Rosetta: Extracting Protocol Semantics using Binary Analysis with Applications to Protocol Replay and NAT Rewriting

Juan Caballero_, Dawn Song†_
_Carnegie Mellon University †UC Berkeley
jcaballero@cmu.edu dawnsong@cs.berkeley.edu

October 9, 2007
CMU-CyLab-07-014

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Rosetta: Extracting Protocol Semantics using Binary Analysis with Applications to Protocol Replay and NAT Rewriting

Juan Caballero*, Dawn Song†*
*Carnegie Mellon University †UC Berkeley
jcaballero@cmu.edu dawnsong@cs.berkeley.edu

Abstract

Rewriting a previously seen dialog between two entities, so that it is accepted by another entity, is important for many applications including: the *protocol replay* problem and the *NAT rewriting* problem. Both problems are instances of a larger problem that we call the *dialog rewriting* problem. The challenge in dialog rewriting is that the dynamic fields, e.g., hostnames, IP addresses, session identifiers or timestamps, in the original dialog need to be rewritten for the modified dialog to succeed. This is particularly difficult because the protocol used in the original dialog might be unknown.

In this paper, our goal is to generate a transformation function that can be used to rewrite the values of the dynamic fields. For this, we propose binary analysis techniques to solve the main two challenges: 1) how to automatically identify the dynamic fields, and 2) how to automatically rewrite the values in the dynamic fields.

We have implemented Rosetta, a system that creates the transformation function using our proposed techniques. Our results show that we are able to identify different types of dynamic fields present in commonly used protocols such as FTP, DNS and ICQ , and that we are able to rewrite the values in the dynamic fields, even when those fields use complex encodings to represent the data, thus enabling the protocol replay and NAT rewriting problems.

1 Introduction

We call *dialog rewriting* the problem of rewriting a previously seen *original dialog* betwen two entities, so that it is accepted by another entity that was not involved in the original dialog. Dialog rewriting is an important problem for many applications. For example, dialog rewriting is important for Network Address Translation (NAT) [14, 16], where protocols include ports and IP addresses in the payload, which need to be rewritten for the communication to succeed. Other examples where dialog rewriting is important are when demonstrating software vulnerabilities to third parties by replaying exploits that might be hidden in the middle of complex dialogs [19], and when building application layer gateways to scrub confidential data being leaked out of an enterprise network [27].

The main challenge in dialog rewriting is that an exact replay of the original dialog does not work, because the protocol messages often include *dynamic fields*. Dynamic fields are fields that contain data, which might change for different sessions or different hosts, such as hostnames, IP addresses, ports, checksums, and timestamps Intuitively, these dynamic fields need to be identified and rewritten in the *modified dialog*, otherwise the modified dialog will likely fail. This problem includes what previous work has called the *application dialog replay* [11, 25].

The difficulty in automatically rewriting the dynamic fields is figuring out what needs to be rewritten, and how it should be rewritten. In particular, we need to figure out what type of data to write in them and what encoding to use for that type of data. This is difficult because many applications use closed protocols, with no publicly available specification, so the type of data and encoding being used in the fields are unknown. Even when the protocol is known, the problem is important because a manual approach is tedious and because there are times when the implementations actually differ from the protocol specification.

We divide the problem of dialog rewriting into two stages: the offline *analysis* stage and the online *editing* stage. In the *analysis stage* the problem is figuring out how to rewrite the dynamic fields. To solve this problem we need to generate the *transformation function*, which contains all the information about how to rewrite the dynamic fields with new values. In the *editing* stage, the problem is how to locate the dynamic fields in a message. Once located, the *rewriter*, i.e., the entity that performs the dialog rewriting, just needs to apply the transformation function with the new values for the dynamic fields.

The focus of this paper is on the analysis stage. Thus, our problem is how to create the transformation function. The transformation function needs to contain information about how to solve two problems: 1) how to automatically

identify the dynamic fields, which includes how to locate them in the dialog and how to infer the type of data they contain, and 2) how to automatically rewrite the values in the dynamic fields, which includes understanding how the data is encoded in the dynamic field.

In this paper we propose a novel approach, which employs binary analysis techniques to solve those two problems during the analysis stage. Then, we show how to use those techniques to create the transformation function that the rewriter will use to rewrite the dynamic fields. Our techniques are based on analyzing program binaries implementing the protocol. Binary analysis is specially useful when dealing with closed applications, since the application's source code is usually not available. The advantage of using program binaries compared to network traces is that network traces only contain syntactic information while the program binaries contain a wealth of information about the protocol semantics, because the program binaries contain all the information about the protocol specification.

In addition to the dialog rewriting problem, our techniques are also useful to understand the protocol and to facilitate usage of protocol semantics for other problems such as building application layer proxies, protocol reverse-engineering [7, 11], intrusion detection systems [13], fingerprint generation [6], and protocol analyzers for network monitoring [4, 26].

There has been previous work on application dialog replay, which is one instance of the dialog rewriting problem. Leita et al. [19, 20] and Cui et al. [11, 12] propose to locate dynamic fields in network traces by finding fields that change value across different sessions. In addition, Cui et al [12] have proposed heuristics to identify a few types of dynamic fields in the context of replaying previously captured network traces. The fundamental limitation of both approaches is the lack of protocol semantics in network traces. To overcome the lack of semantics in network traces they need to make assumptions about the type of encoding used in the dynamic fields or rely on statistical variability across different sessions. But these techniques fail to identify many types of dynamic fields. Another line of work [25] has used binary analysis techniques to solve the related, problem of replaying complete program executions. The problem they address is different from ours because our goal is to rewrite the network dialog, not the complete execution.

We use two different binary analysis techniques to identify dynamic fields in unknown protocols: we identify dynamