4) the browser treats the chameleon as HTML and runs the attacker's JavaScript.

To mount a content-sniffing XSS attack, the attacker must craft a file that will be accepted by the honest site and be treated as HTML by the user's browser. Crafting such a file requires exploiting a mismatch between the site's upload filters and the browser's content-sniffing algorithm. A *chameleon* document is a file that both conforms to a benign file format (such as PostScript) and contains HTML. Most file formats admit chameleon documents because they contain fields for comments or metadata (such as EXIF [95]). Site upload filters typically classify documents into different MIME types and then check whether that MIME type belongs to the site's list of allowed MIME types. These sites typically accept chameleon documents because they are formatted correctly. The browser, however, often treats a well-crafted chameleon as HTML.

The existence of chameleon documents has been known for some time [182]. Recently, security researchers have suggested using PNG and PDF chameleon documents to launch XSS attacks [115, 80, 86], but these researchers have not determined which MIME types are vulnerable to attack, which browsers are affected, or whether existing defenses actually protect sites.

6.3 Problem Definition and Approach Overview

In this section, we first define the problem of finding filtering-failure attacks, then we present our running example, and finally we give an overview of our approach.

6.3.1 Problem Definition

Given a filter and the application that the filter tries to model, a filtering-failure attack is an input that is considered safe by the filter and can potentially be harmful for the application. Thus, a filtering-failure attack is an evasion attack that bypasses the filter and still can compromise the application. A filter can be modeled as a boolean predicate ($M^{safe}_{filter}(x)$) on an input x, which returns true if the input x is considered safe and false if the input is considered dangerous. In our approach the application's processing of the input is also modeled as a boolean predicate that captures all inputs to the program $x \in I$ that cause the program to reach a particular output state. The exact semantics of the output state depend on the application. We explain how to model the application for a content-sniffing XSS attack next.

The content-sniffing algorithm (CSA) in the user's browser can be modeled as a deterministic multi-class classifier that takes as input the payload of an HTTP response, the URL of the request, and the response's *Content-Type* header, and produces as output a MIME type for use by the browser. This multi-class classifier can be split into binary classifiers, one per MIME type returned by the content-sniffing algorithm, where each binary classifier is a model that returns true if the payload of the HTTP response is considered to belong to that MIME type and false otherwise (for instance $M^{text/html}_{csa}(x)$, or $M^{html}_{csa}(x)$ for brevity).

For a content-sniffing XSS attack, we seek inputs that are accepted by the web site's upload filter and for which the content-sniffing algorithm of the browser outputs a MIME type that can contain active content, such as *text/html* [3]. Thus, we can model the browser's content-sniffing algorithm as a binary classifier that returns true if the HTTP payload is considered HTML, $M^{html}_{csa}(x)$. To find a content-sniffing XSS attack that is accepted by the web site's upload filter and interpreted as HTML by the browser, we construct the following query: $M^{safe}_{filter}(x) \wedge M^{html}_{csa}(x)$. If the solver returns an input that satisfies such query, then we have found a content-sniffing XSS attack.

6.3.2 Running example

Figure 6.2 shows an example content-sniffing algorithm (*sniff*) that takes as input the proposed MIME type (*ct*) and the content (*data*), and returns a suggested MIME type. It sniffs an HTML document when the proposed MIME type is *text/plain* and JPEG and GIF images if the proposed MIME type is *application/octet-stream*. A possible content-sniffing XSS attack for this algorithm would require a Content-Type of *text/plain* and the content to contain the string <*html*>, because that is the only option to return a MIME type that can contain active code. An attacker could use the following input to try to bypass an upload filter in a website and run JavaScript in the browser:

```
CT: text/plain
DATA: GIF89a<html><script>alert("XSS");</script></html>
```

[3]Other MIME types that can run active content are *application/pdf*, *application/x-msdownload*, and *application/x-shockwave-flash*. For simplicity, we focus on content-sniffing XSS attacks, where the attacker embeds JavaScript in some content that the content-sniffing algorithm interprets as HTML. To consider multiple MIME types we can simply create a disjunction of the models: $M^{html}_{csa}(x) \vee M^{flash}_{csa}(x) \vee M^{pdf}_{csa}(x)$.

```
const char* text_mime="text/plain",binary_mime="application/octet-stream";
const char* html_mime="text/html",gif_mime="image/gif",jpeg_mime="image/jpeg";

const char* sniff(char *ct, char *data) {
  // Sniff HTML from text/plain
  if (strcmp(ct,text_mime) == 0) {
    if (strstr(data,"<html>") != 0) return html_mime;
    else return text_mime;
  }
  // Sniff GIF, JPEG from application/octet-stream
  if (strcmp(ct,binary_mime) == 0) {
    if ((strncasecmp(data,"GIF87a",6)==0) || (strncasecmp(data,"GIF89a",6)==0))
      return gif_mime;
    if ((data[0] == 0xFF) && (data[1] == 0xD8))
      return jpeg_mime;
  }
  return NULL;
}
```

Figure 6.2: Our running example, a simple content-sniffing algorithm that takes as input the proposed MIME type and the raw data, and returns a suggested MIME type.

6.3.3 Approach

In this section we provide an overview of our filtering-failure attack generation approach. We do so in the context of context-sniffing XSS attacks. Content-sniffing XSS attacks are particular to a web site and the browser that the user employs to access that web site. Our approach to generate content-sniffing XSS attacks is to first build a model of the web site's upload filter and the content-sniffing algorithm (CSA) in the user's browser. The model for the web site's upload filter $M_{filter}^{safe}(x)$ is a boolean predicate that returns true if the input x is considered safe and accepted by the web site and false if the input is considered dangerous and rejected. The model of the content-sniffing algorithm in the browser is a boolean predicate M_{csa}^{html} that captures all contents in an HTTP response that are classified as HTML. Armed with both models, we find content-sniffing XSS attacks by querying a solver for an input that satisfies $M_{filter}^{safe}(x) \wedge M_{csa}^{html}(x)$. If the solver returns such an input we have found a content-sniffing XSS attack, that is, an input that is accepted by the web site but interpreted as HTML by the user's browser.

The main challenge is creating high coverage models for the filter and the content-sniffing algorithm. To extract those models we employ *string-enhanced white-box exploration*. String-enhanced white-box exploration is similar in spirit to previous white-box exploration techniques used for automatic test case generation [33, 78, 79]. Unlike previous work, our technique incrementally builds a model from the explored paths and reasons directly about string operations. By reasoning directly about string operations, we increase the coverage achieved by the exploration per unit of time and improve the fidelity of our models.

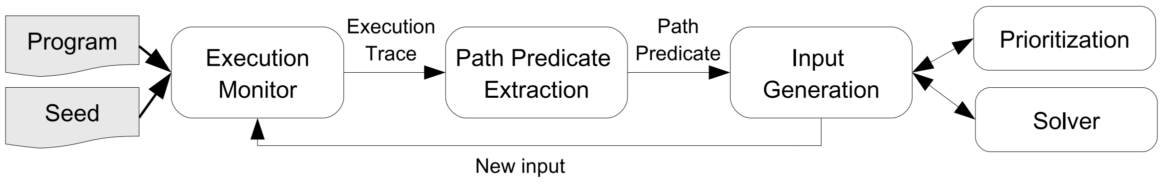


Figure 6.3: White-box exploration.

In this chapter, we use string-enhanced white-box exploration to extract models for the closed-source content-sniffing algorithms of two browsers: Internet Explorer 7 and Safari 3.1. For the upload filters used by MediaWiki and HotCRP, we manually extract their models from their publicly available source code. Next, we briefly describe the standard white-box exploration and introduce string-enhanced white-box exploration.

White-box exploration. White-box exploration is an iterative process that incrementally explores new execution paths in the program by generating new inputs that traverse those paths. Figure 6.3 illustrates the process. In each iteration, also called a *round* or a *test*, an input is sent to the program under analysis, running inside an execution monitor. From this execution a path predicate is produced that captures all the inputs that would follow the same execution path in the program than this input. Given the path predicate, the *input generator* produces a new input by negating one constraint in the path predicate and asking a solver to produce an input that satisfies the new predicate with the negated constraint. This process can be repeated for each constraint in the path predicate, generating from a single execution many new inputs. Since many inputs can be generated from each path predicate, and many path predicates will be generated during the exploration, the *prioritization module* is in charge of assigning priorities to the newly generated inputs and selecting the input with the highest priority to start a new round of the iterative process. In our white-box exploration approach the path predicate is extracted offline, from an execution trace, as presented in Section 5.3. Another possibility is to extract it during execution using forward symbolic execution in addition to the concrete run [78, 32]. This iterative process starts with an initial *seed* input, and runs until there are no more paths to explore, or a user-specified maximum run-time is reached.

String-enhanced white-box exploration. String-enhanced white-box exploration improves white-box exploration in two ways: 1) it includes string constraints in the path predicate, so that it can reason about string operations, and 2) it produces a model as the exploration progresses that is the disjunction of all the path predicates that reach the desired output state. Figure 6.4 illustrates string-enhanced white-box exploration and highlights the changes with respect to standard white-box exploration.

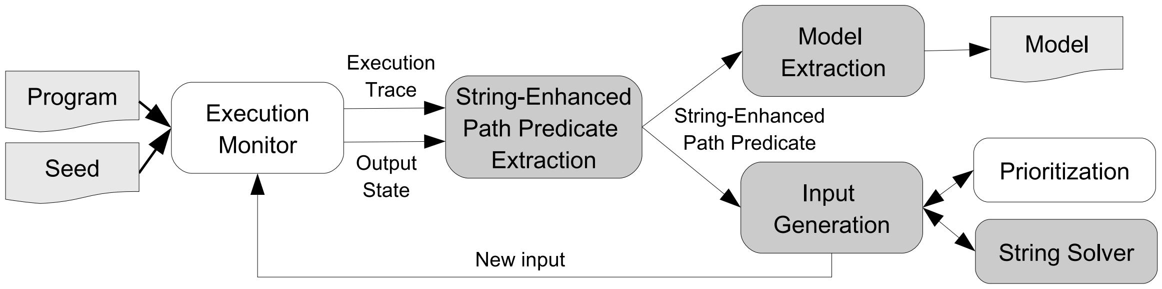


Figure 6.4: String-Enhanced White-box exploration. The gray modules have been modified from the standard white-box exploration.

In string-enhanced white-box exploration the path predicate contains constraints on the output of the string functions invoked by the program (e.g., *strlen* or *strcmp*), which replace the byte-level constraints that those string functions would otherwise introduce. This string-enhanced path predicate enables reasoning directly about the string operations, which in turn increases the coverage that the exploration achieves per unit of time. The increase in coverage is due to eliminating the time spent exploring inside the string functions.

String-enhanced white-box exploration excludes well-known string functions from the exploration and replaces the constraints generated inside those string functions with constraints on their output. This approach resembles the use of uninterpreted functions in compositional white-box exploration [3]. String functions are a sweet spot for composition in that 1) they appear in many programs, 2) some types of programs use them heavily (e.g., content-sniffing algorithms, parsers or filters), 3) they contain loops, which may be unbounded or have a very large bound since they depend on the function's input, 4) their prototype is usually known, or can be obtained with limited work, and 5) they are easy to reason about (as compared for example to system calls where one might have to reason about the underlying operating system or even the hardware). Rather than creating function summaries for the string functions as they are executed as in [76, 3], string-enhanced white-box exploration simply replaces them with string operators in an abstract syntax and then relies on a solver with support for a theory of strings to reason about those constraints.

6.4 String-Enhanced White-Box Exploration

This section details how our string-enhanced white-box exploration technique works. Overall, our string processing comprises four steps. First, we create a string-enhanced path predicate where constraints generated inside string functions have been replaced with constraints on the output of those string functions (Section 6.4.1). Then, the constraints on the output of the string functions are translated into an abstract string syntax (Section 6.4.2). Next, the system solves the constraints using a two-step approach that first represents each string as an array of some maximum length and a length variable, and then translates the abstract string operators into a representation that is understood by an off-the-shelf solver that supports theory of arrays and integers (Section 6.4.3). Finally, the answer from the solver is used to build an input that starts a new iteration of the exploration (Section 6.4.4).

Our string handling is designed to abstract the underlying representation of the strings so that it can be used with programs written in different languages. For example, in this chapter we apply it to the content-sniffing algorithm of Internet Explorer 7, which uses C strings (where strings are often represented as null-terminated character arrays), as well as to the content-sniffing algorithm of Safari 3.1, which uses a C++ string library (where strings are represented as objects containing a character array and an explicit length). One important characteristic of C/C++ strings is that one can operate with them using string functions such as *strlen* or *strcmp*, but also directly access the underlying array of characters. Thus, the string-enhanced path predicate may contain constraints on the output of string functions and constraints on the individual bytes that comprise them.

6.4.1 Generating the String-Enhanced Path Predicate

In this section, we present how the string-enhanced path predicate is generated. In a nutshell, adding string support to the path predicate comprises three steps: 1) introducing string symbols, which requires identifying the memory locations that hold the inputs strings when the function to model is invoked, 2) turning off symbolic execution inside the string functions, and 3) introducing string constraints by creating new symbols for the output of the string functions. For simplicity, we introduce this processing in the context of a symbolic execution monitor that performs both symbolic and concrete execution and outputs the string-enhanced path predicate. In reality, our implementation breaks this processing in two. First, the execution monitor collects the execution trace, as well as information about the string functions that were invoked during the run. Then, the path predicate extraction takes as input the execution trace and the string information and outputs the string-enhanced path predicate.

Introducing string symbols. The symbolic execution monitor uses the function hooks introduced in Chapter 2. To start the symbolic execution, the system sets a function hook for the function to be explored (i.e., the *sniff* function in our running example). When the function is called, the code stub performs the following operations: 1) reads the parameters of the function from the stack, 2) determines the length of the user-defined input strings (i.e., the *ct* and *data* parameters of the *sniff* function in our running example), 3) adds to the symbolic context the memory locations comprising each input string, and 4) sets a return hook. When the function returns, the return hook logs the return values of the function (i.e., the suggested MIME type) and stops the symbolic execution.

When creating a new string symbol, the representation of the string is abstracted. In particular, the function hook uses the function's prototype to determine whether the input strings are null-terminated arrays of characters or objects containing an array and an explicit length variable. If the string is null-terminated the location of the null-character does not become symbolic. If the string is an object with an explicit length variable then, in addition to the memory locations that comprise the string, the length variable also becomes a symbol.

Introducing string constraints. To introduce string constraints the system uses function hooks for some predefined string functions. The function hooks for the string functions differ from the functionality described above. To distinguish between both types of function hooks, we term the function hooks for string functions, *string function hooks*. A string function hook performs the following operations: 1) reads the parameters of the function from the stack, 2) checks if any of the parameters of the function is symbolic; if none are symbolic then it returns, 3) turns off symbolic execution inside the function, so that no constraints will be generated inside the string function, 4) sets up a return hook. When the string function returns, the return hook makes the return values of the function symbolic.

Currently, the execution monitor provides hooks for over 100 string functions for which prototypes are publicly available. The prototypes of those string functions can be found, among others, at the Microsoft Developer Network [138], the WebKit documentation [222], or the standard C library [93].

The user is expected to provide a string function hook for any function that is currently not available in the framework. When a program to be explored uses functions that have no publicly available prototype, some manual reverse engineering of the binary is needed to extract the function's prototype. For example, the content-sniffing algorithm in Internet Explorer 7 uses two string functions that have no publicly available prototype: *shlwapi.dll::Ordinal_151* and *shlwapi.dll::Ordinal_153*[4]. Our analysis found that *shlwapi.dll::Ordinal_151* is a case sensitive comparison of some maximum length, which can use the existing string function hook for *msvcrt.dll::strncmp*. Our analysis of *shlwapi.dll::Ordinal_153* uncovered that it is a case insensitive version of *shlwapi.dll::Ordinal_151*, which can use the existing string function hook for *msvcrt.dll::strncasecmp*. The time spent doing such analysis was close to an hour per function. Once obtained, the string function hooks are added to the framework so that they can be reused in the future.

[4]*shlwapi.dll* is the Shell Light Weight Utility Library, a Windows library that contains functions for URL paths, registry entries, and color settings.

String function classes. String functions are grouped into classes, where two string functions in the same class would generate the same path predicate if the specific function called in the source code was replaced with any other function in the same class. For example, *msvcrt.dll::strstr* and *msvcr71.dll::strstr* both belong to the same *STRSTR* class. Grouping string functions into classes and assigning the same symbol type to the output of all function in the same class reduces the number of different constraints that the system needs to translate into the abstract string syntax. Currently, the framework supports 14 classes of string functions: *STRSTR*, *STRCPY*, *STRNCPY*, *STRCMP*, *STRCASECMP*, *STRNCMP*, *STRLEN*, *STRNCASECMP*, *COMPARESTRING*, *CFEQUAL*, *STRCHR*, *MEMCHR*, *WCTOMB*[5], and *MBTOWC*[6].

To generate the string-enhanced path predicate, every time one of the predefined string functions is called during execution, the symbolic execution monitor introduces new symbols for the output of the string function. Then, when the program uses those symbols (e.g., in a comparison) a string constraint is introduced in the path predicate. The path predicate output by the execution monitor contains a mixture of string constraints (i.e., on the output of the string functions), and constraints on some of the bytes of the input strings, which can be generated by either functions that are not hooked or when the program directly accesses the bytes in the string, as shown in line 14 of our running example.

6.4.2 The Abstract String Syntax

We have designed an intermediate syntax that abstracts the representation of the strings and defines a common set of functions and predicates that operate on strings. This *abstract string syntax* represents the minimal interface we would like a solver, using strings as first-order types, to provide. Table 6.1 presents the functions and predicates that comprise our abstract string syntax. The strings in the abstract string syntax are immutable. Thus, operations such as modifying a string, copying a string, translating the string to upper case or concatenating two strings, always return a new string.

In our abstract string syntax each string can be seen as a variable-length array, where each element of the array has no encoding and is of fixed length[7]. Having no string encoding enables support for both binary and text strings. For simplicity, we term each element of the array a *character*, even if they may represent binary data. For text strings, an element of the array can be seen as a Unicode code-point[8]. Case-insensitive operators rely on the *chrupper* function, which forms the

[5]Converts a wide character to a multi-byte character

[6]Converts a multi-byte character to a wide character

[7]Our implementation uses 16-bit integers to represent a character. Although a 16-bit integer is not enough to hold all Unicode code points, it is enough for the applications we consider. Each character could be represented as a 32-bit integer if all Unicode code points are needed.

[8]A Unicode code-point is different from a *grapheme*, which is closer to what end-users consider as characters. For example a character with a dieresis (e.g., ä) is a grapheme, but could be encoded as two Unicode code points.

Functions		
(strlen String)	$S \rightarrow I$	(strlen s) returns the length of s
(substr String Int Int)	$S \times I \times I \rightarrow S$	(substr s i j) returns the substring of s starting at position i and ending at position j (inclusive)
(strcat String String)	$S \times S \rightarrow S$	(strcat s_1 s_2) returns the concatenation of s_1 and s_2
(strupper String)	$S \rightarrow S$	(strupper s) returns an uppercase version of s
(strncopy String Int)	$S \times I \rightarrow S$	(strncopy s i) returns a string of length i that equals the first i characters of s
(strfromwide String)	$S \rightarrow S$	(strfromwide s) returns a narrow character version of s
(strtostring Char)	$C \rightarrow S$	(strtostring c) returns a string containing only the character c
(chrat String Int)	$S \times I \rightarrow C$	(chrat s i) returns the character at position i in string s
(chrupper Char Char)	$C \rightarrow C$	(chrupper c) returns an uppercase version of c
Predicates		
(strcontains String String)	$S \times S \rightarrow B$	(strcontains s_1 s_2) returns true if s_2 is a substring of s_1 at any position
(strcontainsat String String Int)	$S \times S \times I \rightarrow B$	(strcontainsat s_1 s_2) returns true if s_2 is contained in s_1 starting at position i in s_1
(= String String)	$S \times S \rightarrow B$	(= s_1 s_2) returns true if s_1 is equal to s_2
(distinct String String)	$S \times S \rightarrow B$	(distinct s_1 s_2) returns true if s_1 is not equal to s_2
(strlt String String)	$S \times S \rightarrow B$	(strlt s_1 s_2) returns true if s_1 is lexicographically less-than s_2
(strle String String)	$S \times S \rightarrow B$	(strle s_1 s_2) returns true if s_1 is lexicographically less-or-equal s_2
(strgt String String)	$S \times S \rightarrow B$	(strgt s_1 s_2) returns true if s_1 is lexicographically greater-than s_2
(strge String String)	$S \times S \rightarrow B$	(strge s_1 s_2) returns true if s_1 is lexicographically greater-or-equal s_2
(strcaseequal String String)	$S \times S \rightarrow B$	(strcaseequal s_1 s_2) returns true if s_1 is equal to s_2 case-insensitive
(strcasedistinct String String)	$S \times S \rightarrow B$	(strcasedistinct s_1 s_2) returns true if s_1 is not equal to s_2 case-insensitive
(strcaselt String String)	$S \times S \rightarrow B$	(strcaselt s_1 s_2) returns true if s_1 is lexicographically less-than s_2 case-insensitive
(strcasele String String)	$S \times S \rightarrow B$	(strcasele s_1 s_2) returns true if s_1 is lexicographically less-or-equal s_2 case-insensitive
(strcasegt String String)	$S \times S \rightarrow B$	(strcasegt s_1 s_2) returns true if s_1 is lexicographically greater-than s_2 case-insensitive
(strcasege String String)	$S \times S \rightarrow B$	(strcasege s_1 s_2) returns true if s_1 is lexicographically greater-or-equal s_2 case-insensitive

Table 6.1: Abstract string syntax.

basis for *strupper*. Our current *chrupper* function uses the ASCII uppercase conversion (i.e., only code points U+0061 ('a') through U+007a ('z') have an uppercase version). We plan to enhance this function to represent the Unicode *uppercase* character property. Note that it is considered a valid operation to apply the case-insensitive functions to binary strings, as programs may (either incorrectly or abusing the semantics of the function) perform such operations.

All encoding is removed when converting to the abstract string syntax. For example, conversions from UTF-8 to UTF-16 and vice versa, used by the content-sniffing algorithm in Internet Explorer 7 for the *Content-Type* string, are handled during the translation to the abstract string syntax. Note that, while widening conversions (e.g., UTF-8 to UTF-16) are straightforward to handle, narrowing conversions (e.g., UTF-16 to UTF-8) can be lossy, and thus need a special conversion function (*strfromwide*). Our current implementation for *strfromwide* only handles conversions when all characters in the string belong to the ASCII charset, which is enough for programs that take as input ASCII strings.

String constraint	**Abstract String Syntax**
STRCMP(s_1,s_2) = 0 ; COMPARESTRING(s_1,s_2) = 2; CFEQUAL(s_1,s_2) = 1; s_2 = STRCPY(s_1)	= s_1 s_2
STRCMP(s_1,s_2) $\neq$ 0 ; COMPARESTRING(s_1,s_2) $\neq$ 2; CFEQUAL(s_1,s_2) = 0	distinct s_1 s_2
STRCMP(s_1,s_2) $<$ 0; COMPARESTRING(s_1,s_2) $<$ 2	strlt s_1 s_2
STRCMP(s_1,s_2) $>$ 0; COMPARESTRING(s_1,s_2) $>$ 2	strgt s_1 s_2
STRSTR(s_1,s_2) $\neq$ 0	strcontains s_1 s_2
STRSTR(s_1,s_2) $=$ 0	not (strcontains s_1 s_2)
STRCASECMP(s_1,s_2) = 0	strcaseequal s_1 s_2
STRCASECMP(s_1,s_2) $<$ 0	strcaselt s_1 s_2
STRCASECMP(s_1,s_2) $>$ 0	strcasegt s_1 s_2
STRNCMP(s_1,s_2,n) = 0	= (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRNCMP(s_1,s_2,n) $<$ 0	strlt (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRNCMP(s_1,s_2,n) $>$ 0	strgt (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRNCASECMP(s_1,s_2,n) = 0	strcaseequal (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRNCASECMP(s_1,s_2,n) $<$ 0	strcaselt (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRNCASECMP(s_1,s_2,n) $>$ 0	strcasegt (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRCHR(s,c) $\neq$ 0	strcontains s (strtostring c)
STRCHR(s,c) $=$ 0	not (strcontains s (strtostring c))
MEMCHR(s,c,n) $\neq$ 0	strcontains (substr s 0 $(n-1)$) (strtostring c)
MEMCHR(s,c,n) $=$ 0	not (strcontains (substr s 0 $(n-1)$) (strtostring c))
s_2 = STRNCPY(s_1,n)	= s_2 (substr s_1 0 $(n-1)$)
s_2 = MBTOWC(s_1)	= s_2 s_1
s_2 = WCTOMB(s_1)	= s_2 (strfromwide s_1)

Table 6.2: Translation of string constraints to the abstract string syntax.

Translating to the abstract string syntax. Table 6.2 presents the translation from the constraints generated on the output of the 14 classes of supported string functions to the abstract string syntax. Table 6.2 shows one of the benefits of using an abstract string syntax: constraints from functions with different prototypes but similar functionality (e.g, COMPARESTRING(s_1,s_2) $<$ 2, STRCMP(s_1,s_2) $<$ 0), can be translated to the same basic string operation (e.g., lexicographical less-than). Constraints on individual bytes are translated using the character extraction operator, *chrat*[9]. For example, the constraint `if (data[0] == 0xff)` $\{\ldots\}$ in line 14 of our running example, would be translated as `(chrat data 0) = 0xff`. This is possible because the execution monitor knows for each memory location if it belongs to a symbolic string and the offset into the string, which can be used to identify the character index. In our running example, if the function *sniff* is run with the following inputs:

```
CT: application/octet-stream
DATA: GIF89a\000\000
```

the string-enhanced path predicate translated to the abstract string syntax would be:

[9]Currently, we do not deal with unaligned accesses such as reading a single byte from a UTF-16 string, but such accesses could be translated as extracting the character corresponding to the offset being accessed and then masking the other byte.

Predicate	Translation
= s_1 s_2	$l(s_1) = l(s_2) \wedge \bigwedge_{i=0}^{i=l(s_1)-1} s_1[i] = s_2[i]$
strcaseequal s_1 s_2	$l(s_1) = l(s_2) \wedge \bigwedge_{i=0}^{i=l(s_1)-1} chrupper(s_1[i]) = chrupper(s_2[i])$
strcontains s_1 s_2	$\bigvee_{i=0}^{i=l(s_1)-1}(l(s_1) \geq l(s_2) + i) \wedge (\bigwedge_{j=0}^{j=l(s_2)-1} s_1[i+j] = s_2[j])$
= s_2 substr s_1 i j	$l(s_2) = j - i + 1 \wedge \bigwedge_{k=0}^{k=j-i} s_2[k] = s_1[i+k]$
= s strtostring c	$l(s) = 1 \wedge s[0] = c$
= s_2 (strfromwide s_1)	$l(s_2) = l(s_1) \wedge \bigwedge_{i=0}^{i=l(s_1)}(s_1[i] < 256 \wedge s_2[i] = s_1[i])$

Table 6.3: Predicate translation. For simplicity, the negation of the above predicates is not shown.

```
(distinct ct "text/plain") &&
(= ct "application/octet-stream") &&
(strcasedistinct (substr data 0 5) "GIF87a") &&
(strcaseequal (substr data 0 5) "GIF89a")
```

where the first constraint corresponds to the false branch in the conditional on line 6 of the running example, the second to the true branch of the conditional on line 11, and the final two correspond to the two clauses in the conditional on line 12 (false and true branches respectively). The value returned by the function is *image/gif*.

6.4.3 Solving the Constraints

We have designed our abstract string syntax so that it contains the predicates and functions we expect a solver that supports strings as first-class types to offer. However, at the beginning of this project, no publicly available solver supported strings as first-class types. To solve the string constraints in the path predicate we built a custom string constraint solver that leverages the fact that many off-the-shelf SMT solvers have support for array and integer theories. Our custom string constraint solver first represents each string (i.e., input strings plus any strings derived from them, for example through *strcpy*) as a pair of an array of some given maximum size and a length variable; and translates the operators in the abstract string syntax to constraints on the corresponding arrays and length variables. Then it uses STP to solve those constraints.

Simultaneous work reports on solvers that support a theory of strings [15, 85, 103]. Given our design, rather than translating the abstract string operations into a theory of arrays and integers, we could as well generate constraints in a theory of strings instead, benefiting from any performance improvements provided by these specialized solvers.

Upper bound on string length. Each string is represented as an array of some given maximum size (ml) and a length variable. The maximum string size is an important parameter. If it is too short the solver might not be able to solve some constraints. For example, in our running example if the maximum string size is set to 16 bytes, then the constraint generated in line 11 would be unsolvable since *ct* could not equal *application/octet-stream*. On the other hand the tighter the maximum length, the less time that it will take the solver to find a satisfying answer, if there is one. For some programs, such as content-sniffing algorithms, the maximum length of the input strings is known. For example, the maximum length of the content-sniffing buffer, which corresponds to the `data` string in our running example, is 1024 bytes for Safari 3.1, and 256 bytes for Internet Explorer 7 [9]. In practice, most signatures used by content-sniffing algorithms apply to the first few bytes in the content-sniffing buffer and we can use even a smaller upper bound. In this work we use $ml = 64$ bytes as the maximum string size.

Translating from the abstract string syntax. Table 6.3 shows how the constraints on the output of the string functions are translated to the theory of arrays and integers. Each string variable s is represented by its length $l(s)$ and an array of bytes $s[i]$, where the index i ranges from 0 to $ml - 1$. The maximum length ml is a translation-time constant, but the lengths $l(s)$ may not be, so unless the length of a string is constant, bounds that are shown involving $l(s)$ are in fact translated by expanding them up to a bound based on ml and guarding with additional conditions on $l(s)$. Note that the translation shown for strfromwide is restricted to the case of 8-bit code-points; a more complex translation would be needed for applications that use characters with longer encodings (e.g., an Arabic character in Unicode requires two bytes). For example, the constraint:

```
distinct ct "text/plain"
```

would be translated as:

$$\neg((ct\_len = 10) \wedge (ct[0] = `t`) \wedge (ct[1] = `e`) \wedge (ct[2] = `x`) \wedge (ct[3] = `t`) \wedge (ct[4] = `/`) \wedge (ct[5] = `p`) \wedge (ct[6] = `l`) \wedge (ct[7] = `a`) \wedge (ct[8] = `i`) \wedge (ct[9] = `n`))$$

where ct_len is the length integer that represents the length of the ct array. Note that the solver only understands about integers, but we use the text representation of the character here for the reader's benefit (e.g., the constraint would use 0x74 instead of 't').

Similarly, the constraint:

```
strcaseequal (substr data 0 5) "GIF89a"
```

would be translated as:

$$(ct\_len \geq 6) \wedge (chrupper(ct[0]) = `G`) \wedge (chrupper(ct[1]) = `I`) \wedge (chrupper(ct[2]) = `F`) \wedge (chrupper(ct[3]) = `8`) \wedge (chrupper(ct[4]) = `9`) \wedge (chrupper(ct[5]) = `a`)$$

Additional constraints. The translation introduces some additional constraints to each query to the solver. For each input string defined by the user, it adds a constraint to force the length of the string to be between zero and the predefined maximum length of the string, $0 \leq l(s) \leq ml(s)$. In addition, for ASCII strings it adds constraints to force each byte in the string to belong to the ASCII

```
HTTP/1.1·200·OK\n
Server:·Apache/2.2.4·(Fedora)\n
Content-Type:·text/plain\n\n
<html>·This· is·html·</html>\n\n
```

Figure 6.5: A complete input with the input strings highlighted.

charset, $\bigwedge_{i=0}^{i=ml(s)-1} 0 \leq s[i] \leq 127$. A special case happens when converting to the abstract string syntax a string constraint from a function that assumes the input strings to be null-terminated, such as the string functions in the C library. In this case the execution monitor adds some additional constraints to the path predicate to exclude the null character from the possible code values. This prevents the solver from producing inputs that actually violate the generated length constraints. For example, the condition in line 7 in our running example produces the following constraint:

```
strcontains ct "<html>"
```

If the null character is allowed to be part of the *ct* string, then the solver could return the following satisfying assignment for *ct*: $l(ct) = 8 \wedge s[0] = `a` \wedge s[1] = `\backslash 0` \wedge s[2] = ` < ` \wedge s[2] = `h` \wedge s[3] = `t` \wedge s[4] = `m` \wedge s[5] = `l` \wedge s[6] = ` > `$. Given the null terminated representation expected by *strstr*, that string would have an effective length of 1 character, and the generated input would not traverse the true branch of the conditional.

6.4.4 Input Generation

Once the solver returns a satisfying assignment for a query, the system needs to generate a new input that can be sent to the program, so that another round of the exploration can happen. However, the values from the symbolic strings might not completely define a program input. For example, Figure 6.5 shows a complete input used in the exploration of the content-sniffing algorithm of Safari 3.1, where the inputs strings are highlighted and the spaces have been replaced by dots. In this case, the program input is generated by querying the solver for an $http$ input that satisfies:

```
= http (strcat(strcat(strcat(strcat "HTTP/1.1 200 OK\n..." ct) "\n\n")
        data) "\n\n")
```

Generating an input that reaches the entry point. The function being explored might run in the middle of some longer execution. To guarantee that the generated inputs will reach the function under study, we need to add all constraints on the input strings generated by the code that executes before the function under study, as additional constraints to each query to the solver. For example, when analyzing the content-sniffing algorithm in Safari, we need to add any constraints on the Content-Type header or the HTTP payload that occur in the execution before the content-sniffing algorithm is called. To identify such constraints we run the execution monitor making the whole HTTP message symbolic[10]. All constraints on the input strings before the call to the content-sniffing algorithm are included as additional constraints. Such constraints may include, among others, parsing constraints that require the MIME type string not to contain any HTTP delimiters such as end of line characters, or constraints that force the Content-Type value to be one of a list of MIME types that trigger the content-sniffing algorithm.

6.5 Evaluation

In this section we present our evaluation results. First we introduce our setup. Then we show statistics from the models of the content-sniffing algorithms in two popular browsers that we extract using string-enhanced white-box exploration and compare the coverage of string-enhanced white-box exploration with that of byte-level white-box exploration. Next, we use the extracted models as well as models for upload filters to automatically generate chameleon documents that can be used to launch attacks. Finally, we detail two content-sniffing XSS attacks that affect the Wikipedia web site and the HotCRP conference management Web application.

6.5.1 Setup

We have extracted models from the content-sniffing algorithm of two major browsers, for which source code is not available: Safari 3.1 and Internet Explorer 7. In both cases we have evaluated the browser running on a Windows XP Service Pack 3 operating system.

In addition, we have manually written a model for the signatures used by the Unix *file* tool [66]. The Unix *file* tool is an open-source command line tool, deployed in many Unix systems, which given a file outputs its MIME type and some associated information. The signatures of the Unix *file* tool are used by the MIME detection functions in PHP (e.g., *finfo_file*). Those functions in turn are used by the upload filter of many web sites. For example, the MIME detection functions from PHP are used by popular open-source code such as MediaWiki [136], which is used by Wikipedia to handle uploaded content.

As described in Section 6.4.1, a prerequisite for the exploration is to identify the prototype of the function that implements the content-sniffing algorithm, as well as any string functions used by that function, for which a hook is not already available. To this end we use available documentation, commercial off-the-shelf tools [90], as well as our own binary analysis tools [199]. We describe this step next.

[10]Since there may exist multiple paths to the content-sniffing algorithm, we might have to rerun this step with different inputs. One indication to rerun this step is if during the exploration the tool reports that some inputs are not reaching the content-sniffing algorithm (i.e., empty path predicates).

Model	**Seeds**	**Path count**	**% HTML paths**	**Avg. Paths per seed**	**Avg. Time per path**	**# Inputs generated**	**Avg. path depth**	**# blocks found**	**Avg. blocks per seed**
Safari 3.1	7	1558	12.4%	222.6	16.8 sec	7166	12.1	205	193.9
IE 7	7	948	8.6%	135.4	26.6 sec	64721	212.1	450	388.5

Table 6.4: Model statistics.

Content sniffing is performed in Internet Explorer 7 by the function *FindMimeFromData* available in the *urlmon.dll* library [143]. We obtain the function prototype, including the parameters and return values, from the Microsoft Developer Network (MSDN) documentation [141].

Although a large portion of Safari 3.1 is open-source as part of the WebKit project, the content-sniffing algorithm is implemented in *CFNetwork.dll*, the networking library in the Mac OS X platform, which is not part of the WebKit project. In addition to extracting the prototype of the content-sniffing algorithm, we also had to add to the execution monitor two string function hooks for functions that have a publicly available prototype: *CoreFoundation.dll::CFEqual* and *CoreFoundation.dll::CFStringCompare*. Since the *CoreFoundation.dll* library provides the fundamental data types, including strings, which underlie the MacOS X framework, these hooks can be reused by many other applications that use this framework.

6.5.2 Model Extraction

In this section we present some statistics about the models of the content-sniffing algorithms of Internet Explorer 7 and Safari 3.1, extracted using string-enhanced white-box exploration. We term the process of exploring from one seed until no more paths are left to explore, or a user-specified maximum run-time is reached, an *exploration run*.

Each model is created by combining multiple exploration runs, each starting from a different seed. To obtain the seeds we first select some common MIME types and then we randomly choose one file of each of those MIME types from the hard-drive of one of our workstations. For our experiments each exploration run lasts 6 hours and the seeds come from 7 different MIME types: application/java, image/gif, image/jpeg, text/html, text/vcard, video/avi, video/mpeg. The same seeds are used for both browsers.

Table 6.4 summarizes the extracted models. The table shows the number of seeds used in the exploration, the number of path predicates that comprise each model, the percentage of path predicates in the previous column where the content-sniffing algorithm returned the MIME type *text/html*, the average number of paths per seed, the average time in seconds needed to generate a path predicate, the number of inputs generated, the average number of branches in each path (i.e., the path depth), the number of distinct program blocks discovered during the complete exploration from the 7 seeds, and the average number of blocks discovered per seed.

The number of paths that return *text/html* is important because the disjunction of those paths forms the M_{csa}^{html} model, which we use in Section 6.5.4 to find content-sniffing XSS attacks. The content-sniffing algorithm in Safari 3.1 is smaller because it has signatures for 10 MIME types, while the content-sniffing algorithm in Internet Explorer 7 contains signatures for 32 different MIME types. This is shown in Table 6.4 by shorter path predicates that require less time to be produced. The longer path predicates for Internet Explorer 7 also explain why the number of inputs generated for Internet Explorer 7 is almost an order of magnitude larger than for Safari 3.1.

Exploring from multiple seeds helps increase the coverage for Internet Explorer 7 because the content-sniffing algorithm in Internet Explorer 7 decides which signatures to apply to the content depending on whether it considers the content to be text or binary data. Thus, it is more efficient to do one exploration run for 6 hours starting from a binary seed (e.g., application/pdf) and another exploration run for 6 hours from a text seed (e.g., text/html) than to do a single exploration run for 12 hours starting from either a binary or a text seed. We have not observed this effect in Safari 3.1. We discuss more about how to compute the number of blocks discovered in the next section.

6.5.3 Coverage

In this section we illustrate the increase in coverage per unit of time that string-enhanced white-box exploration exhibits compared to byte-level white-box exploration. First, we detail how we measure the number of blocks discovered and then present the coverage results.

Methodology. For each execution trace produced during the exploration, every time an instruction that transfers control is seen (e.g., an unconditional jump, conditional jump, call, or return instruction), the address of the next instruction to be executed is stored. This address represents the first instruction in a block (i.e., the *block address*). The number of distinct block addresses is our coverage metric. This approach may underestimate the number of blocks discovered[11], but gives a reasonable approximation of the blocks covered by the exploration without requiring static analysis of the binary to extract all basic blocks. A difference with fuzzing approaches is that we do not want to maximize coverage of the whole program, only of the function that implements the content-sniffing algorithm. Thus, we are not interested in measuring coverage in auxiliary functions such as memory allocation functions (e.g., malloc), string functions (e.g., strcmp), or synchronization functions for critical sections. Our goal is to count blocks inside the content sniffing function, as

[11]When compared to counting basic blocks in a control-flow graph, our approach may underestimate the number of basic blocks because one block found during execution could be represented as multiple basic blocks in the control-flow graph. This happens when some path contains a jump whose target location is in the middle of one block previously discovered dynamically. In the CFG this case counts as two basic blocks while dynamically, since we deal with each path separately, it only counts as one.

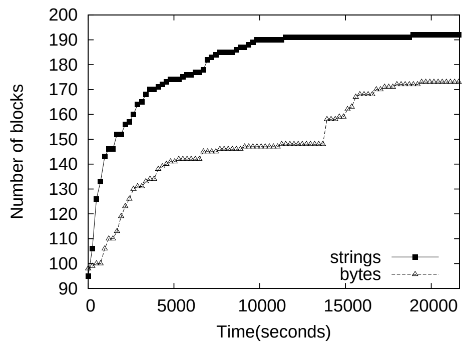


Figure 6.6: String-enhanced white-box exploration versus byte-level white-box exploration on the Safari 3.1 content-sniffing algorithm. Each curve represents the average number of blocks discovered for 7 exploration runs each starting from a different seed and running for 6 hours.

well as any other function that the algorithm invokes, which do not belong to the standard Windows libraries. Our tool approximates this behavior by automatically ignoring blocks inside functions that appear in the list of functions exported by name.

Coverage results. Figure 6.6 shows the number of blocks that the system discovers over time on the Safari 3.1 content-sniffing algorithm, when the exploration uses strings (square line) and when we disable the string processing and the path predicate only contains byte-level constraints (triangle line). Each curve represents the average number of blocks discovered for 7 exploration runs each starting from a different seed and lasting 6 hours. The *strings* curve corresponds to the 7 exploration runs from which the model of the content-sniffing algorithm in Safari 3.1 was extracted (shown in Table 6.4), while the *bytes* curve is the average of 7 byte-level exploration runs starting from the same seeds used for extracting the model.

The graph shows that the string-enhanced white-box exploration achieves higher coverage than the byte-level exploration on the same amount of time. Thus, it better employs the resources associated to the exploration. This happens because 61.6% of all byte-level constraints occur inside string functions. Thus, the byte-level exploration expends considerable time exploring inside the string functions, and no new blocks in the content-sniffing algorithm are discovered during that time.

6.5.4 Finding Content-Sniffing XSS Attacks

The first step to generate a content-sniffing XSS attack is to find an input that is accepted by the site's upload filter and interpreted by the content-sniffing algorithm in the browser as a privileged MIME type such as *text/html*. We call such an input a *chameleon* document. The chameleon document is basically a content-sniffing XSS attack without the malicious JavaScript payload. In this Section

Browser	HotCRP filter	Unix file tool
Internet Explorer 7	2	6
Safari 3.1	0	6

Table 6.5: Number of MIME types for which a chameleon is possible for the different combinations of content-sniffing algorithms and upload filters.

we show how to automatically find chameleon documents. Then, in the Section 6.5.5 we describe two examples of content-sniffing XSS attacks based on these results.

To generate chameleon documents, we use the models for the content-sniffing algorithms of Internet Explorer 7 and Safari 3.1, presented in Section 6.5.2. In addition, we manually create models for the HotCRP upload filter and the Unix *file* tool [66]. The signatures from the Unix *file* tool are used by the MIME detection functions in PHP, which in turn are used in the upload filter of multiple web sites such as MediaWiki [136]. Upload filters usually test the uploaded content against signatures for the different MIME types that should be accepted. If signature A matches the content then the MIME type associated with signature A will be sent in the *Content-Type* header when the content is delivered in an HTTP response. For example, the HotCRP upload filter accepts only content that it believes to be PDF or PostScript files. Our manually generated model for the HotCRP upload filter $M_{hotcrp}^{accept} = M_{hotcrp}^{pdf} \vee M_{hotcrp}^{ps}$ is the following predicate:

```
(strcaseequal "%PDF-" (substr content 0 4)) ||
(strcaseequal "%!PS-" (substr content 0 4))
```

If the first condition (M_{hotcrp}^{pdf}) returns true then the value of the *Content-Type* header in the HTTP response will be *application/pdf* and if the second condition (M_{hotcrp}^{ps}) returns true then it will be *application/postscript*. For each MIME type in the upload filter and for each browser, we query the solver whether a chameleon document can be produced. For example, to obtain a chameleon PostScript document that is interpreted as HTML by Internet Explorer 7 we query the solver for an input that satisfies $M_{IE}^{html} \wedge M_{hotcrp}^{postcript}$, which returns:

```
CT: application/postcript
DATA: %!PS-tRaTwad<Htmlswatarecz
```

Thus, the solver is able to produce a chameleon document in this case. Note that the first 5 bytes of the input correspond to the PostScript signature used by the HotCRP upload filter. Thus, this input is accepted by HotCRP as *application/postscript*. In addition, the input returned by the solver contains the substring "*<Html*", which satisfies the *text/html* signature used by the content-sniffing algorithm in Internet Explorer 7. Thus, this input is considered *text/html* by Internet Explorer 7 and if JavaScript code is included in the payload, it will be executed by the browser.

We repeat the above procedure for each MIME type in the upload filters that is supported by at least one of the browsers. Table 6.5 summarizes the results. For HotCRP, chameleon PDF and

PostScript documents can be created that will be interpreted as *text/html* by Internet Explorer 7. For the Unix *file* tool, chameleon documents that will be interpreted as *text/html* by Internet Explorer 7 can be created for 6 different MIME types: *application/postscript*, *audio/x-aiff*, *image/gif*, *image/tiff*, *text/xml*, and *video/mpeg*. Chameleon documents that will be interpreted as *text/html* by Safari 3.1 can also be created for 6 different MIME types: *application/postscript*, *audio/x-aiff*, *image/gif*, *image/png*, *image/tiff*, and *video/mpeg*. Next, we describe two chameleon documents in more detail.

Internet Explorer 7 and Unix file tool chameleon. Querying the solver for an input that is accepted as *audio/x-aiff* by the Unix *file* tool and is interpreted as *text/html* by the content-sniffing algorithm in Internet Explorer 7 returns the following answer:

```
CT: audio/x-aiff
DATA: <htmLpflAIFF\t\t\t\227\t\t\t\003\t\008\201\t
```

The first 5 bytes of the DATA string, *"<htmL"*, satisfy one of the HTML signatures used by Internet Explorer 7 for *text/html*, while the string *"AIFF"* in bytes 8 – 11 , satisfies the *audio/x-aiff* signature for the Unix *file* tool. This input would not match Internet Explorer's *audio/x-aiff* signature which is:

```
(strncmp(DATA,"MROF",4) == 0) ||
((strncmp(DATA,"FORM",4) == 0) &&
 ((strncmp(DATA[8],"AIFF",4) == 0) || (strncmp(DATA[8],"AIFC",4) == 0)))
```

Thus, the input will be considered *audio/x-aiff* by a filter based on the Unix *file* tool and *text/html* by Internet Explorer 7.

Safari 3.1 and Unix file tool chameleon. Querying the solver for an input that is accepted as *video/mpeg* by the Unix *file* tool and as *text/html* by Safari 3.1 returns:

```
CT: application/octet-stream
DATA: \000\000\001\187MmM\129\000\002\002TLT\001L\002\001\000<hTMl>e\000
```

Here, the solver returns an input where the first four bytes satisfy the *video/mpeg* signature of the Unix *file* tool, and the tag *<hTMl>* satisfies the *text/html* signature used by the content-sniffing algorithm in Safari 3.1. Because Safari 3.1 does not have a signature for *video/mpeg*, this input will be considered *video/mpeg* by a filter based on the Unix *file* tool and *text/html* by Safari 3.1.

6.5.5 Concrete Attacks

In this section, we detail two content-sniffing XSS attacks that affect two popular Web applications: HotCRP and Wikipedia. We implement and confirm the attacks using local installations of the sites.

HotCRP. HotCRP is a conference management Web application that lets authors upload their papers in PDF or PostScript format.[12] Before accepting an upload, HotCRP checks whether the file appears to be in the specified format. For PDFs, HotCRP checks that the first bytes of the file are `%PDF-` (case insensitive), and for PostScript, HotCRP checks that the first bytes of the file are `%!PS-` (case insensitive).

HotCRP is vulnerable to a content-sniffing XSS attack because HotCRP will accept the chameleon document in Figure 1.4 as PostScript but Internet Explorer 7 will treat the same document as HTML. To mount the attack, the attacker submits a chameleon paper to the conference. When a reviewer attempts to view the paper, the browser treats the paper as HTML and runs the attacker's JavaScript as if the JavaScript were part of HotCRP, which lets the attacker give the paper a high score and recommend the paper for acceptance.

Wikipedia. Wikipedia is a popular Web encyclopedia that lets users upload content in several formats, including SVG, PNG, GIF, JPEG, and Ogg/Theora [225]. The Wikipedia developers are aware of content-sniffing XSS attacks and have taken measures to protect their site. Before storing an uploaded file in its database, Wikipedia performs three checks:

1. Wikipedia checks whether the file matches one of the whitelisted MIME types. For example, Wikipedia's GIF signature checks if the file begins with `GIF`. Wikipedia uses PHP's MIME detection functions, which in turn use the signature database from the Unix *file* tool [66].
2. Wikipedia checks the first 1024 bytes for a set of blacklisted HTML tags, aiming to prevent browsers from treating the file as HTML.
3. Wikipedia uses several regular expressions to check that the file does not contain JavaScript.

Even though Wikipedia filters uploaded content, we uncover a subtle content-sniffing XSS attack. We construct the attack in three steps, each of which defeats one of the steps in Wikipedia's upload filter:

1. By beginning the file with `GIF88`, the attacker satisfies Wikipedia's requirement that the file begin with `GIF` without matching Internet Explorer 7's GIF signature, which requires that file begin with either `GIF87` or `GIF89`.
2. Wikipedia's blacklist of HTML tags is incomplete and contains only 8 of the 33 tags needed. To circumvent the blacklist, the attacker includes the string `<a href`, which is not on Wikipedia's blacklist but causes the file to match Internet Explorer 7's HTML signature.
3. To evade Wikipedia's regular expressions, the attacker can include JavaScript as follows:

[12] A conference organizer can disable either paper format.

```
<object src="about:blank"
  onerror="... JavaScript ...">
</object>
```

A file constructed in this way passes Wikipedia's upload filter but is treated as HTML by Internet Explorer 7. To complete the attack, the attacker uploads this file to Wikipedia and directs the user to view the file. These attacks demonstrate the importance of extracting precise models because the attacks hinge on subtle differences between the upload filter used by Wikipedia and the content-sniffing algorithm used by the browser.

The production instance of Wikipedia mitigates content-sniffing XSS attacks by hosting uploaded content on a separate domain. This approach does limit the severity of this vulnerability, but the installable version of Wikipedia, *MediaWiki*, which is used by over 750 Web sites in the English language alone [136], hosts uploaded user content on-domain in the default configuration and is fully vulnerable to content-sniffing XSS attacks. After we reported this vulnerability, Wikipedia has improved its upload filter to prevent these attacks.

6.6 Related Work

In this section we first present previous work on automatic test case generation and automatic signature generation, which is related to our work in that they also use white-box exploration or related symbolic execution techniques. Then, we introduce previous work that verifies security properties using software model checking techniques, which requires models of the programs to be verified, and can benefit from automated techniques to extract such models. Next, we describe previous research on cross-site scripting attacks, including content-sniffing XSS attacks, which we automatically find in this work. Finally, we introduce simultaneous work on solvers that support a theory of strings and outline some current defenses against content-sniffing XSS attacks.

Automatic test case generation. Previous work on automatic test case generation is another application of white-box exploration [78, 33, 79, 31]. There are two main differences between our model extraction technique using string-enhanced white-box exploration and previous work on automatic test case generation. First, the goal is different: the goal of automatic test case generation is to find bugs in a program, while the goal of our model extraction is to generate an accurate representation of a program that can be used for reasoning about its security implications. Second, the white-box exploration techniques used by previous work on automatic test case generation are not efficient on programs that heavily rely on string operations, which are the main target of our string-enhanced white-box exploration. There is related work on compositional approaches to white-box exploration that build function summaries as the exploration progresses and reuse the summaries if the function is later encountered in a context already covered in the summary [76, 3]. Our string-enhanced white-box exploration can be seen as a compositional approach that uses manually generated string function summaries.

Xu et al. [234] use source code annotations to augment white-box exploration with length abstractions for strings, which allows a tool to reason about the length of a string independent of its contents. Saxena et al. [188] propose loop-extended symbolic execution that broadens the coverage of dynamic symbolic execution in programs with loops, by introducing symbolic variables for the number of times each loop executes. These techniques could be used to replace the manually written string function summaries we use here, among other applications, but integrating them with a string decision procedure as in this report is future work. Previous work has also proposed improvements to white-box exploration techniques to reduce the number of paths that need to be explored (i.e., the path explosion problem) by using a compositional approach [77] or trying to identify parts of paths that have already been explored [16]. Such techniques can be combined with our string-enhanced white-box exploration technique to further enhance the exploration.

Automatic signature generation. Previous work on automatic signature generation produces symbolic-execution based vulnerability signatures directly from the vulnerable program binary [47, 21, 46, 23]. Such signatures model the conditions on the input required to exploit a vulnerability in the program. The difference between those signatures and our models is that vulnerability signatures try to cover only paths to a specific program point (namely, the vulnerability) rather than all paths inside some given function. A significant shortcoming of early proposals is that the signatures have low coverage, typically covering a single execution path [47]. More recent approaches have proposed to cover more execution paths by removing unnecessary conditions using path slicing techniques [46], iteratively exploring alternate paths to the vulnerability and adding them to the signature [46, 21], or using static analysis techniques [23]. Also related is work that examines the accuracy of the signatures used by a NIDS by generating exploit mutations and checking if those mutations still exploit the application [214, 185]. The inputs they find are filtering-failure attacks. Our approach can produce such inputs more efficiently by relying on models of the filter and the application, rather than on black-box probing.

Property verification. Model checking techniques can be used to determine whether a formal model (including of a program) satisfies a property [40]. They have been applied to security problems such as statically verifying security properties [36], verifying temporal-logic properties of an access control system [96], and evaluating attack scenarios in a network that contains vulnerable applications [183, 195]. But such techniques typically require the availability of a model, which limits the applicability to other security problems. In this paper we present a technique to automatically extract models from binaries, which can enable the application of model checking techniques to other applications.

Cross-site scripting attacks. Cross-site scripting (XSS) attacks, where an attacker injects active code (e.g., JavaScript) into HTML documents, are an important and widely studied class of attacks [98, 133, 83, 155, 48]. Content-sniffing XSS attacks are a class of XSS attacks where the attacker embeds executable code into different types of content. Previous references to content-sniffing XSS attacks focus on the construction of chameleon documents that Internet Explorer sniffs as HTML. A blog post from 2004 discusses a JPEG/HTML chameleon [182]. A 2006 disclosure describes a content-sniffing XSS attack that exploits an incorrect *Content-Type* header [86]. More recently, PNG and PDF chameleons have been used to launch content-sniffing XSS attacks [80, 172, 184]. Spammers have reportedly used similar attacks to upload text files containing HTML to open wikis [80]. Previously content-sniffing XSS attacks have been manually generated. In this work we show how to automatically generate content-sniffing XSS attacks.

String constraint solvers. Simultaneous work reports on solvers that support a theory of strings [15, 85, 103]. Even though during the course of this work no string constraint solver was publicly available, we designed our abstract string syntax so that it could use such a solver whenever available. Thus, rather than translating the abstract string operations into a theory of arrays and integers, we could easily generate constraints in a theory of strings instead, benefiting from any performance improvements provided by these specialized solvers.

Defenses. Current defenses that can ameliorate content-sniffing XSS attacks include transforming the uploaded content (e.g., converting a PNG image to JPEG format), disabling content-sniffing in the browser, and hosting the uploaded content in a separate domain so that the attacker can only gain access to the domain that hosts the content (e.g., upload.wikimedia.org) instead of the main domain (wikipedia.org). For an in-depth discussion on current defenses, their shortcomings, as well as novel defenses based on building more secure content-sniffing algorithms we refer the reader to our original paper [9].

6.7 Conclusion

In this chapter we have presented an automatic approach for generating filtering-failure attacks. Our approach extracts high coverage models from the filter, as well as the application's functionality that the filter is designed to protect. Then it compares those two models using a solver to produce inputs that are accepted by the filter but still can compromise the application.

We have proposed string-enhanced white-box exploration, a model extraction technique that extracts multi-path models from binary code. String-enhanced white-box exploration builds on previous white-box exploration techniques. Unlike previous work, it incrementally builds a model from the explored paths and reasons directly about string operations. By reasoning directly about string operations, it increases the coverage achieved by the exploration and improves the fidelity of the extracted models.

We have applied our approach to generate content-sniffing XSS attacks, a class of cross-site scripting attacks in which an attacker uploads some malicious content to a benign web site, which the user's browser interprets as HTML, enabling the attacker to run JavaScript, embedded in the malicious content, in the user's browser in the context of the site that accepted the content. We have used string-enhanced white-box exploration to extract models of the closed-source content-sniffing algorithms for two widely-used browsers: Internet Explorer 7 and Safari 3.1. We have used these models to automatically find content-sniffing XSS attacks that affect two popular Web applications: MediaWiki, an open-source wiki application used by many sites including Wikipedia, and the HotCRP conference management application.

Chapter 7

Protocol-Level Vulnerability Signature Generation

7.1 Introduction

Software vulnerabilities are prevalent, with over 4,500 new publicly disclosed vulnerabilities in 2009 [205, 56]. A popular defense for software vulnerabilities is *signature-based input filtering*, which has been widely deployed in Intrusion Prevention (IPS) and Intrusion Detection (IDS) systems. Signature-based input filtering matches program inputs against a set of signatures and flags matched inputs as attacks. It provides an important means to protect vulnerable hosts when patches are not yet available or have not yet been applied. Furthermore, for legacy systems where patches are no longer provided by the vendor, or critical systems where any changes to the code might require a lengthy re-certification process, signature-based input filtering is often the only practical solution to protect the vulnerable program.

The key technical challenge to effective signature-based defense is to automatically and quickly generate signatures that have zero false positives and zero false negatives, what we call *perfect signatures*. In addition, it is desirable to generate signatures without access to the source code. This is crucial to wide deployment since it enables third-parties to generate signatures for commercial-off-the-shelf (COTS) programs, without relying on software vendors, thus enabling a quick response to newly found vulnerabilities.

Due to the importance of the problem, many different approaches for automatic signature generation have been proposed. Early work proposed to generate *exploit-based signatures* using patterns that appeared in the observed exploits, but such signatures can have high false positive and negative rates [112, 105, 197, 159, 237, 118, 120, 119, 220]. More recently, researchers proposed to generate *vulnerability-based signatures*, which are generated by analyzing the vulnerable program and its execution and the actual conditions needed to exploit the vulnerability and can guarantee a zero false positive rate [47, 21].

Vulnerability-based signatures. A vulnerability is a point in a program where execution might "go wrong". We call this point the *vulnerability point*. A vulnerability is only exploited when a certain condition, the *vulnerability condition*, holds on the program state when the vulnerability point is reached. Thus, to exploit a vulnerability, the input needs to satisfy two conditions: (1) it needs to lead the program execution to reach the vulnerability point; (2) the program state needs to satisfy the vulnerability condition at the vulnerability point. We call the boolean predicate that denotes whether an input message will make the program execution reach the vulnerability point the *vulnerability point reachability predicate* (VPRP).

A vulnerability-based signature can be seen as model (as introduced in Section 5.2.1) where the output state is defined by the pair of the vulnerability point and the vulnerability condition. Thus, a vulnerability-based signature is simply a conjunction of the vulnerability point reachability predicate, which specifies the program inputs that reach the vulnerability point, and the vulnerability condition that needs to hold at the vulnerability point. Thus, the problem of automatically generating a vulnerability-based signature can be decomposed into two: identifying the output state formed by the vulnerability point and the vulnerability condition, and identifying the vulnerability point reachability predicate. While both problems are important, in this chapter we focus on how to generate vulnerability point reachability predicates. The problems of identifying the vulnerability point and the vulnerability condition have been addressed as part of a parallel project [188]. Note that a vulnerability point reachability predicate can also be used as a signature (i.e., with a `true` post-condition at the output state). However, such signatures can have false positives because there may be inputs that reach the vulnerability point but do not exploit the application.

Coverage is a key challenge. One important problem with early vulnerability-based signature generation approaches [47] is that the signatures only capture a single path to the vulnerability point (i.e., their VPRP contains only one path). However, the number of paths leading to the vulnerability point can be very large, sometimes infinite. Thus, such signatures are easy to evade by an attacker with small modifications of the original exploit message, such as changing the size of variable-length fields, changing the ordering of the fields (e.g., HTTP headers), or changing field values that drive the program through a different path to the vulnerability point. Acknowledging the importance of enhancing the coverage of vulnerability-based signatures, recent work tries to incorporate multiple paths into the VPRP either by static analysis [21, 23], or by dynamic analysis [46, 55]. However, performing precise static analysis on binaries is hard due to issues such as indirection, pointers and loops, and current dynamic analysis approaches rely on heuristic-based black-box probing, which is less effective at extending the coverage than white-box approaches.

In this paper, we propose *protocol-level constraint-guided exploration*, a new approach to automatically generate vulnerability point reachability predicates with high coverage. Our approach has 3 main characteristics: 1) it is based on white-box exploration (i.e., instead of heuristics-based exploration as in ShieldGen [55] and Bouncer [46]), 2) the white-box exploration works at the *protocol-level* and generates protocol-level signatures at the end, and 3) it effectively *merges* explored execution paths to remove redundant exploration. The three points seamlessly weave together and amplify each others benefit. By using white-box exploration, our approach significantly increases the effectiveness and efficiency of the exploration compared to previous heuristics-based approaches. By using protocol information to lift the symbolic constraints from the byte level to the protocol level, our approach reduces the exploration space for programs that use highly-structured protocols or file formats, and produces protocol-level signatures, which compared to byte-level signatures are more compact and cover variants of the exploits caused by variable-length fields and field reordering. By merging paths in the exploration, our exploration further reduces the exploration space, avoiding the exploration of duplicate paths that otherwise could make the exploration space increase exponentially.

Elcano. We have implemented protocol-level constraint-guided exploration in a signature generation tool called *Elcano*. We have evaluated the effectiveness of Elcano using 6 vulnerabilities on real-world programs. The generated signatures achieve perfect or close-to-perfect results in terms of coverage. Using a 6 hour time limit for the exploration, our approach discovered all possible paths to the vulnerability point for 4 out of the 6 vulnerabilities, thus generating a complete VPRP. For those four signatures, the generation time ranges from under one minute to 23 minutes. In addition, the resulting signatures are compact: the number of constraints in the resulting VPRP is in most cases small and those constraints are often small themselves. Compact signatures can be more easily understood by humans, which facilitates deployment.

Other applications. In addition to signature generation, a high coverage vulnerability point reachability predicate is useful for other applications such as exploit generation [22] and patch testing. For example, the Microsoft patch MS05-018 missed some paths to the vulnerability point and as a result left the vulnerability still exploitable after the patch [144]. This situation is not uncommon. A quick search on the CVE database returns 13 vulnerabilities that were incorrectly or incompletely patched [56]. Our technique could assist software developers to build more accurate patches. Furthermore, our protocol-level constraint-guided approach can increase the effectiveness of generating high-coverage test cases and hence be very valuable to software testing and bug finding.

7.2 Problem Definition and Approach Overview

In this section, we first introduce the problem of automatic generation of protocol-level vulnerability point reachability predicates, then present our running example, and finally give the overview of our approach.

7.2.1 Problem Definition

Automatic generation of protocol-level vulnerability point reachability predicates. Given a parser implementing a given protocol specification, the vulnerability point, and a seed input that reaches the vulnerability point, the problem of automatic generation of protocol-level vulnerability point reachability predicates is to automatically generate a predicate function F, such that when given some input mapped into a message field tree by the parser, F evaluates over the message field tree: if it evaluates to *true*, then the input is considered to be able to reach the vulnerability point, otherwise it is not.

Parser availability and specification quality. The problem of automatic generation of protocol-level vulnerability point reachability predicates assumes the availability of a parser implementing a given protocol or file specification. The parser given some input data can map it into fields, according to the specification, or fail if the input is malformed. In the latter case, the IDS/IPS could opt to block the input or let it go through while logging the event or sending a warning. Such a parser is available for common protocols (e.g., Wireshark [227]), and many commercial network-based IDS or IPS have such a parser built-in. In addition, recent work has shown how to create a generic parser that takes as input protocol specifications written in an intermediate language [173, 17].

The quality of the specification used by the parser matters. While obtaining a high quality specification is not easy, this is a one time effort, which can be reused for multiple signatures, as well as other applications. For example, in our experiments we extracted a WMF file format specification. At the time of writing, there are 37 vulnerabilities related to WMF in the CVE Database [56], where our specification could be reused. Similarly, an HTTP specification could be reused in over 1500 vulnerabilities. In Chapter 3, we propose techniques to automatically extract the format of undocumented protocols from the binary of a program that implements the protocol. Those techniques can be used when the protocol used by the vulnerable program has no public specification.

Vulnerability point availability. Our problem definition assumes that the vulnerability point is given. Note that the vulnerability point may be different than the program point where the abnormal behavior is detected. For example, an integer overflow may be detected when a program crashes due to a memory dereference that uses an invalid pointer. The invalid pointer was created earlier in

```
1  void service() {
2   char msgBuf[4096];
3   char lineBuf[4096];
4   int nb=0, i=0, sockfd=0;
5   nb=recv(sockfd,msgBuf,4096,0);
6   for(i = 0; i < nb; i++) {
7     if (msgBuf[i] == '\n')
8         break;
9     else
10      lineBuf[i] = msgBuf[i];
11  }
12  if (lineBuf[i-1] == '\r')
13    lineBuf[i-1] = '\0'
14  else lineBuf[i] = '\0';
15  doRequest(lineBuf);
16 }
17 void doRequest(char *lineBuf){
18  char vulBuf[128],uri[256];
19  char ver[256], method[256];
20  int is_cgi = 0;
21  sscanf(lineBuf,
22    "%255s %255s %255s",
23    method, uri, ver);
24  if (strcmp(method,"GET")==0 ||
25      strcmp(method,"HEAD")==0){
26    if strncmp(uri,"/cgi-bin/",
27      9)==0  is_cgi = 1;
28    else is_cgi = 0;
29    if (uri[0] != '/') return;
30    strcpy(vulBuf, uri);
31  }
32 }
```

Figure 7.1: Our running example.

the execution by adding the overflown integer to a pointer. In this case, the vulnerability point is the arithmetic instruction that overflows, rather than the instruction that dereferences the invalid pointer. Identifying the vulnerability point is part of a parallel project that aims to accurately describe the vulnerability condition [188]. Such vulnerability points could also be identified using previous techniques [160, 46].

Seed input availability. Our problem definition also assumes that a seed input that reaches the vulnerability point is given. Note that the seed input does not need to *exploit* the vulnerability, it only needs to *reach* the vulnerability point. This seed input enables focusing the exploration on paths that are related to the program functionality that contains the vulnerability. If such seed input is not available, we could use any other program input to start the exploration. In this case, the exploration would iterate (without adding any paths to the VPRP) until finding the first path that reaches the vulnerability point. However, it is hard to know when (or if) such path will be found.

7.2.2 Running Example

Figure 7.1 shows our running example. We represent the example in C language for clarity, but our approach operates directly on program binaries. Our example represents a basic HTTP server and contains a buffer-overflow vulnerability. In the example, the `service` function copies one line of data received over the network into `lineBuf` and passes it to the `doRequest` function that parses it into several field variables (lines 21–23) and performs some checks on the field values (lines 24–31). The first line in the exploit message includes the method, the URI of the requested resource, and the protocol version. If the method is GET or HEAD (lines 24–25), and the first character of the

URI is a slash (line 29), then the vulnerability point is reached at line 30, where the size of `vulBuf` is not checked by the `strcpy` function. Thus, a long URI can overflow the `vulBuf` buffer.

In this example, the vulnerability point is at line 30, and the vulnerability condition is that the local variable `vulBuf` will be overflowed if the size of the URI field in the received message is greater than 127. Therefore, for this example, the vulnerability point reachability predicate is: `((strcmp(FIELD_METHOD,"GET")==0)`$\vee$`(strcmp(FIELD_METHOD,"HEAD")==0))` $\wedge$ `(FIELD_URI[0] = '/')`. The vulnerability condition is: `length(FIELD_URI) > 127`, and the conjunction of the two is a perfect protocol-level signature.

7.2.3 Approach

In this chapter we propose a new approach to generate high coverage, yet compact, vulnerability point reachability predicates, called *protocol-level constraint-guided exploration*. Next, we give the motivation and an overview of the three characteristics that comprise our approach.

Constraint-guided. Previous dynamic approaches to generate vulnerability-based signatures use heuristics-based exploration [46, 55]. Heuristic-based exploration suffers from a fundamental limitation: the number of probes needed to exhaustively search the whole space is usually astronomical. In addition, an exhaustive search is inefficient as many probes end up executing the same path in the program. Thus, such approaches often rely on heuristics that are not guaranteed to significantly increase the signature's coverage and can also introduce false positives.

For example, ShieldGen [55] uses the specification of the protocol that corresponds to the exploit message to generate different well-formed variants of the original exploit. It uses various heuristics to create the variants and then checks whether any of the variants still exploits the vulnerability. ShieldGen's heuristics first assume that fields can be probed independently, and then for fixed-length fields it samples just a few values of each field, checking whether the vulnerability point is reached or not for those values. Probing each field independently means that constraints involving multiple fields cannot be found. Take the constraint `SIZE1 + SIZE2` $\leq$ `MSG_SIZE`, where `SIZE1` and `SIZE2` are length fields in the input, and `MSG_SIZE` represents the total length of the received message. The authors of ShieldGen acknowledge that their signatures cannot capture this type of constraints, but such constraints are commonly used by programs to verify that the input message is well-formed and failing to identify them will introduce either false positives or false negatives, depending on the particular heuristic.

False positives can be introduced because only a few random values of each field are tested. If all tested values for a field exploit the vulnerability then they consider that any field value would also exploit it. Imagine the case where the vulnerability is only exploited when `FIELD` $\leq$ `100`.

If all random probes tested have values smaller than 100, their approach incorrectly generalizes that any value of the field would exploit the vulnerability, when values larger than 100 would not. False negatives can be introduced because probing only a few sample values for each field is likely to miss constraints that are satisfied by only a small fraction of the field values. For example, a conditional statement such as `if (FIELD==10)` $\vee$ `(FIELD==20) then exploit, else safe`, where FIELD is a 32-bit integer, creates two paths to the vulnerability point. Finding each of these paths would require 2^{30} random probes on average to discover. Creating a signature that covers both paths is critical since if the signature only covers one path (e.g., `FIELD == 10`), the attacker could easily evade detection by changing FIELD to have value 20.

To overcome these limitations, we propose to use white-box exploration to increase the coverage of the final signatures by automatically discovering new paths that reach the vulnerability point and adding them to the VPRP. However, simply applying previous white-box exploration approaches [33, 78, 79] does not scale well to real-world programs that use complex, highly structure inputs such as protocols and file formats. In fact, in Bouncer [46] the authors acknowledge that they wanted to use white-box exploration but failed to do so due to the large number of paths that need to be explored and thus had to fall back to the heuristics-based probing approach. To make white-box exploration feasible and effective we have incorporated two other key characteristics into our approach as described below.

Protocol-level path predicates. Previous white-box exploration approaches generate *byte-level path predicates*, which are evaluated directly on the input bytes. Such byte-level path predicates in turn generate *byte-level signatures*, which are also specified on the input bytes. However, previous work has shown that signatures are better specified at the protocol-level instead of the byte level [237, 55]. We call such signatures *protocol-level signatures*.

Our contribution here is to show that, by lifting byte-level path predicates to *protocol-level path predicates*, so that they operate on protocol fields rather than on the input bytes, we can make white-box exploration scale with highly structured inputs, as using constraints at the protocol-level hugely reduces the number of paths to be explored compared to using byte-level constraints. The state reduction is achieved in two ways. First, the parsing logic often introduces huge complexity in terms of the number of execution paths that need to be analyzed. For example, in our experiments, 99.8% of all constraints in the HTTP vulnerabilities are generated by the parsing logic. While such parsing constraints need to be present in the byte-level path predicates, they can be removed in the protocol-level path predicates. Second, the byte-level constraints introduced by the parsing logic makes the VPRP match only inputs that have the same field structure as the seed exploit message, for example same size of the variable-length fields and same field sequence (when protocols such as HTTP allow fields to be reordered). Unless the parsing constraints are removed the resulting signature would be very easy to evade by an attacker by applying small variations to the field structure of the exploit message. Finally, the vulnerability point reachability predicates at the protocol level are smaller and easier to understand by humans.

Merging execution paths. As the exploration discovers new paths leading to the vulnerability point, they need to be added to the vulnerability point reachability predicate. The simplistic approach is to blindly explore new paths by reversing constraints and at the end create a vulnerability point reachability predicate that is a disjunction (i.e., an enumeration) of all the discovered paths leading to the vulnerability point. This is the approach that our string-enhanced white-box exploration uses in Chapter 6 to create multi-path models of the content-sniffing algorithms in Internet Explorer 7 and Safari 3.1. Such approach has two main problems. First, blindly reversing constraints produces a search space explosion, since the number of paths to explore becomes exponential in the number of constraints, and much larger than the real number of paths that exist in the program. We explain this in detail in Section 7.4. In addition, merely enumerating the discovered paths generates signatures that quickly explode in size.

To address these issues, we utilize the observation that the program execution may fork at one branch condition into different paths for one processing task, and then merge back to perform another task. For example, a task can be a validation check on the input data. Each independent validation check may generate one or multiple new paths (e.g., looking for a substring in the HTTP URL generates many paths), but if the check is passed then the program moves on to the next task, which usually merges the execution back into the original path. Thus, in our exploration, we use a *protocol-level exploration graph* to identify such potential merging points. This helps alleviate the search space explosion problem, and allows our exploration to quickly reach high coverage.

7.2.4 Architecture Overview

We have implemented our approach in a system called Elcano[1]. The architecture of Elcano is shown in Figure 7.2. It comprises of two main components: the *constraint extractor* and the *exploration module*, and two off-the-shelf assisting components: the *execution monitor*, introduced in Chapter 2, and the *parser*.

The overall exploration is an iterative process that incrementally explores new execution paths, similar to the one introduced in Section 6.3.3. In each iteration (that we also call test), an input is sent to the program under analysis, running inside the execution monitor. The execution monitor produces an execution trace that captures the complete execution of the program on the given input. The execution monitor also logs the test result, i.e., whether the vulnerability point was reached or

[1]Elcano was a Spanish explorer who completed the first circumnavigation of the world.

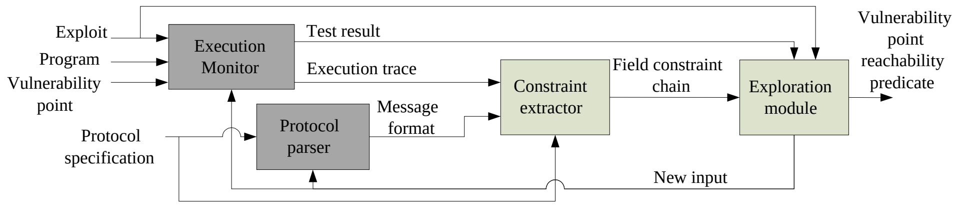


Figure 7.2: Elcano architecture overview. The darker color modules are given, while the lighter color components have been designed and implemented in this work.

not during the execution. In addition, the parser extracts the message format for the input, according to the given protocol specification. Then, given the execution trace and the message format, the constraint extractor obtains the *field constraint chain*. The field constraint chain is conceptually similar to the *path predicate* used in previous chapters, but the constraints are at the protocol-level, the parsing constraints have been removed, and each constraint is tagged with additional information. We detail the field constraint chain and its construction in Section 7.3.

The exploration module maintains the *protocol-level exploration graph*, which stores the current state of the exploration, i.e., all the execution paths that have been so far explored. Given the field constraint chain, the exploit message and the test result, the exploration module merges the new field constraint chain into the current protocol-level exploration graph. Then, the exploration module uses the protocol-level exploration graph to select a new path to be explored and generates a new input that will lead the program execution to traverse that path. Given the newly generated input, another iteration begins. We detail the exploration module in Section 7.4.

The process is started with the seed exploit message and runs iteratively until there are no more paths to explore or a user-specified time-limit is reached. At that point the exploration module outputs the VPRP. The VPRPs produced by Elcano are written using the Vine language [19] with some extensions for string operations introduced in Chapter 6.

7.3 Extracting the Field Constraint Chain

In this section we present the constraint extractor, which given an execution trace, produces a field constraint chain. The architecture of the constraint extractor is shown in Figure 7.3. First, given the execution trace the *path predicate extractor* performs symbolic execution with the input represented as a symbolic variable and extracts the *path predicate*, which is essentially the conjunction of all branch conditions dependent on the symbolic input in the execution captured in the execution trace. The path predicate extractor has been introduced in Section 5.3. It produces a byte-level path predicate which evaluates on the input bytes.

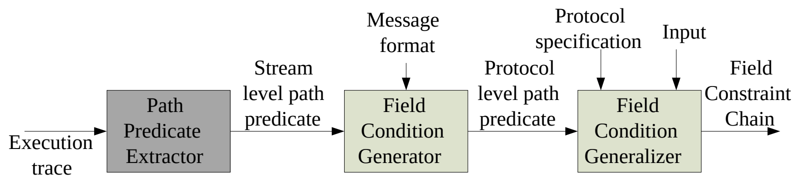

Execution trace
Path Predicate Extractor
Stream level path predicate
Message format
Field Condition Generator
Protocol level path predicate
Protocol specification
Input
Field Condition Generalizer
Field Constraint Chain

Figure 7.3: Constraint Extractor Architecture. The darker color module is given, while the lighter color components have been designed and implemented in this work.

To enable constraint-guided exploration, Elcano needs to lift the path predicate from the byte-level to the protocol-level, where the constraints are instead on field variables of the input. In addition, the constraint extractor needs to remove the parsing constraints, which dramatically reduces the exploration space and makes the exploration feasible. To accomplish this, first the *field condition generator* lifts the byte-level path-predicate to the protocol-level, and then the *field condition generalizer* generalizes it by removing the parsing constraints and outputs the *field constraint chain*. The field constraint chain differs from the protocol-level path-predicate in that the parsing constraints have been removed, each constraint is annotated with additional information, and the constraints are ordered as they appeared in the execution.

7.3.1 The Field Condition Generator

Given the byte-level path-predicate generated by the path predicate extractor and the message format of the input given by the parser, the field condition generator outputs a protocol-level path-predicate. It performs this in two steps. First, it translates each byte symbol `INPUT[x]` in the byte-level path-predicate into a field symbol `FIELD_fieldname [x - start(fieldname)]` using the message field tree produced by the parser (Section 3.2), which contains the mapping from each field to the range of bytes that it takes in the message. Second, it tries to combine symbols on consecutive bytes of the same field. For example, the byte-level path-predicate might include the following constraint: `(INPUT[6] << 8 | INPUT[7]) == 0`. If the message format states that inputs 6 and 7 belong to the same 16-bit `ID` field, then the constraint first gets translated to `(FIELD_ID[0] << 8 | FIELD_ID[1]) == 0` and then it is converted to `FIELD_ID == 0` where `FIELD_ID` is a 16-bit field symbol.

The message format provided by the parser is in the form of a message field tree (introduced in Section 3.2), where one parent field may have multiple children and the root of the tree represents the whole message. For example, the `lineBuf` variable in our running example represents the `Request-Line` field, which in turn contains 3 other fields: `Method`, `Request-URI`, and `HTTP-Version`. Thus, a constraint such as: `strstr(lineBuf,"../")` $\neq$ `0` would be translated as `strstr(FIELD_Request-Line, "../")` $\neq$ `0`. A constraint on the whole message would be applied on the root `MSG` field.

Benefits. This step lifts the byte-level path-predicate to the protocol-level, breaking the artificial constraints that the byte-level path-predicate imposes on the position of fields inside the exploit message. For example, protocols such as HTTP allow some fields in a message (i.e., the headers that follow the Request-Line/Status-Line) to be ordered differently without changing the meaning of the message. Thus, two exploit messages could have the same fields ordered differently and a byte-level vulnerability point reachability predicate generated from one of them would not flag that the other exploit also reaches the vulnerability point. In addition, if variable-length fields are present in the exploit message, changing the size of such fields changes the position of all fields that come behind it in the exploit message. Again, such trivial variation of the exploit message could defeat byte-level signatures. Thus, by expressing constraints using field symbols, protocol-level signatures naturally allow a field to move its position in the input.

7.3.2 The Field Condition Generalizer

The field condition generalizer takes as input the protocol-level path-predicate generated by the field condition generator, the protocol specification and the input that was sent to the program and outputs a field constraint chain where the parsing-related constraints have been removed.

First, the field condition generalizer assigns a symbolic variable to each byte of the input and processes the input according to the given protocol specification. This step generates symbolic constraints that capture the constraints on the input which restrict the message format of the input to be the same as the message format returned by the parser on the given input. We term these constraints the parsing constraints. Then, the field condition generalizer removes the parsing constraints from the protocol-level path-predicate by using a fast syntactic equivalence check. If the fast syntactic check fails, the field condition generalizer uses a more expensive equivalence check that uses a constraint solver.

Benefits. The parsing constraints in the protocol-level path-predicate over-constrain the variable-length fields, forcing them to have some specific size (e.g., the same as in the exploit message). Thus, removing the parsing constraints allows the vulnerability point reachability predicate to handle exploit messages where the variable-length fields have a size different than in the original exploit message. In addition, for some protocols such as HTTP, the number of parsing constraints in a single protocol-level path-predicate can range from several hundreds to a few thousands. Such a huge number of unnecessary constraints would blow up the size of the vulnerability point reachability predicate and negatively impact the exploration that we will present in Section 7.4 because each constraint would introduce a new path to be explored. Note that the parsing constraints are enforced by the parser, so we can safely remove them from the protocol-level path-predicate while still having the constraints enforced during the signature matching time.

The field constraint chain. To assist the construction of the protocol-level exploration graph (explained in Section 7.4), the constraint extractor constructs the *field constraint chain* using the generalized protocol-level path-predicate (after the parsing constraints have been removed). A field constraint chain is an enhanced version of the protocol-level path-predicate where each branch condition is annotated with its execution index [232], which is unique for each point in an execution and can be used to identify program points that correspond to each other across executions of the program. These annotated branch conditions are put in an ordered chain using the same order as they appear in the execution path.

7.4 The Exploration Module

In this section we present the exploration module. The architecture of the exploration module is illustrated in Figure 7.4. It is comprised of three components: the *explorer*, the *prioritization engine*, and the *input generator*, plus an off-the-shelf *solver*. The exploration module performs 3 main tasks in each iteration of the exploration: (1) given the field constraint chain, the explorer adds it to the current protocol-level exploration graph producing an updated graph; (2) given the updated protocol-level exploration graph, the prioritization engine decides which new path to explore next; (3) given the new path, the *input generator* generates an input that makes the program execute that path.

The new input is then used to start another iteration of the whole process as shown in Figure 7.2, that is, the new input is replayed to the program running in the execution monitor and a new field constraint chain is generated by the constraint extractor, which is passed to the explorer and so on. The prioritization engine is in charge of stopping the whole process once there are no more paths to explore or a user-specified time-limit is reached. When the exploration stops, the exploration module outputs the VPRP.

Section 7.4.1 details how the field constraint chain is added to the current protocol-level exploration graph and Section 7.4.2 describes how to generate a new input from the updated protocol-level exploration graph and Section 7.4.3 shows how the VPRP is output.

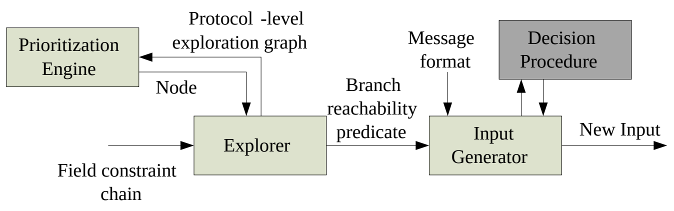


Figure 7.4: Exploration module architecture. The darker color module is given, while the lighter color components have been designed and implemented in this work.

7.4.1 Merging Execution Paths into the Protocol-Level Exploration Graph

Our exploration builds a *protocol-level exploration graph* as the exploration progresses. Using a protocol-level exploration graph makes our exploration significantly different from previous white-box exploration approaches [78, 77, 33]. The protocol-level exploration graph provides two fundamental benefits: 1) the exploration space is significantly reduced because the constraints are at the protocol level, and 2) it becomes easy to merge paths, which in turn further reduces the exploration space, and reduces the size of the vulnerability point reachability predicate.

Each node in the protocol-level exploration graph represents an input-dependent branching point (i.e., a conditional jump) in the execution. Each node contains a protocol-level predicate and some additional information about the state of the program when the branching point was reached. Each node can have two edges representing the branch taken if the node's predicate evaluated to true (T) or false (F). We call the node where the edge originates the *source node* and the node where the edge terminates the *destination node*. If a node has an *open edge* (i.e, one edge is missing), it means that the corresponding branch has not yet been explored. Figure 7.5 illustrates the exploration graph for our running example after discovering two paths to the vulnerability point. Note that nodes B, C, and D have open edges because one of their branches has not yet been explored.

Intuition. When a new field constraint chain is added to the protocol-level exploration graph, it is important to merge all constraints in the field constraint chain that are already present in the graph. Failure to merge a constraint creates a duplicate node, which in turn effectively doubles the exploration space because the subtree hanging from the replicated node would need to be explored as well. Thus, as the number of duplicated nodes increases, the exploration space increases exponentially.

The key intuition behind why merging is necessary is that it is common for new paths generated by taking a different branch at one node, to quickly merge back into the original path. This happens because programs may fork execution at one constraint for one processing task, and then merge back

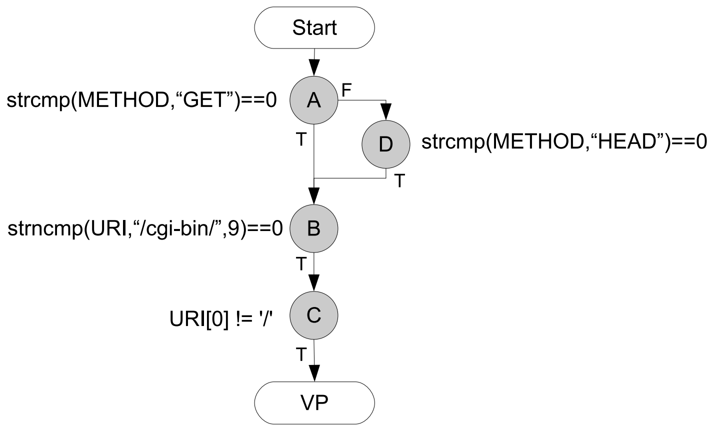


Figure 7.5: An example exploration graph for our running example. Note that nodes B, C, and D all have open edges because their false branches have not yet been explored.

to perform another task. One task could be a validation check on the input data. Each independent check may generate one or multiple new paths (e.g., looking for a substring in the URI generates many paths), but if the check is passed then the program moves on to the next task (e.g., another validation check), which usually merges the execution back into the original path. For example, when parsing a message the program needs to determine if the message is valid or not. Thus, it will perform a series of independent validity checks to verify the values of the different fields in the message. As long as checks are passed, the program still considers the message to be valid and the execution will merge back into the original path. But, if a check fails then the program will move into a very different path, for example sending an error message.

The intuition on the merging is that two nodes can be merged if they represent the same program point and they are reached with the same program state. To identify the program point, each constraint in the field constraint chain is annotated with its execution index [232], which is unique for each point in an execution and can be used to identify points that correspond to each other across executions of the program. To identify the program state we use a technique similar to the one introduced in [16] where we compute the set of all values (both concrete and symbolic) written by the program during the execution up to the point where the constraint is executed. Thus, we merge nodes that satisfy 4 conditions: same instruction address, same execution index, equivalent predicate, and same program state. Note that using the program state is important to avoid introducing errors due to implicit flows. For example, in Figure 7.5 it could happen that before the D constraint the program sets the string variable x to the string “GET”. In this case, there exists an implicit flow from the constraint A to the variable x. Then, before the C constraint, the program could check if the variable x has value “GET”, but the variable x would not be symbolic due to the implicit flow and such constraint would not be added to the graph. To avoid incorrectly merging nodes with different state, we use the write set to approximate the program state at the time the constraint is evaluated.

Merging a new path into the exploration graph. To insert a new field constraint chain into the protocol-level exploration graph, the explorer starts merging from the top until it finds a node that it cannot merge, either because it is not in the graph yet, or because the successor in the new field constraint chain is not the same one as in the graph. We call the predecessor of the node that cannot be merged (i.e., the last node merged) the *split node*. To check if a node is already in the graph, the explorer checks if the node to be inserted is equivalent (same instruction address, same execution index, equivalent predicate, and same state) to any other node already in the graph.

Once a split node has been identified the graph keeps trying to merge the rest of the nodes in the new field constraint chain until it finds a node that it can merge, which we term the *join node*. At that point, the explorer adds all the nodes in the new field constraint chain between the split node and the join node as a sequence of nodes in the graph hanging from the split node and merging at the join node. The process of looking for a split node and then for a join node is repeated until the sink of the new field constraint chain is reached. At that point, if the explorer was looking for a join node then all nodes between the last split node and the sink are added to the graph as a sequence that hangs from the last split node and ends at the sink.

For example, Figure 7.6 illustrates the graph construction for our running example. In Figure 7.6A the graph contains only the original field constraint chain generated by sending the seed exploit message to the program, which contains the three nodes introduced by lines 24, 26, and 29 in our running example (since the parsing constraints have already been removed). The sink of the original field constraint chain is the vulnerability point node (VP). Figure 7.6B shows the second field constraint chain that is added to the graph, which was obtained by creating an input that traverses the false branch of node A. When adding the field constraint chain in Figure 7.6B to the graph in Figure 7.6A, the explorer merges node A and determines that A is a split node because A's successor in the new field constraint chain is not A's successor in the graph. Then, at node B the explorer finds a join node and adds node D between the split node and the join node in the graph. Finally node C is merged and we show the updated graph in Figure 7.6C.

7.4.2 Generating a New Input

Even after removing the parsing constraints from the protocol-level path predicate and merging duplicated constraints, the number of paths to explore can still be large. Since we are only interested in paths that reach the vulnerability point, we have implemented a simple prioritization scheme that

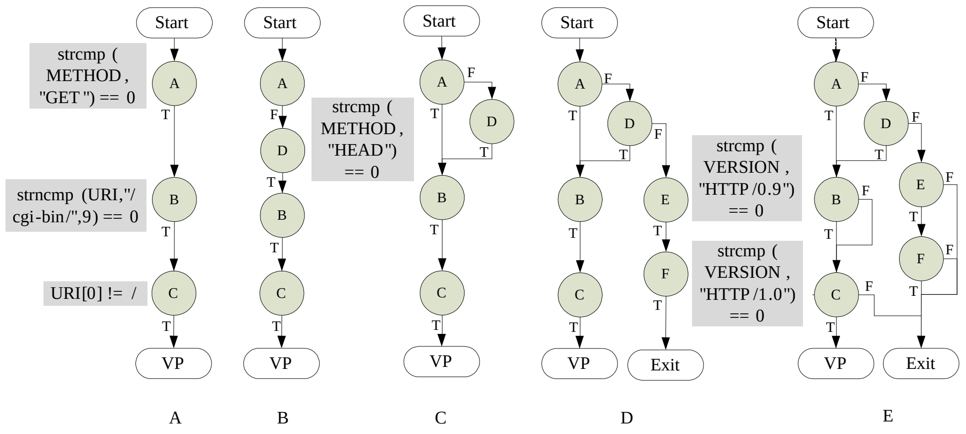

Start
strcmp (METHOD , "GET ") == 0
strncmp (URI,"/ cgi-bin/",9) == 0
URI[0] != /
strcmp (METHOD , "HEAD") == 0
strcmp (VERSION , "HTTP /0.9") == 0
strcmp (VERSION , "HTTP /1.0") == 0
A
B
C
D
E
F
T
F
VP
Exit
A
B
C
D
E

Figure 7.6: Building the protocol-level exploration graph for our running example.

favors paths that are more likely to reach it. The prioritization engine uses a simple weight scheme, where there are three weights 0, 1, and 2. Each weight has its own node queue and the prioritization engine always picks the first node from the highest weight non-empty queue. The explorer assigns the weights to the nodes when adding them to the graph. Nodes that represent loop exit conditions get a zero weight (i.e., lowest priority). Nodes in a field constraint chain that has the vulnerability point as sink get a weight of 2 (i.e., highest priority). All other nodes get a weight of 1. We favor nodes that are in a path to the vulnerability point because if a new path does not quickly lead back to the vulnerability point, then the message probably failed the current check or went on to a different task and thus it is less likely to reach the vulnerability point later. We disfavor loop exit conditions to delay unrolling the same loop multiple times. Such heuristic helps achieve high coverage quickly.

We define a *node reachability predicate* to be the predicate that summarizes how to reach a specific node in the protocol-level exploration graph from the `Start` node, which includes all paths in the graph from the `Start` to that node. Similarly, we define a *branch reachability predicate* to be the predicate that summarizes how to traverse a specific branch of a node. A branch reachability predicate is the conjunction of a node reachability predicate with the node's predicate (to traverse the true branch), or the negation of the node's predicate (to traverse the false branch). To compute a new input that traverses the specific branch selected by the prioritization engine, the explorer first computes the branch reachability predicate. Then, the input generator generates an input that satisfies the branch reachability predicate.

To compute the branch reachability predicate, the explorer first computes the node reachability predicate. The node reachability predicate is essentially the weakest pre-condition (WP) [61] of the source node of the open edge over the protocol-level exploration graph—by definition, the WP captures all paths in the protocol-level exploration graph that reach the node. Then, the explorer computes the conjunction of the weakest pre-condition with the node's predicate or with the negated predicate depending on the selected branch. Such conjunction is the branch reachability predicate, which is passed to the input generator.

For example, in Figure 7.6C if the prioritization engine selects the false branch of node D to be explored next, then the branch reachability predicate produced by the explorer would be: $\overline{A} \wedge \overline{D}$. Similarly, in Figure 7.6D if the prioritization engine selects the false branch of node B to be explored next, then the branch reachability predicate produced by the explorer would be: $(A \vee (\overline{A} \wedge D)) \wedge \overline{B}$.

The input generator generates a new input that satisfies the branch reachability predicate using a 3-step process. First, it uses a constraint solver to generate field values that satisfy the branch reachability predicate. If the constraint solver returns that no input can reach that branch, then the branch is connected to the `Unreachable` node. Second, it extracts the values for the remaining fields (unconstrained by the solver) from the seed exploit message. Third, it checks the message format provided by the parser to identify any fields that need to be updated given the dependencies on the modified values (such as length or checksum fields). Note that here we assume the message field tree output by the parser provides such dependencies. Otherwise, we need to use the techniques in Chapter 3 to identify the length fields and the techniques that we will introduce in Chapter 8 to identify the checksum fields. Using all the collected field values it generates a new input that starts a new iteration.

7.4.3 Extracting the Vulnerability Point Reachability Predicate

Once the exploration ends, the protocol-level exploration graph contains all the discovered paths leading to the vulnerability point. To extract the VPRP from the graph the explorer computes the node reachability predicate for the VP node. For our running example, represented in Figure 7.6E the VPRP is: $(A \vee (\overline{A} \wedge D)) \wedge C$. Note that, a mere disjunction of all paths to the VP, would generate the following VPRP: $(A \wedge B \wedge C) \vee (\overline{A} \wedge D \wedge B \wedge C) \vee (A \wedge \overline{B} \wedge C) \vee (\overline{A} \wedge D \wedge \overline{B} \wedge C)$. Thus, Elcano's VPRP is more compact using 4 conditions instead of 14.

7.5 Evaluation

In this section, we present the results of our evaluation. We evaluate Elcano using 6 vulnerabilities, summarized in Table 7.1. The table shows the program, the CVE identifier for the vulnerability [56], the protocol used by the vulnerable program, the protocol type (i.e., binary or text), the guest operating system used to run the vulnerable program, and the type of vulnerability. We select the vulnerabilities to cover file formats as well as network protocols, multiple operating systems, multiple vulnerability types, and both open-source and closed programs, where no source code is available.

Program	CVE	Protocol	Type	Guest OS	Vuln. Type
gdi32.dll (v3159)	CVE-2008-1087	EMF file	Binary	Windows XP	Buffer overflow
gdi32.dll (v3099)	CVE-2007-3034	WMF file	Binary	Windows XP	Integer overflow
Windows DCOM RPC	CVE-2003-0352	RPC	Binary	Windows XP	Buffer overflow
GHttpd	CVE-2002-1904	HTTP	Text	Red Hat 7.3	Buffer overflow
AtpHttpd	CVE-2002-1816	HTTP	Text	Red Hat 7.3	Buffer overflow
Microsoft SQL Server	CVE-2002-0649	Proprietary	Binary	Windows 2000	Buffer overflow

Table 7.1: Vulnerable programs used in the evaluation.

Program	Original	Non-parsing constraints
Gdi-emf	860	65
Gdi-wmf	4	4
DCOM RPC	535	521
GHttpd	2498	5
AtpHttpd	6034	10
SQL Server	2447	7

Table 7.2: Constraint extractor results for the first test, including the number of constraints in the protocol-level path-predicate and the number of remaining constraints after parsing constraints have been removed.

Program	All branches explored	VPRP
Gdi-emf	no	72
Gdi-wmf	yes	5
DCOM RPC	no	1651
GHttpd	yes	3
AtpHttpd	yes	10
SQL Server	yes	3

Table 7.2: Exploration results, including whether all open edges in the protocol-level exploration graph were explored and the number of constraints remaining in the vulnerability point reachability predicate.

In addition, the older vulnerabilities (i.e., last four) are also selected because they have been analyzed in previous work, and this allows us to compare our system's results to previous ones. Next, we present the constraint extractor results (Section 7.5.1), the exploration results (Section 7.5.2), and the produced signatures (Section 7.5.3).

7.5.1 Removing the Parsing Constraints

In this section we evaluate the effectiveness of the constraint extractor, in particular of the field condition generalizer, at removing the parsing constraints from the protocol-level path-predicate. For simplicity, we only show the results for the protocol-level path-predicate produced by the field condition generator from the execution trace generated by the seed exploit. Note that, during exploration this process is repeated once per newly generated input. Table 7.2 summarizes the results. The *Original* column represents the number of input-dependent constraints in the protocol-level path-predicate and is used as the base for comparison. The *Non-parsing constraints* column shows the number of remaining constraints after removing the parsing constraints.

The removal of the parsing constraints is very successful in all experiments except the DCOM-RPC. Overall, in the four vulnerable programs that include variable-length strings (i.e., excluding

Gdi-wmf and DCOM-RPC), the parsing constraints account for 92.4% to 99.8% of all constraints. For formats that include arrays, such as DCOM RPC, the number of parsing constraints is much smaller but it is important to remove such constraints; otherwise they constrain the array to have the same number of elements as in the exploit message. By removing the parsing constraints, each field constraint chain represents many program execution paths produced by modifying the format of the exploit message (e.g., extending variable-length fields or reordering fields). This dramatically decreases the exploration space making the exploration feasible.

7.5.2 Exploration Results

Table 7.2 shows the results for the exploration phase. We set a user-defined time-limit of 6 hours for the exploration. If the exploration has not completed by that time Elcano outputs the intermediate VPRP and stores the current state of the exploration. This state can later be loaded to continue the exploration at the same point where it was interrupted. The first column indicates whether the exploration completes before the specified time-limit. The second column presents the number of constraints in the intermediate VPRP that is output by the exploration module once there are no more paths to be explored or the time-limit is reached.

The results show that in 4 out of 6 experiments Elcano explored all possible paths, thus generating a complete VPRP (i.e., a VPRP that covers all paths to the vulnerability point). For the DCOM RPC and Gdi-emf experiments, the 6 hour time-limit was reached, thus the VPRPs are not complete. They also show that the number of constraints in the VPRP is in most cases small. The small number of constraints in the VPRP and the fact that in many cases those constraints are small themselves, makes the signatures easy for humans to analyze, as opposed to previous white-box approaches where the large number of constraints in the signature made it hard to gain insight on the quality of the signature. We do that by labeling the nodes in the graph with the full protocol-level constraints.

Performance. Table 7.3 summarizes the performance measurements for Elcano. All measurements were taken on a desktop computer with a 2.4GHz Intel Core2 Duo CPU and 4 GB of memory. The first column presents the VPRP generation time in seconds. For the Gdi-emf and DCOM RPC examples, the 6 hour (21,600 sec.) time-limit is reached. For the rest, the generation time ranges from under one minute for the GHttpd vulnerability up to 23 minutes for the Microsoft SQL vulnerability. Most of the time (between 60% and 80% depending on the example) is spent by the constraint extractor. The remaining columns show the number of tests in the exploration, the average time per test in seconds, and the average size in Megabytes of the execution trace.

Program	Gener. time	# tests	Avg. test time	Trace size
Gdi-emf	21600	502	43.0	28.8
Gdi-wmf	98	6	16.3	3.0
DCOM RPC	21600	235	92.0	3.5
GHttpd	55	6	9.1	3.0
AtpHttpd	282	12	23.5	8.6
SQL Server	1384	11	125.8	27.5

Table 7.3: Performance evaluation. The generation time and the average test time are given in seconds, and the trace size is given in Megabytes.

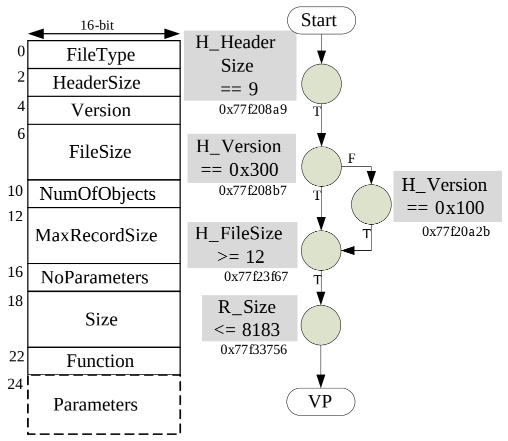


Table 7.3: On the left, the format of the Gdi-wmf exploit file. On the right the vulnerability point reachability predicate.

Compared to Bouncer, where the authors also analyze the SQL Server and GHttpd vulnerabilities, the signatures produced by Elcano have higher coverage (i.e., less false negatives) and are smaller. For example, Bouncer spends 4.7 hours to generate a signature for the SQL Server vulnerability, and the generated signature only covers a fraction of all the paths to the vulnerability point. In contrast, Elcano spends only 23 minutes, and the generated signature covers all input-dependent branches to the vulnerability point. Similarly, for the GHttpd vulnerability the authors stop the signature generation after 24 hours, and again the signature only covers a fraction of all input-dependent branches to the vulnerability point, while Elcano generates a complete signature that covers all input-dependent branches to the vulnerability point in under one minute (according to the ShieldGen authors, who studied this vulnerability and had access to the source code).

7.5.3 Signatures

For the two vulnerabilities in open-source programs (GHttpd and AtpHttpd), we extract the perfect signatures for the vulnerability through manual analysis of the source code. The results show that Elcano's VPRPs exactly match or are very close to the perfect ones that we manually extract. For AtpHttpd the signature misses one path to the vulnerability point where the server uses the `stat` function to check whether a prefix of the URI field is a directory on disk. To add this constraint to the final VPRP we could use a function summary for the stat function similar to the ones we used in Chapter 6 for the string functions. In that case the system that tests the signature would have to be identically configured to the vulnerable system. Another approach would be to ignore such constraints that test local configuration, assuming they are always satisfied. Such approach makes configuration easier but may introduce false positives. For the other signatures we compare with previous results when available. Next, we detail the SQL Server and Gdi-wmf signatures.

SQL server. The parser returns that there are two fields in the seed exploit message: the Command (CMD) and the Database name (DB). The protocol-level path predicate for the path corresponding to the seed contains 7 constraints, which after simplification can be reduced to three. The exploration covers the open edges of those 3 nodes and finds that none of the newly generated inputs reaches the vulnerability point. Thus, no new paths are added to the exploration graph and the VPRP is:
`(FIELD_CMD==4)` $\wedge$ `(strcmp(FIELD_DB,"")`$\neq$`0)` $\wedge$ `(strcasecmp(FIELD_DB,"MSSQLServer")`$\neq$`0)`.

In addition, the vulnerability condition for this vulnerability states that the length of the DB field needs to be larger than 64 bytes [188], which makes the last two constraints in the VPRP redundant. Thus, the final protocol-level signature would be: `(FIELD_CMD == 4)` $\wedge$ `length(FIELD_DB) > 64`. According to the ShieldGen authors, who had access to the source code, this would be a perfect signature.

Gdi-wmf. Figure 7.3 shows on the left the field structure for the seed exploit file and on the right the VPRP. The original protocol-level path-predicate contained the 4 aligned nodes on the left of the graph, while the exploration discovers one new path leading to the vulnerability point that introduces the node on the right. The graph shows that the program checks whether the `Version` field is 0x300 (Windows 3.0) or 0x100 (Windows 1.0). Such constraint is unlikely to be detected by probing approaches, since they usually sample only a few values. In fact, in ShieldGen they analyze a different vulnerability in the same library but run across the same constraint. The authors acknowledge that they miss the second constraint of the disjunction. Thus, an attacker could easily avoid detection by changing the value of the Version field. The vulnerability condition is `((2` $\cdot$ `FIELD_RSIZE) >> 2) < 0` [188]. Thus, the final signature is: `(FIELD_HSIZE == 9)` $\wedge$ `((FIELD_VERSION == 0x300)` $\vee$ `(FIELD_VERSION == 0x100))` $\wedge$ `(FIELD_FILESIZE` $\geq$ `12)` $\wedge$ `(FIELD_RSIZE` $\leq$ `8183)` $\wedge$ `(((2` $\cdot$ `FIELD_RSIZE) >> 2) < 0)`.

7.6 Related Work

In this section, we present related work on automatic signature generation. We refer the reader to the related work sections in earlier chapters for a description of related research on symbolic execution (Section 5.7), automatic test case generation (Section 6.6), compositional approaches to white-box exploration (Section 6.6), and automatic protocol reverse-engineering (Section 3.8).

Exploit-based signatures. Early works on automatic signature generation propose generating exploit-based signatures using machine learning techniques that identify patterns in the observed exploits [112, 105, 197, 159, 237, 118, 220]. However, exploit-based signatures are not guaranteed to correctly describe the vulnerability, require a large set of attack samples, can introduce false positives, and can be easily defeated with exploit variants. There is also work on using host information to increase signature accuracy and generation speed [127, 120, 119, 233]. However, these approaches only use limited host information, and still cannot model the vulnerability accurately. Venkataraman et al. show the limits of using machine learning techniques for signature generation in an adversarial environment [213].

Vulnerability-based signatures. A more recent line of work generates symbolic-execution based vulnerability signatures directly from the vulnerable program binary [47,21,46,23]. The vulnerability-based signatures are guaranteed to have no false positives by design. Vigilante uses dynamic symbolic execution to produce signatures that are boolean predicates that cover a single path to the vulnerability point [47]. Compared with Vigilante, Elcano also produces boolean predicates as signatures but our signatures have higher coverage because they cover multiple execution paths to the vulnerability point. Brumley et al. analyzes three types of signatures: Turing machines, boolean signatures, and regular expressions, and demonstrates that a perfect vulnerability-based signature must be in at least the same language class as the vulnerability language [21]. More recently, Brumley et al. propose a static analysis technique based on weakest-precondition to generate boolean signatures that cover multiple paths to the vulnerability point [23]. Our approach uses dynamic analysis instead of static analysis and can achieve more precision in the presence of indirection, pointers and loops.

Bouncer extends previous dynamic symbolic execution approaches that produce boolean predicates on inputs as signatures [46]. Even though Bouncer makes improvements in increasing the coverage of the generated signatures, it still suffers from several limitations. First, it generates byte-level signatures instead of protocol-level signatures. As a result, it is difficult for Bouncer to handle evasion attacks using variable-length fields and field reordering. Second, Bouncer's exploration is inefficient and largely heuristic-based. As mentioned in their paper, the authors tried to use white-box exploration to explore the program execution space to identify different paths reaching the vulnerability point, but couldn't make the approach scale to real-world programs and thus had to resort to heuristics such as duplicating or removing parts of the input message or sampling certain field values to try to discover new paths leading to the vulnerability point. To make white-box exploration feasible and effective, Elcano incorporates two other key characteristics. It uses protocol information to lift the byte-level path predicates to the protocol level, and it merges protocol-level path predicates to avoid a search space explosion, since duplicated constraints can make the number of paths to explore exponential in the number of constraints, and much larger than the real number of paths that exist in the program.

Protocol-level signatures. Shield proposes a framework for protocol-level vulnerability-based signatures, which are created manually by modeling network protocols [218]. ShieldGen proposes an approach to automatically generate such protocol-level signatures [55]. ShieldGen takes a probing-based approach using protocol format information and does not use information about the program execution. Using the given protocol format, it generates different well-formed variants of the original exploit using various heuristics and then checks whether any of the variants still exploits the vulnerability. ShieldGen's heuristics first assume that fields can be probed independently, and then for fixed-length fields it samples just a few values of each field, checking whether the vulnerability point is reached or not for those values. Probing each field independently means that constraints involving multiple fields cannot be found. Probing only a few sample values for each field is likely to miss constraints that are satisfied by only a small fraction of the field values. Moreover, the type of constraints in ShieldGen are limited to either "a field has the same value as in the seed exploit message" or "a field can have any value". In contrast, our approach supports more granular constraints such as "a field can have only values larger than five". Compared to ShieldGen our approach does not uses heuristics but uses white-box exploration to increase the coverage of the signature, addressing the above limitations.

7.7 Conclusion

In this chapter we have proposed protocol-level constraint-guided exploration, an approach for automatically generating vulnerability point reachability predicates, with application to signature generation, exploit generation and patch verification. Our approach produces high coverage, yet compact, vulnerability point reachability predicates that capture many paths to the vulnerability point and thus are more difficult to evade using exploit variants.

Compared to previous white-box exploration approaches, protocol-level constraint-guided exploration provides two key characteristics to make the exploration scale to programs that parse complex, highly-structured inputs such as protocols and file formats. First, our approach lifts the byte-level path predicates in previous approaches to the protocol-level, so that they evaluate on symbolic variables that represent the fields in the input. This lifting provides two main benefits. It removes the parsing constraints that produce a huge amount of paths that otherwise would need to be explored, greatly reducing the exploration space. In addition, it produces predicates that match exploit variants that modify the field structure of an exploit, e.g., by increasing the size of the variable-length fields or reordering the fields in the exploit message. Second, our approach merges protocol-level path predicates into a protocol-level exploration graph. Such merging removes redundant constraints that otherwise would have to be explored. Without merging, the search space could exponentially increase and become much larger than the real number of paths in the program.

We have applied our approach to generate signatures for 6 vulnerabilities on real-world programs. The generated vulnerability point reachability predicates achieve perfect or close-to-perfect coverage. Using a 6 hour time limit for the exploration, our exploration generates a complete vulnerability point reachability predicate, covering all possible paths to the vulnerability point for 4 out of 6 vulnerabilities. In addition, the number of constraints in the resulting vulnerability point reachability predicate is in most cases small, which makes them more suitable for human analysts.

Chapter 8

Stitched Vulnerability Discovery

8.1 Introduction

Vulnerability discovery in *benign* programs has long been an important task in software security: identifying software bugs that may be remotely exploitable and creating program inputs to demonstrate their existence. However, little research has addressed vulnerabilities in *malware*. Do malicious programs have vulnerabilities? Do different binaries of the same malware family share vulnerabilities? How do we automatically discover vulnerabilities in malware? What are the implications of vulnerability discovery in malware to malware defense, law enforcement and cyberwarfare? In this chapter we take the first step toward addressing these questions. In particular, we propose new symbolic reasoning techniques for automatic input generation in the presence of complex functions such as decryption and decompression, and demonstrate the effectiveness of our techniques by finding bugs in real-world malware. Our study also shows that vulnerabilities can persist for years across malware revisions. Vulnerabilities in botnet clients are valuable in many applications: besides allowing a third party to terminate or take control of a bot in the wild, they also reveal genealogical relationships between malware samples. We hope our work will spur discussions on the implications and applications of malware vulnerability discovery.

Dynamic symbolic execution techniques [106] have recently been used for a variety of input generation applications such as vulnerability discovery [78, 33, 79], automatic exploit generation [22, 104], and finding deviations between implementations [20]. By computing symbolic constraints on the input to make the program execution follow a particular path, and then solving those constraints, dynamic symbolic execution allows a system to automatically generate an input to execute a new path. Repeating this process gives an automatic exploration of the program execution space. However, traditional dynamic symbolic execution is ineffective in the presence of certain common computation tasks, including the decryption and decompression of data, and the computation of checksums and hash functions; we call these *encoding functions*. Encoding functions result in symbolic formulas that can be difficult to solve, which is not surprising, given that cryptographic hash functions are designed to be impractical to invert [161]. Encoding functions are used widely in malware as well as benign applications. In our experiments, the traditional dynamic symbolic execution approach fails to explore the execution space of the malware samples effectively.

To address the challenges posed by the presence of encoding functions, we propose a new approach, *stitched* dynamic symbolic execution [27]. This approach first automatically identifies potential encoding functions and their inverses (if applicable). Then it decomposes the symbolic constraints from the execution, separating the constraints generated by each encoding function from the constraints in the rest of the execution. The solver does *not* attempt to solve the (hard) constraints introduced by the encoding functions. Instead it focuses on solving the (easier) constraints from the remainder of the execution. Finally, the approach re-stitches the solver's output using the encoding functions or their inverses, creating a program input that can be fed back to the original program.

For instance, our approach can automatically identify that a particular function in an execution is performing a computation such as decrypting the input. Rather than using symbolic execution inside the decryption function, it applies symbolic execution on the outputs of the decryption function, producing constraints for the execution after the decryption. Solving those constraints generates an unencrypted message. Then, it executes the inverse (encryption) function on the unencrypted message, generating an encrypted message that can be fed back as the input to the original program.

More generally, we decompose two kinds of computations: serial computations that transform data into a new form that replaces the old data (e.g., decompression and decryption), and side computations that generate constraints that can be satisfied by choosing values for another part of the input (e.g., checksums). For clarity, we explain our techniques in the context of dynamic symbolic execution, but they equally apply to concrete fuzz testing [63, 145] and taint-directed fuzzing [74].

Our stitched dynamic symbolic execution approach applies to programs that use complex encoding functions, regardless if benign or malicious. In this chapter, we use it to enable the first automated study of bugs in malware. The closest previous work we know of has focused on finding bugs on the remote administration tools that attackers use to control the malware, as opposed to the malware programs themselves, running on the compromised hosts [179, 64]. Using stitched dynamic symbolic execution we find 6 new, remotely trigger-able bugs in 4 prevalent malware families that include botnet clients (Cutwail, Gheg, and MegaD) and trojans (Zbot). A remote network attacker can use these bugs to terminate or subvert the malware. At least one of the bugs can be exploited, e.g., by an attacker different than the botmaster, to take over the compromised host. To confirm the value of our approach, we show that traditional white-box exploration would be unable to find most of the bugs we report without the new techniques we introduce.

Malware vulnerabilities have a great potential for different applications such as malware removal or cyberwarfare. Some malware programs such as botnet clients are deployed at a scale that rivals popular benign applications. For instance, the recently-disabled Mariposa botnet was sending messages from more than 12 million unique IP addresses at the point it was taken down, and stole data from more than 800,000 users [45]. Our goal in this research is to demonstrate that finding vulnerabilities in widely-deployed malware such as botnet clients is technically feasible. However, the implications of the usage of malware vulnerabilities require more investigation. For example, some of the potential applications of malware vulnerabilities raise ethical and legal concerns that need to be addressed by the community. Thus, another goal of this research is to raise awareness and spur discussion in the community about the positives and negatives of the different uses of malware vulnerabilities.

The remainder of this chapter is organized as follows: Section 8.2 defines the problem we address, Section 8.3 describes our approach in detail, Section 8.4 gives additional practical details of our implementation, Section 8.5 describes our case studies finding bugs in malware, Section 8.6 discusses the implications of our results, Section 8.7 surveys related work, and finally, Section 8.8 concludes.

8.2 Problem Definition & Approach Overview

In this section, we describe the problem we address and give an overview of our approach.

8.2.1 Problem Definition

Often there are parts of a program that are not amenable to dynamic symbolic execution. A class of common culprits, which we call *encoding functions*, includes many instances of decryption, decompression, and checksums. For instance, consider the code in Figure 8.1, which is an example modeled after a botnet client. A C&C message for this botnet comprises 4 bytes with the message length, followed by 20 bytes corresponding to a SHA-1 hash [161], followed by an encrypted payload. The bot casts the received message into a message structure, decrypts the payload using AES [57], verifies the integrity of the (decrypted) message body using the SHA-1 hash, and then takes a malicious action such as sending spam based on a command in the message body. Dynamic symbolic execution attempts to create a new valid input by solving a modified path predicate. Suppose we run the program on a message that causes the bot to participate in a DDoS attack: at a high level, the path condition takes the form

$$m' = \text{Dec}(m) \wedge h_1 = \text{SHA1}(m') \wedge m'[0] = 101 \tag{8.1}$$

```
1  struct msg {
2      long msg_len;
3      unsigned char hash[20];
4      unsigned char message[];
5  };
6  void process(unsigned char* network_data) {
7      unsigned char *p;
8      struct msg *m = (struct msg *) network_data;
9      aes_cbc_decrypt(m->message, m->msg_len, key);
10     p = compute_sha1(m->message, m->msg_len);
11     if (memcmp(p, m->hash, 20))
12         exit(1);
13     else {
14         int cmd = m->message[0];
15         if (cmd == 101)
16             ddos_attack(m);
17         else if (cmd == 142)
18             send_spam(m);
19           /* ... */
20     }
21 }
```

Figure 8.1: A simplified example of a program that uses layered input processing, including decryption (line 9) and a secure hash function for integrity verification (lines 10-12).

where m and h_1 represent two relevant parts of the program input treated as symbolic: m is the encrypted payload `m->message`, and h_1 is the message checksum `m->hash`. Dec represents the AES decryption, while SHA1 is the SHA-1 hash function. To see whether it can create a message to cause a different action, dynamic symbolic execution will attempt to solve the modified path condition

$$m' = \text{Dec}(m) \land h_1 = \text{SHA1}(m') \land m'[0] \neq 101 \tag{8.2}$$

which differs from the original in inverting the last condition.

However, solvers tend to have a very hard time with conditions such as this one. As seen by the solver, the Dec and SHA1 functions are expanded into a complex combination of constraints that mix together the influence of many input values and are hard to reason about [59]. The solver cannot easily recognize the high-level structure of the computation, such as that the internals of the Dec and SHA1 functions are independent of the parsing condition $m'[0] \neq 101$. Such encoding functions are also just as serious an obstacle for related techniques like concrete and taint-directed fuzzing. Thus, the problem we address is how to perform input generation, using dynamic symbolic execution, for programs that use encoding functions.

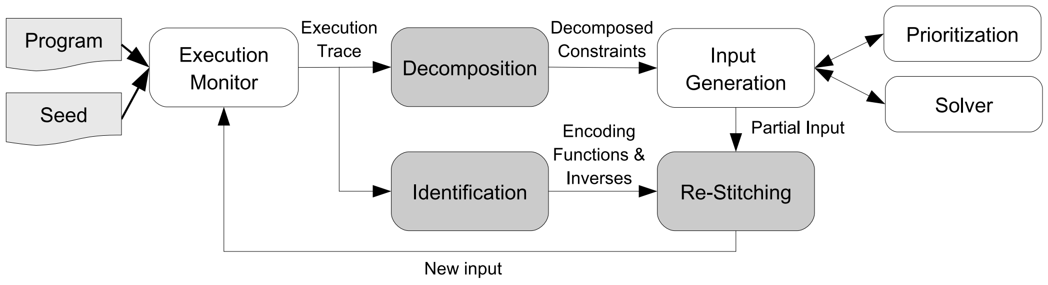


Figure 8.2: Architectural overview of our approach. The gray modules comprise stitched dynamic symbolic execution, while the white modules are the same as in traditional dynamic symbolic execution.

8.2.2 Approach Overview

We propose an approach of *stitched* dynamic symbolic execution to perform input generation in the presence of encoding functions. The insight behind our approach is that it is possible to avoid the problems caused by encoding functions, by identifying the (hard) constraints they introduce and bypassing them to concentrate on the (easier) constraints introduced by the rest of the program. Then, we can used concrete execution of the encoding functions and their inverses to re-stitch an input. For instance, in the path predicate of formula 8.2, the first and second constraints come from encoding functions. Our approach can verify that those constraints are independent from each other and from the message parsing process (exemplified by the constraint $m'[0] \neq 101$). Thus, those two constraints can be decomposed, and the solver can concentrate on the remaining constraints. Solving the remaining constraints gives a partial input in the form of a value for m', and our tool can then re-stitch this into a complete program input by concretely executing the encoding functions or their inverses, specifically h_1 as SHA1(m') and m as Dec$^{-1}(m')$.

Stitched dynamic symbolic execution. Figure 8.2 presents the architectural overview of our approach. Stitched dynamic symbolic execution comprises three modules: identification, decomposition, and re-stitching. The identification module finds the encoding functions present in a program execution (such as decryption and checksums) and looks for any required inverses both in the execution as well as in external sources that we describe in the next paragraph. The identification module can be run in each iteration of the exploration or only in a subset of iterations. We describe the identification module in Section 8.3.3. The decomposition and re-stitching modules run in every iteration of the exploration. On each iteration, the decomposition module separates the constraints in the path predicate into two groups: those introduced by the encoding functions and those introduced by the rest of the execution. We describe the decomposition module in Section 8.3.1. The decomposition module passes the constraints from the rest of the execution to the input generation module, which is similar to the one in Chapters 6 and 7. The input generation module produces a partial input that explores a new path inside the program's execution. The re-stitching module takes as input the partial input, as well as the encoding functions and inverses from identification and re-stitches the partial input into a complete program input that can be used to start another iteration of the exploration. If as in Figure 8.1 there are multiple layers of encoding functions, each layer is decomposed in turn, and then the layers are re-stitched in reverse order. We describe the re-stitching module in Section 8.3.2.

Identifying encoding functions and their inverses. For identifying encoding functions, the identification modules performs a trace-based dependency analysis that is a general kind of dynamic tainting. This analysis detects functions that highly *mix* their input, i.e., an output byte depends on many input bytes. The intuition is that high mixing is what makes constraints difficult to solve. For example, a block cipher in CBC mode highly mixes its input and the constraints it introduces during decryption are hard to solve, but a stream cipher does not mix its input and thus the constraints it introduces can be easily solved. Although this may be counterintuitive, note that a stream cipher such as RC4 decrypts the input by performing, for each input byte, an xor operation with a pseudo-random key stream. For the solver, the pseudo-random key stream is concrete and thus it only needs to revert a simple xor operation between the input and a constant. In other words, keys that are not derived from the input are simply constants for the solver. Thus, our identification technique targets encoding functions that highly mix their inputs.

In addition to the encoding functions, our approach may also require their inverses (e.g., for decryption and decompression functions). The intuition behind finding inverses is that encoding functions and their inverses are often used in concert, so their implementations can often be found in the same binaries or in widely-available libraries (e.g., OpenSSL [165] or zlib [240]). In this chapter, we propose a technique that given a function, identifies whether its inverse is present in a set of other functions. We detail the identification of encoding functions and their inverses in Section 8.3.3. We further discuss the availability of inverse functions in Section 8.6.2.

8.3 Stitched Dynamic Symbolic Execution

In this section we describe the three modules that comprise stitched dynamic symbolic execution: decomposition (Section 8.3.1), re-stitching (Section 8.3.2), and identification (Section 8.3.3).

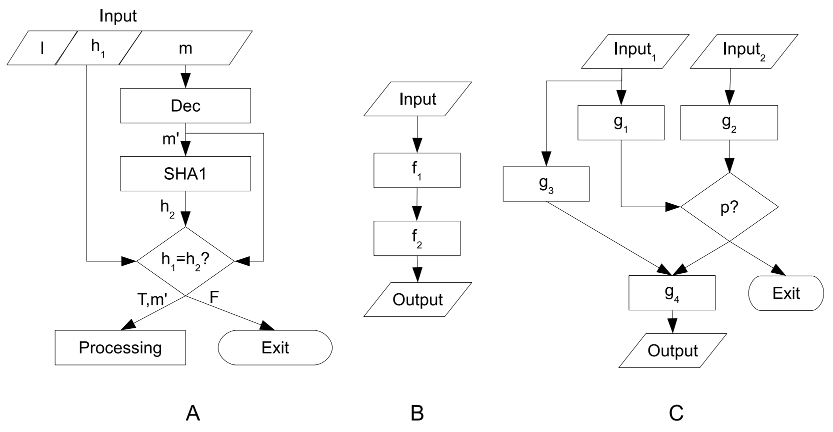


Figure 8.3: On the left a graphical representation of the decomposition of our running example in Figure 8.1. The other two figures represent the two types of decomposition that our approach supports: serial decomposition (B) and side-condition decomposition (C).

8.3.1 Decomposition

Decomposition is the process of breaking down a program into smaller components. Figure 8.3A shows a decomposition diagram for our running example in Figure 8.1, where components are represented by boxes and diamonds, and arrows represent the inputs and outputs of those components, that is, the dependencies between components. Analyzing part of a program separately corresponds to cutting the dependencies that link its inputs to the rest of the execution.

For a formula generated by symbolic execution, we can make part of the formula independent by renaming the variables it refers to. Following this approach, it is not necessary to extract a component as if it were a separate program. Our tool can simply perform dynamic symbolic execution on the entire program, and achieve a separation between components by using different variable names in some of the extracted constraints. We propose two generic forms of decomposition: serial decomposition (Figure 8.3B) and side-condition decomposition (Figure 8.3C). For each form of decomposition, we explain which parts of the program are identified for decomposition, and describe what local and global dependency conditions are necessary for the decomposition to be correct.

Serial decomposition. The first style of decomposition our approach performs is between successive components, in which the first layer is a transformation producing input to the second layer. More precisely, it involves what we call a *surjective transformation*. There are two conditions that define a surjective transformation. First, once a value has been transformed, the pre-transformed form of the input is never used again. Second, the transformation must be an onto function: every element in its codomain can be produced with some input. For example, if a function $y = x^2$ returns a signed 32-bit integer, the codomain contains 2^{32} elements. In that case, the image is a subset of the codomain that does not include for example the value -1, as it is not a possible output of the function. In Figure 8.3B, f_1 is the component that must implement a surjective transformation. Some examples of surjective transformations include decompression and decryption. The key insight of the decomposition is that we can analyze the part of the program downstream from the transformation independently, and then simply invert the transformation to re-stitch inputs. For instance, in the example of Figure 8.1, the decryption operation is a surjective transformation that induces the constraint $m' = \text{Dec}(m)$. To analyze the rest of the program without this encoding function, we can just rename the other uses of m' to a new variable (say m'') that is otherwise unconstrained, and analyze the program as if m'' were the input. Bypassing the decryption in this way gives

$$h_1 = \text{SHA1}(m'') \wedge m''[0] = 101 \tag{8.3}$$

as the remaining path condition.

Side-condition decomposition. The second style of decomposition our approach performs separates two components that operate on independent parts of the same input. Intuitively, a *free side-condition* is a constraint on part of a program's input that can effectively be ignored during analysis of the rest of a program, because it can always be satisfied by choosing values for another part of the input. We can be free to change this other part of the input if it does not participate in any constraints other than those from the side-condition. More precisely, a program exhibiting a free side-condition takes the form shown in Figure 8.3C. The side-condition is the constraint that the predicate p must hold between the outputs of g_1 and g_2. The side-condition is free because whatever value $Input_1$ takes, p can be satisfied by making an appropriate choice $Input_2$. An example of a free side-condition is that the checksum computed over a program's input (g_1 applied on $Input_1$) must equal (p) the checksum field parsed from a message header (g_2 applied on $Input_2$). Note that g_2 is often the identity function but could also be another transformation.

To perform decomposition given a free side-condition, we simply replace the side-condition with a value that is always true. For instance the SHA-1 hash of Figure 8.1 participates in a free side-condition $h_1 = \text{SHA1}(m'')$ (assuming we have already decomposed the decryption function as mentioned above). But h_1 does not appear anywhere else among the constraints, so we can analyze the rest of the program as if this condition were just the literal *true*. This gives the path condition:

$$\text{true} \wedge m''[0] = 101 \tag{8.4}$$

Multiple encoding layers. If a program has more than one encoding function, we can repeat our approach to decompose the constraints from each encoding function in turn, creating a multi-layered decomposition. The decomposition operates from the outside in, in the order the encoding functions are applied to the input, intuitively like peeling the layers of an onion. As shown, in the example of Figure 8.1, our tool decomposes first the decryption function and then the hash-checking function, finally leaving only the botnet client's command parsing and malicious behavior for exploration.

8.3.2 Re-stitching

After decomposing the constraints, our tool solves the constraints corresponding to the remainder of the program (excluding the encoding function(s)), as in non-stitched dynamic symbolic execution, to give a partial input. The re-stitching step builds a complete program input from this partial input by concretely execution encoding functions and their inverses. If the decomposition is correct, such a complete input is guaranteed to exist, but we construct it explicitly so that the exploration process can re-execute the program from the beginning. Once we have found a bug, a complete input confirms (independent of any assumptions about the analysis technique) that the bug is real, allows easy testing on other related samples, and is the first step in creating a working exploit.

For serial decomposition, we are given an input to f_2, and the goal is to find a corresponding input to f_1 that produces that value. This requires access to an inverse function for f_1; we discuss finding one in Section 8.3.3. (If f_1 is many-to-one, any inverse will suffice.) For instance, in the example of Figure 8.1, the partial input is a decrypted message, and the full input is the corresponding AES-encrypted message.

For side-condition decomposition, the solver returns a value for the first part of the input that is processed by g_1 and g_3. The goal is to find a matching value for the rest of the input that is processed by g_2, such that the predicate p holds. For instance, in Figure 8.1, g_1 corresponds to the function `compute_sha1`, g_2 is the identity function copying the value `m->hash`, and p is the equality predicate. We find such a value by executing g_1 forwards, finding a value related to that value by p, and applying the inverse of g_2. A common special case is that g_2 is the identity function and the predicate p is just equality, in which case we only have to re-run g_1. For Figure 8.1, our tool must simply re-apply `compute_sha1` to each new message.

8.3.3 Identification

In this section we address the question of how to automatically identify candidate decomposition sites. Specifically, we first discuss how to identify encoding functions, then how to test if the decomposition is possible, and finally how to find inverses of those encoding functions when needed.

Identifying encoding functions. There are two properties of an encoding function that make it suitable for decomposition and re-stitching. First, the encoding function should be difficult to reason about symbolically. Second, the way the function is used should match one of the decomposition patterns described in Section 8.3.1. Our identification approach checks for these two properties.

An intuition that partially explains why many encoding functions are hard to reason about is that they produce constraints that mix together many parts of the program input, which makes constraint solving difficult. For instance, this is illustrated by a contrast between an encryption function that uses a block cipher in CBC mode, and one that uses a stream cipher. Though the functions perform superficially similar tasks, the block cipher encryption is a barrier to dynamic symbolic execution because of its high mixing, while a stream cipher is not. Because of the lack of mixing, a constraint solver can efficiently determine that a single plaintext byte can be modified by making a change to the corresponding ciphertext byte. We use this intuition for detecting encoding functions for decomposition: the encoding functions we are interested in tend to mix their inputs. But we exclude simple stream ciphers from the class of encoding functions we consider, since it is easy enough to solve them directly.

For identifying encoding functions, we perform a trace-based dependency analysis that is a general kind of taint propagation. Given the selection of any subset of the program state as a taint source, the analysis computes which other parts of the program state have a data dependency on that source. We can potentially track the dependencies of values on any earlier part of the program state, e.g., by treating every output of a function as a dependency (taint) source. But for this work we confine ourselves to using the inputs to the entire program (i.e., from system calls) as dependency sources. To be precise our analysis assigns an identifier to each input byte, and determines, for each value in an execution, which subset of the input bytes it depends on. We call the number of such input bytes the value's *taint degree*. If the taint degree of a byte is larger than a configurable threshold, we refer to it as high-taint-degree. We group together a series of high-taint-degree values in adjacent memory locations as a single buffer; our decomposition applies to a single such buffer.

This basic technique could apply to buffers anywhere in an execution, but we further enhance it to identify functions that produce high-taint-degree buffers as output. This has several benefits: it reduces the number of candidate buffers that need to be checked in later stages, and in cases where the tool needs to later find an inverse of a computation, it is convenient to search using a complete function. Our tool considers a buffer to be an output of a function if it is live at the point in time that a return instruction is executed. Also, to ensure we identify a function that includes the complete encoding functionality, our tool uses the dependency analysis to find the first high-taint-degree computation that the output buffer depends on, and chooses the function that encloses both this first computation and the output buffer.

In the example of Figure 8.1, the output buffers of `aes_cbc_decrypt` and `compute_sha1` would both be found as candidates by this technique, since they both would contain bytes that depend on all of the input bytes (the final decrypted byte, and all of the hash value bytes).

This identification process may need to be run in each iteration of the exploration because new encoding functions may appear that had not been seen in previous iterations. As an optimization, the tool runs the identification on the first iteration of the exploration, and then, on each new iteration, it checks whether the solver times out when solving any constraint. If it does, it re-runs the identification on the current execution trace.

Checking dependence conditions. Values with a high taint degree as identified above are candidates for decomposition because they are potentially problematic for symbolic reasoning. But to apply decomposition to them, they must also appear in a proper context in the program. Intuitively, the structure of the program must be like those in Figure 8.3B and Figure 8.3C. To be more precise, we describe (in-)dependence conditions that limit what parts of the program may use values produced by other parts of the program. The next step in our identification approach is to verify that the proper dependence conditions hold (on the observed execution). This checking is needed to avoid improper decompositions, and it also further filters the potential encoding functions identified based on taint degree.

Intuitively, the dependence conditions require that the encoding function be independent of the rest of the program, except for the specific relationships we expect. For serial decomposition, our tool checks that the input bytes that were used as inputs to the surjective transformation are not used later in the program. For example, if a program has the following code:

```
m’ = Dec(m);
if (m[0] == 5) && (m’[0] == 5) then {
  process();
}
```

our approach flags that serial decomposition is not possible because the input m is used after it is decrypted by Dec. Otherwise the solver would return that in order to reach the *process()* function both $m'[0]$ and $m[0]$ need to have value 5 but it has no way of generating a program input m that satisfies those two constraints since the relationship introduced by Dec has been removed by the decomposition step (and if it had not been removed it would be to difficult to reason about).

For side-condition decomposition, our tool checks that the result of the free side-condition predicate is the only use of the value computed from the main input (e.g., the computed checksum), and that the remaining input (e.g., the expected checksum from a header) is not used other than in the free side-condition. Our tool performs this checking using the same kind of dynamic dependency analysis used to measure taint degree. In the example of Figure 8.1, our tool checks that the encrypted input to `aes_cbc_decrypt` is not used later in the program (it cannot be, because it is overwritten). It also checks that the hash buffer pointed to by p is not used other than in the `memcmp` on line 11, and that the buffer `m->hash`, containing the expected hash value, is not used elsewhere.

Identifying inverse functions. Recall that to re-stitch inputs after serial decomposition, our approach requires the inverses of surjective transformation functions. This requirement is reasonable because surjective functions like decryption and decompression are commonly the inverses of other functions (encryption and compression) that apply to arbitrary data. These functions and their inverses are often used in concert, so their implementations can often be found in the same binaries or in publicly available libraries (e.g., [240, 165]).

To locate the relevant inverse functions in the code being analyzed, as well as in publicly available libraries, we check whether two functions are each others' inverses by random testing. If f and f' are two functions, and for several randomly-chosen x and y, $f'(f(x)) = x$ and $f(f'(y)) = y$, then f and f' are likely inverses of each other over most of their domains. Suppose f is the encoding function we wish to invert. Starting with all the functions from the same binary module that were exercised in the trace, we infer their interfaces using our BCR tool (described in Chapter 4). To prioritize the candidates, we use the intuition that the encryption and decryption functions likely have similar interfaces. Note that here it does not matter the full semantics of the parameters, just the syntax and the limited semantics we extract like pointers, lengths, and keys. For example, it does not matter for an asymmetric function that the encryption takes as input the public key and the decryption takes as input the private key; both are just considered keys. For each candidate inverse g, we compute a 4-element feature vector counting how many of the parameters are used only for input, only for output, or both, and how many are pointers. We then sort the candidates in increasing order of the Manhattan distances (sum of absolute differences) between their features and those of f.

For each candidate inverse g, we execute $f \circ g$ and $g \circ f$ on k random inputs each, and check whether they both return the original inputs in all cases. If so, we consider g to be the inverse of f. To match the output interface of g with the input interface of f, and vice versa, we generate missing inputs either according to the semantics inferred by BCR (such as buffer lengths), or randomly; if there are more outputs than inputs we test each possible mapping. Increasing the parameter k improves the confidence in the resulting identification, but the choice of the parameter is not very sensitive: test buffers have enough entropy that even a single false positive is unlikely, but since the tests are just concrete executions, they are inexpensive. If we do not find an inverse among the executed functions in the same module, we expand the search to other functions in the binary, in other libraries shipped with the binary, and in standard libraries.

For instance, in the example of Figure 8.1, our tool requires an AES encryption function to invert the AES decryption used by the bot program. In bots, it is common for the encryption function to appear in the same binary, since the bot often encrypts its reply messages with the same cipher, but in the case of a standard function like AES we could also find the inverse in a standard library like OpenSSL [165].

Once an inverse function is identified, we use BCR to extract the function [24]. The hybrid disassembly technique used by BCR extracts the body of the function, including instructions that did not appear in the execution, which is important because when re-stitching a partial input branches leading to those, previously unseen, instructions may be taken. We further discuss the availability of inverse functions in Section 8.6.2.

8.4 Implementation

In this section we provide some implementation details including the vulnerability detection techniques we use and our Internet-in-a-Workstation environment.

Vulnerability detection. Our tool supports several techniques for vulnerability detection and reports any inputs flagged by these techniques. It detects program termination and invalid memory access exceptions. Executions that exceed a timeout are flagged as potential infinite loops. It also uses TEMU's taint propagation module to identify whether the input (e.g., network data) is used in the program counter or in the size parameter of a memory allocation.

Decomposition and re-stitching details. Following the approach introduced in Section 8.3.1, our tool implements decomposition by making local modifications to the constraints generated from execution, with some additional optimizations. For serial decomposition, it uses hooks (Section 2.3.2) to implement the renaming of symbolic values. As a further optimization, the hook temporarily disables taint propagation inside the encoding function so that no symbolic constraints are generated. To save the work of recomputing a checksum on each iteration in the case of side-condition decomposition, our tool can also directly force the conditional branch implementing the side condition to take the same direction it did on the original execution.

Internet-in-a-workstation. We have developed an environment where we can run malware in isolation, without worrying about the containment problem, i.e., without worrying about malicious behavior leaking to the Internet. Many malware programs, e.g., bots, act as network clients that start connections to remote C&C servers. Thus, the input that our tool needs to feed to the program in each iteration is often the response to some request sent by the program.

All network traffic generated by the program, running in the execution monitor, is redirected to the local workstation in a manner that is transparent to the program under analysis. In addition, we have developed two helper tools: a modified DNS server which can respond to any DNS query with a preconfigured or randomly generated IP address, and a generic replay server. The generic replay server takes as input an XML file that describes a network dialog as an ordered sequence of connections, where each connection can comprise multiple messages in either direction. It also takes as input the payload of the messages in the dialog. Such a generic server simplifies the task of setting up different programs and protocols. Given a network trace of the communication we generate the XML file describing the dialog to explore, and give the replay server the seed messages for the exploration. Then, at the beginning of each exploration iteration our tool hands new payload files (i.e., the re-stitched program input) to the replay server so that they are fed to the network client program under analysis when it opens a new connection.

8.5 Evaluation

This section evaluates our approach by finding bugs in malware that uses complex encoding functions. It demonstrates that our decomposition and re-stitching approach finds some bugs in malware that would not be found without it, and that it significantly increases the efficiency of the exploration in other cases. It presents the malware bugs we find and shows that these bugs have persisted in the malware families for long periods of time, sometimes years.

Malware samples. The first column of Table 8.1 presents the four popular families of malware that we have used in our evaluation. Three of them (Cutwail, Gheg, and MegaD) are spam bots, while Zbot is a trojan used for stealing private information from compromised hosts. All four malware families act as network clients, that is, when run they attempt to connect to a remote C&C server rather than opening a listening socket and awaiting for commands. All four of them use encryption to obfuscate their network communication, avoid signature-based NIDS detection, and make it harder for analysts to reverse-engineer their C&C protocol. Cutwail, Gheg, and MegaD use proprietary encryption algorithms, while Zbot uses the well-known RC4 stream cipher. In addition to encryption, Zbot also uses an MD5 cryptographic hash function to verify the integrity of a configuration file received from the server.

Experimental setup. For each bot we are given a network trace of the bot communicating with the C&C server, while it runs in a contained network. From the network trace we extract an XML representation of the dialog between the bot and the C&C server, as well as the payload of the network packets in that dialog. This information is needed by the replay server to provide the correct

Name	Program size (KB)	Input size (bytes)	# Inst. ($\times 10^3$)	Decryption		Checksum/hash		Runtime (sec)
				Algorithm	MTD	Algo.	MTD	
Zbot	126.5	5269	1307.3	RC4-256	1	MD5	4976	92
MegaD	71.0	68	4687.6	64-bit block cipher	8	none	n/a	105
Gheg	32.0	271	84.5	8-bit stream cipher	128	none	n/a	5
Cutwail	50.0	269	23.1	byte-based cipher	1	none	n/a	2

Table 8.1: Summary of the applications on which we performed identification of encoding functions. MTD stands for maximum taint degree.

sequence of network packets to the bot during exploration. For example, this is needed for MegaD where the response sent by the replay server comprises two packets that need to be sent sequentially but cannot be concatenated together due to the way that the bot reads from the socket. As a seed for the exploration we use the same content observed in the dialog captured in the network trace. Other seeds can alternatively be used. Although our setup can support exploring multiple connections, we focus the exploration on the first connection started by the bot. For the experiments we run our tool on a 3GHz Intel Core 2 Duo Linux workstation with 4GB of RAM running Ubuntu Server 9.04. The emulated guest system where the malware program runs is a Microsoft Windows XP SP3 image with 512MB of emulated RAM.

8.5.1 Identification of Encoding Functions and Their Inverses

The first step in our approach is to identify the encoding functions. The identification of the encoding functions happens on the execution trace produced by the seed at the beginning of the exploration. We set the taint degree threshold to 4, so that any byte that has been generated from 5 or more input bytes is flagged. Table 8.1 summarizes the results. The identification finds an encoding function in three of the four samples: Gheg, MegaD, and Zbot. For Cutwail, no encoding function is identified. The reason for this is that Cutwail's cipher is simple and does not contain any mixing of the input, which is the property that our encoding function identification technique detects. Without input mixing the constraints generated by the cipher are not complex to solve. We show this in the next section. In addition, Cutwail's trace does not contain any checksum functions. The identification does not throw any false positives.

For Zbot, the encoding function flagged in the identification corresponds to the MD5 checksum that it uses to verify the integrity of the configuration file it downloads from the C&C server. In addition to the checksum, Zbot uses the RC4 cipher to protect its communication, which is not flagged by our technique. This happens because RC4 is a stream cipher that does no mixing of the input, i.e., it does not use input bytes to update its internal state, only the key which is a constant for the solver. The input is simply combined with a pseudo-random keystream using bit-wise exclusive-

or. Since the keystream is not derived from the input but from a key in the data section, it is concrete for the solver. Thus, the solver only needs to invert the exclusive-or computation to generate an input, which means that RC4 introduces no hard-to-solve constraints.

For the other two samples (Gheg and MegaD) the encoding function flagged by the identification corresponds to the cipher. MegaD uses a 64-bit block cipher, which mixes 8 bytes from the input before combining them with the key. Gheg's cipher uses a one-byte key that is combined with the first input byte to produce a one-byte output that is used also as key to encode the next byte. This process repeats and the mixing (taint degree) of each new output byte increases by one. Neither Gheg nor MegaD uses a checksum.

Once the encoding functions have been identified, our tool introduces new symbols for the outputs of those encoding functions, effectively decomposing the constraints in the execution into two sets and ignoring the set of hard-to-solve constraints introduced by the encoding function.

The results of our encoding function identification, for the first iteration of the exploration, are summarized in Table 8.1, which presents on the left the program name and program size, the size of the input seed, and the number of instructions in the execution trace produced by the seed. The decryption and checksum columns describe the algorithm type and the maximum taint degree (MTD) the algorithm produces in the execution. The rightmost column shows the runtime of the identification algorithm, which varies from a few seconds to close to two minutes. Because the identification is reused over a large number of iterations, the amortized overhead is even smaller.

Identifying the inverse functions. For Gheg and MegaD, our tool needs to identify the inverse of the decryption function so that it can be used to re-stitch the inputs into a new program input for another iteration. The encryption function for MegaD is the same one identified in Chapter 4 using the technique that flags functions with a high ratio of bitwise and arithmetic instructions. We use it to check the accuracy of our new identification approach.

As described in Section 8.3.3, our tool extracts the interface of each function in the execution trace that belongs to the same module as the decoding function, and then prioritizes them by the similarity of their interface to the decoding function. For both Gheg and MegaD, the function with the closest prototype is the encryption function, as our tool confirms by random testing with $k = 10$ tests. These samples illustrate the common pattern of a matching encryption function being included for two-way communication, so we did not need to search further afield for an inverse.

8.5.2 Stitched vs. Non-Stitched

In this section we compare the number of bugs found by our tool when it uses decomposition and re-stitching, which we call *full* exploration, and when it does not, which we call *vanilla* exploration.

Name	Vulnerability type	Disclosure public identifier	Encoding functions	Search time (min.)	
				full	vanilla
Zbot	Null dereference	OSVDB-66499 [170]	checksum	17.8	>600
	Infinite loop	OSVDB-66500 [169]	checksum	129.2	>600
	Buffer overrun	OSVDB-66501 [168]	checksum	18.1	>600
MegaD	Process exit	n/a	decryption	8.5	>600
Gheg	Null dereference	OSVDB-66498 [167]	decryption	16.6	144.5
Cutwail	Buffer overrun	OSVDB-66497 [166]	none	39.4	39.4

Table 8.2: Description of the bugs our tool finds in malware. The column "full" shows the results using stitched dynamic symbolic execution, while the "vanilla" column gives the results with traditional (non-stitched) dynamic symbolic execution. ">600" means the tool run for 10 hours and did not find the bug.

Full exploration uses the identified decoding functions to decompose the constraints into two sets, one with the constraints introduced by the decryption/checksum function and the other with the remaining constraints after that stage. In addition, each iteration of the MegaD and Gheg explorations uses the inverse function to re-stitch the inputs into a program input. Vanilla exploration is comparable to previous dynamic symbolic execution tools. In both full and vanilla cases, our tool detects bugs using the techniques described in Section 8.4.

In each iteration of the exploration, our tool collects the execution trace of the malware program starting from the first time it receives network data. It stops the trace collection when the malware program sends back a reply, closes the communication socket, or a bug is detected. If none of those conditions is satisfied the trace collection is stopped after 2 minutes. For each collected trace, our tool analyzes up to the first 200 input-dependent control flow branches and automatically generates new constraints that would explore new paths in the program. It then queries STP to solve each generated set of constraints, uses the solver's response to generate a new input, and adds it to the pool of inputs to test on future iterations. Because constraint solving can take a very long time without yielding a meaningful result, our tool discards a set of constraints if STP runs out of memory or exceeds a 5-minute timeout for constraint solving.

We run both vanilla and full explorations for 10 hours and report the bugs found, which are summarized in Table 8.2. Detailed descriptions of the bugs follow in Section 8.5.3. We break the results in Table 8.2 into three categories. The first category includes Zbot and MegaD for which full exploration finds bugs but vanilla exploration does not. Full exploration finds a total of 4 bugs, three in Zbot and one in MegaD. Three of the bugs are found in under 20 minutes and the second Zbot bug is found after 2 hours. Vanilla exploration does not find any bugs in the 10-hour period. This happens due to the complexity of the constraints being introduced by the encoding functions.

In particular, using full exploration the 5-minute timeout for constraint solving is never reached and STP never runs out of memory, while using vanilla exploration more than 90% of the generated constraints result in STP running out of memory.

The second category comprises Gheg for which both vanilla and full explorations find the same bug. Although both tools find the same bug, we observe that vanilla exploration requires almost ten times as long as full exploration to do so. The cipher used by Gheg uses a one-byte hard-coded key that is combined with the first input byte using bitwise exclusive-or to produce the first output byte, that output byte is then used as key to encode the second byte also using bitwise exclusive-or and so on. Thus, the taint degree of the first output byte is one, for the second output byte is two and so on until the maximum taint degree of 128 shown in Table 8.1. The high maximum taint degree makes it harder for the solver to solve and explains why vanilla exploration takes much longer than full exploration to find the bug. Still, the constraints introduced by the Gheg cipher are not as complex as the ones introduced by the Zbot and MegaD ciphers and the solver eventually finds solutions for them. This case shows that even in cases where the solver will eventually find a solution, using decomposition and re-stitching can significantly improve the performance of the exploration.

The third category comprises Cutwail for which no encoding functions with high taint degree are identified and thus vanilla exploration and full exploration are equivalent.

In summary, full exploration using decomposition and re-stitching clearly outperforms vanilla exploration. Full exploration finds bugs in cases where vanilla exploration fails to do so due to the complexity of the constraints introduced by the encoding functions. It also improves the performance of the exploration in other cases were the encoding constraints are not as complex and will eventually be solved.

8.5.3 Malware Vulnerabilities

In this section we present the results of our manual analysis to understand the bugs discovered by our tool and our experiences reporting the bugs. Note that all vulnerabilities have been validated by replaying the inputs found by the exploration to the bot programs and monitoring how they crash the bot's process. In addition, for the Zbot buffer overrun vulnerability we have created an exploit that hijacks execution.

Zbot. Our tool finds three bugs in Zbot. The first one is a null pointer dereference. One of the C&C messages contains an array size field, which the program uses as the size parameter in a call to *RtlAllocateHeap*. When the array size field is larger than the available memory left in its local heap, the allocation returns a null pointer. The return value of the allocation is not checked by the program, which later attempts to write to the buffer, crashing when it dereferences the null pointer.

The second bug is an infinite loop condition. A C&C message comprises of a sequence of blocks. Each block has a 16-byte header and a payload. One of the fields in the header represents the size of the payload, s. When the trojan program finishes processing a block, it iteratively moves to the next one by adding the block size, $s + 16$, to a cursor pointer. When the value of the payload size is $s = -16$, the computed block size becomes zero, and the trojan keeps processing the same block over and over again.

The last bug is a stack buffer overrun. As mentioned above, a C&C message comprises of a sequence of blocks. One of the flags in the block header determines whether the block payload is compressed or not. If the payload is compressed, the trojan program decompresses it by storing the decompressed output into a fixed-size buffer located on the stack. When the length of the decompressed payload is larger than the buffer size, the program will write beyond the buffer. If the payload is large enough, it will overwrite a function return address and can eventually lead to control flow hijacking. Thus, this vulnerability is exploitable and we have successfully crafted a C&C message that exploits the vulnerability and hijacks the execution of the malware.

MegaD. Our tool finds one input that causes the MegaD bot to exit cleanly. We analyzed this behavior using the MegaD grammar in Appendix A and found that the bug is present in the handling of the *ping* message (type `0x27`). If the bot receives a ping message and the bot identifier (usually set by a previously received C&C message) has not been set, then it sends a reply *pong* message (type `0x28`) and terminates. This behavior highlights the fact that, in addition to bugs, our stitched dynamic symbolic execution can also discover C&C messages that cause the malware to cleanly exit (e.g., kill commands), if those commands are available in the C&C protocol. These messages cannot be considered bugs but can still be used to disable the malware. They are specially interesting because they may have been designed to completely remove all traces of the malware running in the compromised host. In addition, their use could raise fewer ethical and legal questions than the use of an exploit would.

Gheg. Our tool finds one null pointer dereference bug in Gheg. The bug is similar to the one in Zbot. One of the C&C messages contains an array size field, whose value is multiplied by a constant (0x1e8) and the result used as the size parameter in a call to *RtlAllocateHeap*. The program does not check the return value of the allocation and later writes into the allocated buffer. When the array size field value is larger than the available memory in its local heap, the allocation fails and a null pointer is returned. The program fails to check that the returned value is a null pointer and tries to dereference it.

Family	MD5	First seen	Reported by
Zbot	0bf2df85*7f65	Jun-23-09	Prevx
	1c9d16db*7fc8	Aug-17-09	Prevx
	7a4b9ceb*77d6	Dec-14-09	ThreatExpert
MegaD	700f9d28*0790	Feb-22-08	Prevx
	22a9c61c*e41e	Dec-13-08	Prevx
	d6d00d00*35db	Feb-03-10	VirusTotal
	09ef89ff*4959	Feb-24-10	VirusTotal
Gheg	287b835b*b5b8	Feb-06-08	Prevx
	edde4488*401e	Jul-17-08	Prevx
	83977366*b0b6	Aug-08-08	ThreatExpert
	cdbd8606*6604	Aug-22-08	Prevx
	f222e775*68c2	Nov-28-08	Prevx
Cutwail	1fb0dad6*1279	Aug-03-09	Prevx
	3b9c3d65*07de	Nov-05-09	Prevx

Table 8.3: Bug reproducibility across different malware variants. The shaded variants are the ones used for exploration.

Cutwail. Our tool finds a buffer overrun bug that leads to an out-of-bounds write in Cutwail. One of the received C&C messages contains an array. Each record in the array has a length field specifying the length of the record. This field is used as the size parameter in a call to *RtlAllocateHeap*. The returned pointer is appended to a global array that can only hold 50 records. If the array in the received message has more than 50 records, the 51^{st} record will be written outside the bounds of the global array. Near the global array, there exists a pointer to a private heap handle and the out-of-bounds write will overwrite this pointer. Further calls to *RtlAllocateHeap* will then attempt to access the malformed heap handle, and will lead to heap corruption and a crash.

Reporting the bugs. We reported the Gheg bug to the editors of the Common Vulnerabilities and Exposures (CVE) database [56]. Our suggestion was that vulnerabilities in malware should be treated similarly to vulnerabilities in commercial or open source programs, of course without reporting back to the developers. However, the CVE editors felt that malware vulnerabilities were outside the scope of their database. Subsequently, we reported the Gheg vulnerability to the Open Source Vulnerability Database (OSVDB) moderators who accepted it. Since then, we have reported all other vulnerabilities except the MegaD one, which may be considered intended functionality by the botmaster. Table 8.2 presents the public identifiers for the disclosed vulnerabilities. We further address the issue of disclosing malware vulnerabilities in Section 8.6.

8.5.4 Bug Persistence over Time

Bot binaries are updated very often to avoid detection by anti-virus tools. One interesting question is how persistent over time are the bugs found by our tool. To evaluate this, we retest our crashing inputs on other binaries from the same malware families. Table 8.3 shows all the variants, with the shaded variants corresponding to the ones explored by our tool and mentioned in Table 8.1.

We replay the input that reproduces the bug our tool found on the shaded variant on the rest of variants from the same family. The bugs are reproducible across all the variants we tested. These means for instance that the MegaD bug has been present for at least two years (the time frame covered by our variants). In addition, the MegaD encryption and decryption functions (and the key they use), as well as the C&C protocol have not changed, or barely evolved, through time. Otherwise the bug would not be reproducible in older variants. The results for Gheg are similar. The bug reproduces across all Gheg variants, although in this case our most recent sample is from November, 2008. Note that, even though the sample is relatively old it still works, meaning that it still connects to a C&C server on the Internet and sends spam. For Zbot, all three bugs reproduce across all variants, which means they have been present for at least 6 months. These results are important because they demonstrate that there are components in bot software, such as the encryption functions and C&C protocol grammar, that tend to evolve slowly over time and thus could be used to identify the family to which an unknown binary belongs, one widespread problem in malware analysis.

8.6 Discussion

In light of our results, this section provides additional discussion on the applications for the discovered bugs and associated ethical considerations. Then, it presents a potential scenario for using the discovered bugs, and describes some limitations of our approach.

8.6.1 Applications and Ethical Considerations

Malware vulnerabilities could potentially be used in different “benign” applications such as remediation of botnet infestations, for malware genealogy since we have shown that the bugs persist over long periods of time, as a capability for law enforcement agencies, or as a strategic resource in state-to-state cyberwarfare [171]. However, their use raises important ethical and legal questions. For example, there may be a danger of significant negative consequences, such as adverse effects to the infected machines. Also, it is unclear which legal entity would perform such remediation, and whether there exists any entity with the legal right to take such action. On the other hand, having a potential avenue for cleanup and not making use of it also raises some ethical concerns since if such remediation were effective, it would be a significant service to the malware's future third-party victims (targets of DDoS attacks, spam recipients, etc.). Such questions belong to recent and ongoing discussions about ethics in security research (e.g., [62]) that have not reached a firm conclusion. In our view, malware vulnerabilities are a capability that should only be used as a last resort when other solutions such as detection and cleanup are not possible, perhaps because the infected computers cannot be easily disconnected from the network or immediate action is required.

Malware vulnerabilities could also be used for malign purposes. For instance, there are already indications that attackers are taking advantage of known vulnerabilities in web interfaces used to administer botnets to hijack each others' botnets [58]. This raises concerns about disclosing such bugs in malware. In the realm of vulnerabilities in benign software, there has been significant debate on what disclosure practices are socially optimal and there is a partial consensus in favor of some kind of "responsible disclosure" that gives authors a limited form of advance notice. However, it is not clear what the analogous best practice for malware vulnerabilities should be. We have faced this disclosure issue when deciding whether to publicly disclose the vulnerabilities we found and to which extent we should describe the vulnerabilities. We have decided in favor of disclosing the vulnerabilities we found to raise awareness of the fact that such vulnerabilities exist and can be exploited. We also believe further discussion on the proper avenue for disclosing malware vulnerabilities would be beneficial.

Potential application scenario. While we have not used our crashing inputs on bots in the wild, here we hypothetically discuss one possible scenario of how one might do so. The malware programs we analyze start TCP connections with a remote C&C server. To exploit the vulnerabilities we have presented, we need to impersonate the C&C server and feed inputs in the response to the initial request from the malware program. This scenario often happens during a botnet takedown, in which law enforcement or other responding entities identify the IP addresses and DNS names associated with the C&C servers used by a botnet, and appeal to relevant ISPs and registrars to have them de-registered or redirected to the responders. The responders can then impersonate the C&C server: one common choice is a *sinkhole server* that collects statistics on requests but does not reply. But such responders are also in a position to perform more active communication with bots, and for instance vulnerabilities like the ones we present could be used for cleanup if the botnet does not support cleanup via its normal protocol. For example, such a scenario happened recently during the attempted MegaD takedown by FireEye [152]. For a few days FireEye ran a sinkhole server that received the C&C connections from the bots. This sinkhole server was later handed to the Shadowserver Foundation [191].

8.6.2 Limitations

We have found our techniques to be quite effective against the current generation of malware. But since malware authors have freedom in how they design encoding functions, and an incentive to avoid analysis of their programs, it is valuable to consider what measures they might take against analysis.

Preventing access to inverses. To stitch complete inputs in the presence of a surjective transformation, our approach requires access to an appropriate inverse function: for instance, the encryption function corresponding to a decryption function. So far, we have been successful in finding such inverses either within the malware binary, or from standard sources, but these approaches could be thwarted if malware authors made different choices of cryptographic algorithms. For instance, malware authors could design their protocols using asymmetric (public-key) encryption and digital signatures. Since we would not have access to the private key used by the C&C server, we could not forge the signature in the messages sent to the bot. We could still use our decomposition and re-stitching approach to find bugs in malware, because the signature verification is a basically a free side-condition that can be ignored. However, we could only build an exploit for our modified bot, as other bots will verify the (incorrect) signature in the message and reject it. Currently, most malware does not use public-key cryptography, but that may change. In the realm of symmetric encryption, malware authors could deploy different non-standard algorithms for the server-to-bot and bot-to-server directions of communication: though not theoretically infeasible, the construction of an encryption implementation from a binary decryption implementation might be challenging to automate. For instance, Kolbitsch et al. [109] faced such a situation in recreating binary updates for the Pushdo trojan, which was feasible only because the decryption algorithm used was weak enough to be inverted by brute force for small plaintexts.

Obfuscating encoding functions. Malware authors could potentially keep our tool from finding encoding functions in binaries by obfuscating them. General purpose packing is not an obstacle to our dynamic approach, but more targeted kinds of obfuscation would be a problem. For instance, our current implementation recognizes only standard function calls and returns, so if a malware author rewrote them using non-standard instructions our tool would require a corresponding generalization to compensate. Further along the arms race, there are also fundamental limitations arising from our use of a dynamic dependency analysis such as the use of implicit flows (e.g., `if (x == 1) then y = 1`) for obfuscation. Such limitations are similar to previously studied limitations of dynamic taint analysis [34]. Other targeted evasion techniques could take advantage of our checks for whether decomposition is possible by using the received encrypted data after decryption, so that our approach decides that serial decomposition is not possible. To partially handle this case we could generalize our analysis to identify uses of the input that do nothing useful with it. In addition, malware authors could add non-encoding functions with high ratios of arithmetic and bitwise instructions to try to disguise the real encoding functions by increasing the number of false positives of our detection technique.

8.7 Related Work

One closely related recent project is Wang et al.'s TaintScope system [219]. Our goals partially overlap with theirs in the area of checksums, but our work differs in three key aspects. First, Wang et al.'s techniques do not apply to decompression or decryption. Second, TaintScope performs exploration based on taint-directed fuzzing [74], while our approach harnesses the full generality of symbolic execution. (Wang et al. use symbolic execution only for inverting the *encodings* of checksums, i.e., simple transformations on the input checksum such as converting from little-endian to big-endian or from hexadecimal to decimal, a task which is trivial in our applications.) Third, Wang et al. evaluate their tool only on benign software, while we perform the first automated study of vulnerabilities in malware.

The encoding functions we identify can also be extracted to be used elsewhere. Our BCR tool (presented in Chapter 4) as well as the Inspector Gadget [109] tool can be used to extract encryption and checksum functionalities, including some of the same ones our tool identifies. This work uses BCR's interface identification techniques to compare the interfaces of functions while identifying inverses. Inspector Gadget [109] can also perform so-called gadget inversion, which is useful for the same reasons as we search for existing inverse functions. However, their approach does not work on strong cryptographic functions.

Previous work has used alternative heuristics to identify cryptographic operations. For instance ReFormat [221] proposes detecting such functions by measuring the ratio of arithmetic and bitwise instructions to other instructions. We have extended the techniques in ReFormat in Section 3.6. Our use of taint degree as a heuristic in this chapter is more specifically motivated by the limitations of symbolic execution: for instance a simple stream cipher would be a target of the previous approaches but is not for this work.

Decomposition is a broad class of techniques in program analysis and verification, but most previous decomposition techniques are symmetric in the sense that each of the sub-components of the program are analyzed similarly, while a key aspect of our approach is that different components are analyzed differently. In analysis and verification, decomposition at the level of functions, as in systems like Saturn [231], is often called a compositional approach. In the context of tools based on dynamic symbolic execution, there is work on compositional approaches that performs dynamic symbolic execution separately on each function in a program [76, 3]. Because this is a symmetric technique, it would not address our problem of encoding functions too complex to analyze even in isolation. More similar to our approach is grammar-based fuzzing [77, 25], an instance of serial decomposition. However parsers require different specialized techniques than encoding functions.

8.8 Conclusion

We have presented a new approach, stitched dynamic symbolic execution, to allow analysis in the presence of complex functions such as decryption and decompression, that would otherwise be difficult to analyze. Our techniques for automated identification, decomposition, and re-stitching bypass encoding functions like decryption and hashing to find bugs in core program logic. Specifically, these techniques enable the first automated study of vulnerabilities in malware. Our tool finds 6 unique bugs in 4 prevalent malware families. These bugs can be triggered over the network to terminate or take control of a malware instance. They have persisted across malware revisions for months, and even years. Our results demonstrate that finding vulnerabilities in malware is technically feasible. In addition, we start a discussion on the many still unanswered questions about the applications and ethical concerns surrounding malware vulnerabilities, an important security resource.

Part V

Conclusion

Chapter 9

Conclusion

9.1 Discussion

In this section we provide further discussion on the challenge of working with malware that uses obfuscation to defeat analysis and some lessons learned on building an execution trace capture infrastructure.

9.1.1 Evasion and the Malware Arms Race

While the techniques presented in this thesis apply to any program in binary form, one important class of programs that we use to evaluate our techniques is malware. Automatic program binary analysis techniques are specially well-suited for malware because malware is only distributed in binary form and is highly dynamic, with new malware families and versions being constantly introduced, and polymorphic and metamorphic variants of each version generated on a daily (or hourly) basis. However, working with malware poses some additional challenges compared with working with benign programs.

One important challenge in working with malware is obfuscation because malware authors have an incentive to avoid analysis of their programs. Thus, they use a wide array of obfuscation techniques to complicate the analysis of their programs' functionality and data. As it often happens in security, there exists an arms race between new obfuscation techniques being developed that take advantage of limitations in state-of-the-art analysis, and new analysis techniques being deployed to fix those limitations. For example, historically, malware writers have used encryption, polymorphism, metamorphism, and other obfuscation methods to protect their programs from anti-virus and static analysis tools [206,42,216]. Those anti-static-analysis techniques do not hamper our dynamic analysis. Malware can target the limited coverage of dynamic analysis by employing trigger-based behaviors such as time or logic bombs, but white-box exploration techniques like the ones introduced in Chapters 6–7 can be used to increase the coverage of dynamic analysis and find such behaviors. Sharif et al. propose using conditional code obfuscation to hamper analysis techniques based on white-box exploration [192]. Their obfuscation technique leverages hashing and encryption to introduce hard-to-solve constraints for dynamic symbolic execution. However, in Chapter 8 we have proposed stitched dynamic symbolic execution, a decomposition and re-stitching approach that enables white-box exploration in the presence of complex encoding functions.

A fundamental premise of dynamic analysis is that the behavior to analyze can be observed by running the program. At the time of writing, the main obfuscation technique used to defeat dynamic analysis is stopping the execution or changing its behavior if a virtualized or emulated environment is detected [37, 69]. Recent research has addressed how to identify and bypass such anti-virtualization checks [101]. Another fundamental premise of our protocol reverse-engineering and model extraction techniques is that we can learn the input structure or model a code fragment's behavior by monitoring its execution. However, if the code that we monitor does not correspond to the code that implements the functionality under study (e.g., the parser or the content sniffing algorithm), our results would be incorrect. Thus, another obfuscation vector is to translate a binary program into an intermediate language (IL) and ship the IL version along with an abstract machine or emulator that interprets it. With such approach, our analysis may extract information about the emulator itself rather than the emulated program. There already exist commercial programs that enable such obfuscation [41, 207, 215]. To defeat such obfuscation, we need to extract the IL code that is executed, as well as its semantics (e.g., the mapping from IL instructions to x86 code) and perform our analysis using that information. Sharif et al. [193] have studied the problem of automatic reverse-engineering a certain class of malware emulators. We believe more work is needed to produce solutions that generalize to any class of malware emulators.

In the long run, we can expect that if the techniques presented in this thesis become prevalent, malware authors may design obfuscations that specifically target our techniques. For example, malware authors could design their code to be hard to reuse, by mixing unrelated functionality, adding unnecessary parameters, or inlining functions, or they could design polymorphic protocols, which constantly change structure. Although such arms race can sometimes be disheartening for a researcher, in this thesis we have found that malware is an excellent playground for applying program analysis techniques because it quickly exposes limitations and hidden assumptions, making the final proposed techniques and their implementations more robust.

9.1.2 Instruction-Level Execution Traces

Our offline approach to dynamic program binary analysis requires capturing information logs during execution. The most expensive of these logs to capture are execution traces because their size grows linearly with the number of instructions executed. For long-running programs (e.g., network servers) it is often not possible to capture execution traces of complete programs runs. In this thesis we have focused on the analysis of specific security-relevant functionality in programs such as the code that parses and builds protocol messages, cryptographic functions, and content-sniffing algorithms. Our experience has been that, to obtain execution traces that capture the functionality of interest, an execution trace capture infrastructure needs to provide fine-grained triggers that allow an analyst to select when to start and stop trace capture.

One important design goal of an instruction-level execution trace capture infrastructure is to minimize the amount of data that needs to be logged per executed instruction, without losing any information about the execution. However, our experience and that of other related work [13], shows that there exist numerous engineering issues that need to be addressed to obtain a balance between the run-time and space constraints of execution trace collection and the speed of accessing the information in the traces by external applications that consume them.

9.2 Conclusion

Closed-source programs are prevalent in computer systems. In this thesis we have developed dynamic program binary analysis techniques that enable an analyst, with no access to the program's source code, to extract the grammar of undocumented program inputs, to reuse fragments of binary code, and to model security-relevant functionality directly from program binaries. We have applied our techniques to enable and enhance a variety of security applications: active botnet infiltration, deviation detection, finding filtering-failure attacks, vulnerability-based signature generation, and vulnerability discovery.

To extract the grammar of undocumented program inputs, we have presented a new approach for automatic protocol reverse-engineering that uses dynamic program binary analysis to reverse the protocol from the binary of an application that implements it. When the protocol specification is not available, the applications implementing the protocol are the richest source of information about the protocol. We have proposed dynamic analysis techniques for message format extraction and field semantics inference. Our techniques can extract the format and semantics of the protocol messages in both directions of the communication, even when an analyst has access only to the application implementing one side of the communication. We have used our protocol reverse-engineering techniques to extract the grammar of the previously undocumented, encrypted, C&C protocol used by MegaD, a prevalent spam botnet.

To reuse binary code, we have developed automatic techniques for identifying the interface and extracting the instructions and data dependencies of a code fragment from a program binary. Our techniques extract a fragment of binary code so that it is self-contained and can be reused by external source code, independently of the rest of the functionality in the program binary. We have applied our binary code reuse techniques to extract cryptographic functions used by malware, including the encryption, decryption, and key generation functions that MegaD bots use to protect its C&C protocol. Using the C&C protocol grammar extracted by our protocol reverse-engineering techniques and the cryptographic functions extracted by our binary code reuse techniques, we have enabled a network intrusion detection system to decrypt a C&C message flowing through the network, parse the message contents, rewrite some fields in the message, re-encrypt it, and send it on the network. We have used this enhanced network intrusion detection system to enable active botnet infiltration by rewriting messages to convince the botmaster that a bot under our control can send spam, when all the spam traffic sent by the bot is blocked, enabling us to log the spam-related information sent by the botmaster.

To model security-relevant functionality we have proposed model extraction techniques that work directly on program binaries. Our model extraction techniques use white-box exploration to produce high coverage models of the functionality. We enhance previous white-box exploration techniques in three ways. For programs that use strings, our string-enhanced white-box exploration improves the exploration by reasoning directly on strings, rather than individual bytes that form the strings. For programs that parse complex, highly-structured inputs, our protocol-level constraint-guided exploration improves the exploration by using the protocol grammar to reason directly on protocol fields rather than individual input bytes. For programs that use complex encoding functions such as hashing or encryption, our stitched dynamic symbolic execution enables exploring beyond those encoding functions and into the core functionality of the program. For this, it decomposes the symbolic constraints into hard-to-solve constraints induced by the encoding functions and easier constraints induced by the rest of the execution, solves the easier constraints to obtain a partial input, and re-stitches a complete input using the encoding functions and their inverses. We have used our model extraction techniques to model security-relevant functionality on a variety of real-world programs that include malware, Web browsers, and network servers. Our models enable applications such as finding deviations between two implementations of the same protocol, discovering content-sniffing XSS attacks on Web applications, generating protocol-level vulnerability-based signatures, and discovering vulnerabilities in malware.

Altogether, we have built a platform that provides protocol reverse-engineering, binary code reuse, and model extraction modules of functionality for analyzing security-relevant code from program binaries, as well as a variety of tools for dynamic program binary analysis. We have demonstrated the utility of our novel dynamic program binary analysis techniques and approaches on a variety of real world security problems and provided a basis for further techniques to be built on top of our work. We envision techniques that would address problems such as further integration of dynamic and static analysis, deconstruction of the program into functional components, fine-grained analysis of the program state, and analysis of control dependence and implicit flows.

Appendix A

MegaD BinPac grammar

```
type MegaD_Message(is_inbound: bool) = record {
  msg_len : uint16;
  encrypted_payload(is_inbound):
    bytestring &length = 8 * msg_len;
} &byteorder = bigendian;

type encrypted_payload(is_inbound: bool) = record {
  version : uint16; # Constant(0x0100 or 0x0001)
  mtype : uint16;
  data : MegaD_data(is_inbound, mtype);
};

# Known message types
type MegaD_data(is_inbound: bool, msg_type: uint16) =
  case msg_type of {
    0x00 -> m00: msg_0x0;
    0x01 -> m01: msg_0x1;
    0x02 -> m02: msg_0x2;
    0x03 -> m03: msg_0x3;
    0x04 -> m04: msg_0x4;
    0x05 -> m05: msg_0x5;
    0x06 -> m06: msg_0x6;
    0x07 -> m07: msg_0x7;
    0x09 -> m09: msg_0x9;
    0x0a -> m0a: msg_0xa;
    0x0d -> m0d: msg_0xd;
    0x0e -> m0e: msg_0xe;
    0x15 -> m15: msg_0x15;
    0x16 -> m16: msg_0x16;
    0x18 -> m18: msg_0x18;
    0x1c -> m1c: msg_0x1c(is_inbound);
    0x1d -> m1d: msg_0x1d;
    0x21 -> m21: msg_0x21;
    0x22 -> m22: msg_0x22;
    0x23 -> m23: msg_0x23;
    0x24 -> m24: msg_0x24;
    0x25 -> m25: msg_0x25;
    0x27 -> m27: msg_0x27;
    0x28 -> m28: msg_0x28;
    default -> unknown : bytestring &restofdata;
};

type msg_0x0 = record {
  msg0_type : uint8;  # Type of message 0
  msg0_data : MegaD_msg0(msg0_type);
};

# Direction: outbound (To: CC server)
# MegaD supports two subtypes for type zero
type msg_0x0 = record {
  fld_00 : uint8; # <unknown>
  fld_01 : MegaD_msg0(fld_00);
};

type MegaD_msg0(msg0_type: uint8) =
  case msg0_type of {
    0x00 -> m00 : msg_0x0_init;
    0x01 -> m01 : msg_0x0_idle;
    default -> unknown : bytestring &restofdata;
};

type msg_0x0_init = record {
  fld_00 : bytestring &length=16; # Constant(0)
  fld_01 : uint32; # Constant(0xd)
  fld_02 : uint32; # Constant(0x26)
  fld_03 : uint32; # IP address
  pad : bytestring &restofdata; # Padding
};

type msg_0x0_idle = record {
  fld_00 : bytestring &length=8; # Bot ID
  fld_01 : uint32; # Constant(0)
  pad : bytestring &restofdata; # Padding
};
```

```
# Direction: inbound (From: CC server)
type msg_0x1 = record {
  fld_00 : bytestring &length=16; # Cookie
  fld_01 : uint32; # Sleep Timer
  fld_02 : bytestring &length=8; # Bot ID
};

# Direction: inbound (From: CC server)
type msg_0x2 = record {
  fld_00 : uint16; # <unknown>
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (To: CC server)
type msg_0x3 = record {
  fld_00 : uint32; # Cookie
  fld_01 : bytestring &length=8; # Bot ID
};

# Direction: inbound (From: CC server)
type msg_0x4 = record {
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (To: CC server)
type msg_0x5 = record {
  fld_00 : uint32;  # Error code
  fld_01 : uint32; # Cookie
  fld_02 : bytestring &length=8; # Bot ID
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (To: CC server)
type msg_0x6 = record {
  fld_00 : uint32; # Cookie
  fld_01 : bytestring &length=8; # Bot ID
  fld_02 : uint32; # Constant(0)
  pad : bytestring &restofdata; # Padding
};

# Direction: inbound (From: CC server)
type msg_0x7 = record {
  pad : bytestring &restofdata; # Padding
};
```

```
# Direction: outbound (To: CC server)
type msg_0x9 = record {
  fld_01 : bytestring &length=8; # Bot ID
  fld_02 : uint32; # Constant(0)
};
# Direction: inbound (From: CC server)
type msg_0xa = record {
  pad : bytestring &restofdata; # Padding
};

# Direction: inbound (From: CC server)
type msg_0xd = record {
  fld_00 : uint32; # Cookie
  fld_01 : uint32; # <unused>
  fld_02 : uint16; # Length(fld_03)
  fld_03 : bytestring &length=fld_02; # URL
  pad : bytestring &restofdata; # Padding
};

# Direction: inbound (From: CC server)
type msg_0xe = record {
  pad : bytestring &restofdata; # Padding
};

# Direction: inbound (From: CC server)
type msg_0x15 = record {
  pad : bytestring &restofdata; # Padding
};

type host_info = record {
  fld_00 : uint32; # CPU identifier
  fld_01 : uint32; # Tick difference
  fld_02 : uint32; # Tick counter
  fld_03 : uint16; # OS major version
  fld_04 : uint16; # OS minor version
  fld_05 : uint16; # OS build number
  fld_06 : uint16; # Service pack major
  fld_07 : uint16; # Service pack minor
  fld_08 : uint32; # Physical memory(KB)
  fld_09 : uint32; # Available memory(KB)
  fld_10 : uint16; # Internet conn. type
  fld_11 : uint32; # IP address
};
```

```
# Direction: outbound (To: CC server)
type msg_0x16 = record {
  fld_00 : bytestring &length=8; # Bot ID
  fld_01 : uint16; # Length(fld_02)
  fld_02 : host_info; # Host information
  pad : bytestring &restofdata; # Padding
};

# Direction: inbound (From: CC server)
type msg_0x18 = record {
  pad : bytestring &restofdata; # Padding
};

# Direction: inbound or outbound (Template server)
type msg_0x1c(is_inbound: bool) =
  case is_inbound of {
    true  -> m1c_inbound : msg_0x1c_inbound;
    false -> m1c_outbound : msg_0x1c_outbound;
};

# Direction: inbound (From: Template server)
type msg_0x1c_inbound = record {
  fld_00 : uint32; # Stored data
  fld_01 : uint32; # Length
  fld_02 : uint32; # Length(fld_03)
  fld_03 : bytestring &length = fld_02; # Compressed
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (To: Template server)
type msg_0x1c_outbound = record {
  fld_00 : bytestring &length = 16; # Cookie
  fld_01 : uint32; # Constant(0)
};

# Direction: outbound (To: Template server)
type msg_0x1d = record {
  fld_00 : bytestring &length = 16; # Cookie
  fld_01 : uint32; # Constant(0)
};
```

```
# Direction: inbound (From: CC server)
type msg_0x21 = record {
  fld_00 : uint32; # <unknown>
  fld_01 : uint16; # Port
  fld_02 : uint8[] &until($element == 0); # Hostname
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (To: CC server)
type msg_0x22 = record {
  fld_00 : bytestring &length=8; # Bot ID
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (To: CC server)
type msg_0x23 = record {
  fld_00 : uint32; # Error code
  fld_01 : bytestring &length=8; # Bot ID
};

# Direction: inbound (From: CC server)
type msg_0x24 = record {
  fld_00 : uint32; # IP address
  fld_01 : uint16; # Port
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (To: CC server)
type msg_0x25 = record {
  fld_00 : bytestring &length=8; # Bot ID
  pad : bytestring &restofdata; # Padding
};

# Direction: inbound (From: CC server)
type msg_0x27 = record {
  pad : bytestring &restofdata; # Padding
};

# Direction: outbound (To: CC server)
type msg_0x28 = record {
  fld_00 : bytestring &length=8; # Bot ID
  pad : bytestring &restofdata; # Padding
};
```

Bibliography

[1] Adobe Photoshop. http://www.adobe.com/products/photoshop/.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6), June 1990.

[3] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, March 2008.

[4] Apache Web server. http://httpd.apache.org.

[5] Apache bug 13986, October 2002. https://issues.apache.org/bugzilla/show_bug.cgi?id=13986.

[6] O. Arkin. A remote active OS fingerprinting tool using ICMP. *;login: The USENIX Magazine*, 27(2), November 2008.

[7] Autodesk. http://autodesk.com.

[8] G. Balakrishnan. *WYSINWYX: WHat YOu SEe IS NOtWHat YOu EXEcute*. PhD thesis, Computer Science Department, University of Wisconsin at Madison, Madison, WI, August 2007.

[9] A. Barth, J. Caballero, and D. Song. Secure content sniffing for Web browsers *or* how to stop papers from reviewing themselves. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[10] M. A. Beddoe. Network protocol analysis using bioinformatics algorithms. http://www.4tphi.net/ awalters/PI/PI.html.

[11] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[12] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. RFC 3986 (Standard), January 2005.

[13] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and

J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Ottawa, Canada, June 2006.

[14] Bind. `http://www.isc.org/software/bind`.

[15] N. Bjorner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, York, United Kingdom, March 2009.

[16] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, March 2008.

[17] N. Borisov, D. J. Brumley, H. J. Wang, and C. Guo. Generic application-level protocol analyzer and its language. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2007.

[18] S. Bradner. The Internet standards process – revision 3. RFC 2026 (Best Current Practice), October 1996.

[19] D. Brumley. *Analysis and Defense of Vulnerabilities in Binary Code*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 2008.

[20] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the USENIX Security Symposium*, Boston, MA, August 2007.

[21] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[22] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[23] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest pre-conditions. In *Proceedings of the IEEE Computer Security Foundations Symposium*, Venice, Italy, July 2007.

[24] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2010.

[25] J. Caballero, Z. Liang, P. Poosankam, and D. Song. Towards generating high coverage

vulnerability-based signatures with protocol-level constraint-guided exploration. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, Saint-Malo, France, September 2009.

[26] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the ACM Conference on Computer and Communications Security*, Chicago, IL, November 2009.

[27] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the ACM Conference on Computer and Communications Security*, Chicago, IL, October 2010.

[28] J. Caballero and D. Song. Rosetta: Extracting protocol semantics using binary analysis with applications to protocol replay and NAT rewriting. Technical Report CMU-CyLab-07-014, Cylab, Carnegie Mellon University, Pittsburgh, PA, October 2007.

[29] J. Caballero, S. Venkataraman, P. Poosankam, M. G. Kang, D. Song, and A. Blum. FiG: Automatic fingerprint generation. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2007.

[30] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2007.

[31] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Symposium on Operating System Design and Implementation*, San Diego, CA, December 2008.

[32] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the International SPIN Workshop*, San Francisco, CA, August 2005.

[33] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2006.

[34] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, Paris, France, July 2008.

[35] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of the International Conference on Software Engineering*, Portland, OR, May 2003.

[36] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of soft-

ware. In *Proceedings of the ACM Conference on Computer and Communications Security*, Washington, D.C., November 2002.

[37] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the International Conference on Dependable Systems and Networks*, Anchorage, AK, June 2008.

[38] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*, San Diego, CA, August 2004.

[39] C. Cifuentes and M. V. Emmerik. Recovery of jump table case statements from binary code. In *Proceedings of the International Workshop on Program Comprehension*, Pittsburgh, PA, May 1999.

[40] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[41] Code virtualizer. `http://www.oreans.com/codevirtualizer.php`.

[42] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the Symposium on Principles of Programming Languages*, San Diego, CA, January 1998.

[43] D. E. Comer and J. C. Lin. Probing TCP implementations. In *Proceedings of the USENIX Summer Technical Conference*, Boston, MA, June 1994.

[44] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[45] L. Corrons. Mariposa botnet, March 2010. `http://pandalabs.pandasecurity.com/mariposa-botnet/`.

[46] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of the Symposium on Operating Systems Principles*, Bretton Woods, NH, October 2007.

[47] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of the Symposium on Operating Systems Principles*, Brighton, United Kingdom, October 2005.

[48] M. Cova, V. Felmetsger, D. Balzarotti, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[49] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *ACM Transactions on Architecture and Code Optimization*, 3(4), December 2006.

[50] D. Crocker and P. Overell. Augmented BNF for syntax specifications: ABNF. RFC 4234 (Draft Standard), October 2005.

[51] B. Crothers. One million PCs shipped daily, May 2010. `http://news.cnet.com/8301-13924_3-20004677-64.html`.

[52] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol description generation from network traces. In *Proceedings of the USENIX Security Symposium*, Boston, MA, August 2007.

[53] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2006.

[54] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2008.

[55] W. Cui, M. Peinado, H. J. Wang, and M. Locasto. ShieldGen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2007.

[56] CVE. Common vulnerabilities and exposures. `http://cve.mitre.org/cve/`.

[57] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - the Advanced Encryption Standard*. Springer, Heidelberg, Germany, March 2002.

[58] D. Danchev. Someone hijacked my botnet!, February 2009. `http://ddanchev.blogspot.com/2009/02/help-someone-hijacked-my-100k-zeus.html`.

[59] D. De, A. Kumarasubramanian, and R. Venkatesan. Inversion attacks on secure hash functions using SAT solvers. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, Lisbon, Portugal, May 2007.

[60] F. Desclaux and K. Kortchinsky. Vanilla Skype part 1. In *Recon*, Montréal, Canada, June 2006.

[61] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8), August 1975.

[62] D. Dittrich, F. Leder, and T. Werner. A case study in ethical decision making regarding remote mitigation of botnets. In *Proceedings of the Workshop on Ethics in Computer Security*

Research, Tenerife, Spain, January 2010.

[63] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4), July 1984.

[64] J. Eriksson. Giving hackers a taste of their own medicine. In *RSA Conference*, April 2008.

[65] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616 (Draft Standard), June 1999.

[66] File tool. `http://darwinsys.com/file/`.

[67] FileZilla. `http://filezilla-project.org/`.

[68] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[69] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. V. Doorn. Remote detection of virtual machine monitors with fuzzy benchmarking. *SIGOPS Operating Systems Review*, 42(3), April 2008.

[70] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) part one: Format of Internet message bodies. RFC 2045 (Draft Standard), November 1996.

[71] N. Freed and N. Borenstein. Multipurpose Internet mail extensions (MIME) part two: Media types. RFC 2046 (Draft Standard), November 1996.

[72] A. Fritzler. The unofficial AIM/OSCAR protocol specification, April 2008. `http://www.oilcan.org/oscar/`.

[73] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the Computer Aided Verification*, Berlin, Germany, August 2007.

[74] V. Ganesh, T. Leek, and M. C. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[75] GCC inline assembly, March 2003. `http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html`.

[76] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the Symposium on Principles of Programming Languages*, Nice, France, January 2007.

[77] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, AZ, June 2008.

[78] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceed-*

ings of the SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, June 2005.

[79] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[80] A. Gohr and C. Smith. Internet Explorer facilitates XSS, February 2007. `http://www.splitbrain.org/blog/2007-02/12-internet_explorer_facilitates_cross_site_scripting`.

[81] D. W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, ND, June 1997.

[82] I. Guilfanov. A simple type system for program reengineering. In *Proceedings of the Working Conference on Reverse Engineering*, Stuttgart, Germany, October 2001.

[83] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2009.

[84] P. Haffner, S. Sen, O. Spatscheck, and D. Wang. ACAS: Automated construction of application signatures. In *Proceedings of the ACM Workshop on Mining network data*, Philadelphia, PA, October 2005.

[85] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009.

[86] J. Hope. SMF upload XSS vulnerability, December 2009. `http://seclists.org/fulldisclosure/2006/Dec/0079.html`.

[87] HotCRP conference management software. `http://www.cs.ucla.edu/~kohler/hotcrp/`.

[88] ICQ. `http://www.icq.com`.

[89] icqlib: The ICQ library. `http://freshmeat.net/projects/icqlib/`.

[90] IDA Pro disassembler and debugger. `http://www.hex-rays.com/idapro/`.

[91] Intel. Intel64 and IA-32 architectures software developer's manuals. `http://www.intel.com/products/processor/manuals/`.

[92] ISIC: IP stack integrity checker. `http://isic.sourceforge.net/`.

[93] ISO/IEC. The ISO/IEC 9899:1999 C programming language standard, May 2005.

[94] Jbrofuzz. http://sourceforge.net/projects/jbrofuzz/.

[95] JEIDA. The EXIF standard version 2.3, April 2010.

[96] S. Jha, N. Li, M. V. Tripunitara, Q. Wang, and W. H. Winsborough. Towards formal verification of role-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 5(4), Oct.-Dec. 2008.

[97] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, Chicago, IL, July 2009.

[98] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the International World Wide Web Conference*, Banff, Canada, May 2007.

[99] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying spamming botnets using Botlab. In *Proceedings of the Symposium on Networked System Design and Implementation*, Boston, MA, April 2009.

[100] R. Kaksonen. *A Functional Method for Assessing Protocol Implementation Security*. PhD thesis, Technical Research Centre of Finland, Espoo, Finland, 2001.

[101] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. In *Proceedings of the Workshop on Virtual Machine Security*, Chicago, IL, November 2009.

[102] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-automated discovery of application session structure. In *Proceedings of the Internet Measurement Conference*, Rio de Janeiro, Brazil, October 2006.

[103] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *Proceedings of the International Symposium on Software Testing and Analysis*, Chicago, IL, July 2009.

[104] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[105] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the USENIX Security Symposium*, San Diego, CA, August 2004.

[106] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[107] M. Kobayashi. Dynamic characteristics of loops. *IEEE Transactions in Computers*, 33(2), 1984.

[108] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the USENIX Security Symposium*, Montréal, Canada, August 2009.

[109] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[110] B. Korel. A dynamic approach of test data generation. In *Proceedings of the IEEE International Conference on Software Maintenance*, San Diego, CA, November 1990.

[111] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3), October 1988.

[112] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of the Workshop on Hot Topics in Networks*, Boston, MA, November 2003.

[113] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the USENIX Security Symposium*, Baltimore, MD, July 2005.

[114] A. Lanzi, M. Sharif, and W. Lee. K-Tracer: A system for extracting kernel malware behavior. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2009.

[115] E. Lawrence. IE8 security V, July 2008. http://blogs.msdn.com/ie/archive/2008/07/02/ie8-security-part-v-comprehensive-protection.aspx.

[116] C. Leita, M. Dacier, and F. Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, Hamburg, Germany, September 2006.

[117] C. Leita, K. Mermoud, and M. Dacier. ScriptGen: An automated script generation tool for honeyd. In *Proceedings of the Annual Computer Security Applications Conference*, Tucson, AZ, December 2005.

[118] Z. Li, M. Sanghi, B. Chavez, Y. Chen, and M.-Y. Kao. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[119] Z. Liang and R. Sekar. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In *Proceedings of the Annual Computer Security Applications Conference*, Tucson, AZ, December 2005.

[120] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the ACM Conference on Computer and Communications Security*, Alexandria, VA, November 2005.

[121] Libyahoo2: A C library for Yahoo! Messenger. `http://libyahoo2.sourceforge.net`.

[122] M. H. Ligh and G. Sinclair. Kraken encryption algorithm, April 2004. `http://mnin.blogspot.com/2008/04/kraken-encryption-algorithm.html`.

[123] J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *Proceedings of the Working Conference on Reverse Engineering*, Benevento, Italy, October 2006.

[124] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[125] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Atlanta, GA, November 2008.

[126] Z. Lin, X. Zhang, and D. Xu. Reuse-oriented camouflaging trojan: Vulnerability detection and attack construction. In *Proceedings of the International Conference on Dependable Systems and Networks*, Chicago, IL, June 2010.

[127] M. E. Locasto, K. Wang, A. D. Keromytis, and S. J. Stolfo. FLIPS: Hybrid adaptive intrustion prevention. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, Seattle, WA, September 2005.

[128] N. Lutz. Towards revealing attacker's intent by automatically decrypting network traffic. Master's thesis, ETH, Zürich, Switzerland, July 2008.

[129] M86 Security Labs. Megad analysis, March 2009. `http://www.m86security.com/labs/spambotitem.asp?article=896`.

[130] M86 Security Labs. Security threats: Email and Web threats, January 2009. `http://www.marshal.com/newsimages/trace/Marshal8e6_TRACE_Report_Jan2009.pdf`.

[131] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected means of protocol inference. In *Proceedings of the Internet Measurement Conference*, Rio de Janeiro,

Brazil, October 2006.

[132] Mangleme. http://freshmeat.net/projects/mangleme/.

[133] M. Martin and M. S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the USENIX Security Symposium*, San Jose, CA, July 2008.

[134] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the USENIX Security Symposium*, Vancouver, Canada, July 2006.

[135] Mediawiki. http://www.mediawiki.org/.

[136] Mediawiki: Sites using MediaWiki/en. http://www.mediawiki.org/wiki/Sites_using_MediaWiki/en.

[137] Metasploit. http://metasploit.org.

[138] Microsoft. MSDN: The Microsoft developer network. http://msdn.microsoft.com.

[139] Microsoft. Open specifications. http://www.microsoft.com/openspecifications.

[140] Microsoft. Windows live messenger. http://messenger.msn.com.

[141] Microsoft MSDN. FindMimeFromData function. http://msdn.microsoft.com/en-us/library/ms775107(VS.85).aspx.

[142] Microsoft MSDN. Inline assembler. http://msdn.microsoft.com/en-us/library/4ks26t93.aspx.

[143] Microsoft MSDN. MIME type detection in Internet Explorer. http://msdn.microsoft.com/en-us/library/ms775147.aspx.

[144] Microsoft Security Response Center. A dumb patch?, May 2005. http://blogs.technet.com/msrc/archive/2005/10/31/413402.aspx.

[145] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[146] D. Mills. Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI. RFC 4330 (Informational), January 2006.

[147] MiniWeb: The open-source mini HTTP daemon. http://miniweb.sourceforge.net.

[148] M. Mintz and A. Sayers. Unofficial guide to the MSN messenger protocol, December 2003. http://www.hypothetic.org/docs/msn/.

[149] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987.

[150] K. Moore. MIME (Multipurpose Internet Mail Extensions) part three: Message header extensions for non-ascii text. RFC 2047 (Draft Standard), November 1996.

[151] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[152] A. Mushtaq. Smashing the Mega-d/Ozdok botnet in 24 hours, November 2009. `http://blog.fireeye.com/research/2009/11/smashing-the-ozdok.html`.

[153] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of the Symposium on Networked System Design and Implementation*, San Francisco, CA, March 2004.

[154] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Symposium on Operating System Design and Implementation*, Boston, MA, December 2002.

[155] Y. Nadji, P. Saxena, and D. Song. Document structure integrity : A robust basis for XSS defense. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2009.

[156] S. Nanda, W. Li, L.-C. Lam, and T. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *Proceedings of the International Symposium on Code Generation and Optimization*, New York, NY, March 2006.

[157] Nettime. `http://nettime.sourceforge.net`.

[158] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2006.

[159] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[160] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2005.

[161] NIST. Federal information processing standard 180-2: Secure hash standard, August 2002.

[162] Nmap. `http://www.insecure.org`.

[163] NTinternals: The undocumented Microsoft Windows functions. `http://`

undocumented.ntinternals.net/.

[164] Ntpd: Network time protocol daemon. http://www.eecis.udel.edu/~mills/ntp/html/ntpd.html.

[165] OpenSSL: The open source toolkit for SSL/TLS. http://www.openssl.org/.

[166] OSVDB. Cutwail Bot svchost.exe CC Message Handling Remote Overflow, July 2010. http://osvdb.org/66497.

[167] OSVDB. Gheg Bot RtlAllocateHeap Function Null Dereference Remote DoS, July 2010. http://osvdb.org/66498.

[168] OSVDB. Zbot Trojan svchost.exe Compressed Input Handling Remote Overflow, July 2010. http://osvdb.org/66501.

[169] OSVDB. Zbot Trojan svchost.exe Network Message Crafted Payload Size Handling Infinite Loop Remote DoS, July 2010. http://osvdb.org/66500.

[170] OSVDB. Zbot Trojan svchost.exe RtlAllocateHeap Function Null Dereference Remote DoS, July 2010. http://osvdb.org/66499.

[171] W. A. Owens, K. W. Dam, and H. S. Lin. *Technology, Policy, Law, and Ethics Regarding U.S. Acquisition and Use of Cyberattack Capabilities*. The National Academies Press, Washington, DC, USA, 2009.

[172] W. Palant. The hazards of MIME sniffing, April 2007. http://adblockplus.org/blog/the-hazards-of-mime-sniffing.

[173] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for writing application protocol parsers. In *Proceedings of the Internet Measurement Conference*, Rio de Janeiro, Brazil, October 2006.

[174] V. Paxson. Automated packet trace analysis of TCP implementations. In *Proceedings of the SIGCOMM Conference*, Cannes, France, September 1997.

[175] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24), 1999.

[176] Peach fuzzing platform. http://peachfuzzer.com/.

[177] Pidgin. http://www.pidgin.im/.

[178] C. Pierce. Owning Kraken zombies, a detailed dissection, April 2008. http://dvlabs.tippingpoint.com/blog/2008/04/28/owning-kraken-zombies.

[179] B. Potter, Beetle, CowboyM, D. Moniz, R. Thayer, 3ricj, and Pablos. Shmoo-fu: Hacker goo, goofs, and gear with the shmoo. In *DEFCON*, Las Vegas, ND, July 2005.

[180] QEMU: Open source processor emulator. http://wiki.qemu.org.

[181] Queso. http://ftp.cerias.purdue.edu/pub/tools/unix/scanners/queso.

[182] P. Ringnalda. Getting around Internet Explorer MIME type mangling, April 2004. http://weblog.philringnalda.com/2004/04/06/getting-around-ies-mime-type-mangling.

[183] R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.

[184] M. Round. The downside of uploads, February 2008. http://www.malevolent.com/weblog/archive/2008/02/26/uploads-mime-sniffing/.

[185] S. Rubin, S. Jha, and B. P. Miller. Automatic generation and analysis of NIDS attacks. In *Proceedings of the Annual Computer Security Applications Conference*, Tucson, AZ, December 2004.

[186] Samba. http://samba.org.

[187] Savant Web server. http://savant.sourceforge.net.

[188] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, Chicago, IL, July 2009.

[189] B. Schwartz, S. Debray, G. Andrews, and M. Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proceedings of the Workshop on Binary Translation*, Barcelona, Spain, September 2001.

[190] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the European Software Engineering Conference*, Lisbon, Portugal, September 2005.

[191] Shadowserver foundation. http://www.shadowserver.org/.

[192] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[193] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[194] S. Shevchenko. Kraken is finally cracked, April 2008. http://blog.threatexpert.

com/2008/04/kraken-is-finally-cracked.html.

[195] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.

[196] G. Shiffler. Gartner forecast: PC installed base, worldwide, 2004-2012, April 2008.

[197] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the Symposium on Operating System Design and Implementation*, San Francisco, CA, December 2004.

[198] Skype. http://www.skype.com.

[199] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information Systems Security*, Hyderabad, India, December 2008. Keynote invited paper.

[200] Spike. http://www.immunitysec.com/resources-freesoftware.shtml.

[201] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying loops using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 18(6), 1996.

[202] K. Stevens and D. Jackson. Zeus banking trojan report, March 2010. http://www.secureworks.com/research/threats/zeus/.

[203] W. R. Stevens. *TCP/IP Illustrated*, *Volume 1: The Protocols*. Addison-Wesley, 1994.

[204] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 2004.

[205] Symantec. Security threat report: Trends for 2009, April 2010.

[206] P. Szor. *The Art of Computer Virus Research and Defense*. Symantec Press, 2005.

[207] Themida. http://www.oreans.com/themida.php.

[208] TinyICQ. http://www.downv.com/Windows/download-Tiny-ICQ-10064167.htm.

[209] Titanengine. http://www.reversinglabs.com/products/TitanEngine.php.

[210] A. Tridgell. How Samba was written, August 2003. http://samba.org/ftp/tridge/misc/french_cafe.txt.

[211] O. Udrea, C. Lumezanu, and J. S. Foster. Rule-based static analysis of network protocol implementations. In *Proceedings of the USENIX Security Symposium*, Vancouver, Canada, July 2006.

[212] Unpckarc: The unpacker archive. `http://www.woodmann.com/crackz/Tools/Unpckarc.zip`.

[213] S. Venkataraman, A. Blum, and D. Song. Limits of learning-based signature generation with adversaries. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[214] G. Vigna, W. Robertson, and D. Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the ACM Conference on Computer and Communications Security*, Washington, D.C., October 2004.

[215] Vmprotect. `http://www.vmprotect.ru/`.

[216] I. V.Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Proceedings of the USENIX Security Symposium*, Boston, MA, August 2007.

[217] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating Systems Principles*, Asheville, NC, October 1993.

[218] H. J. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the SIGCOMM Conference*, Portland, OR, August 2004.

[219] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[220] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet vaccine: Black-box exploit detection and signature generation. In *Proceedings of the ACM Conference on Computer and Communications Security*, Alexandria, VA, October 2006.

[221] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. ReFormat: Automatic reverse engineering of encrypted messages. In *Proceedings of the European Symposium on Research in Computer Security*, Saint-Malo, France, September 2009.

[222] The WebKit open source project. `http://webkit.org`.

[223] M. Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering*, San Diego, CA, March 1981.

[224] Wikipedia. http://www.wikipedia.org.

[225] Wikipedia: Image use policy. http://en.wikipedia.org/wiki/Image_use_policy.

[226] Wine. http://www.winehq.org/.

[227] Wireshark. http://www.wireshark.org/.

[228] Wireshark: Fuzz testing tools. http://wiki.wireshark.org/FuzzTesting.

[229] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2008.

[230] XED: X86 encoder decoder. http://www.pintool.org.

[231] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the Symposium on Principles of Programming Languages*, Long Beach, CA, January 2005.

[232] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, June 2008.

[233] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*, Alexandria, VA, November 2005.

[234] R. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the International Symposium on Software Testing and Analysis*, Seattle, WA, July 2008.

[235] Yahoo! messenger. http://messenger.yahoo.com.

[236] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[237] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *Proceedings of the USENIX Security Symposium*, Baltimore, MD, July 2005.

[238] H. Yin and D. Song. Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, CA, January 2010.

[239] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of*

the International Conference on Software Engineering, Portland, OR, May 2003.

[240] The zlib library. `http://www.zlib.net/`.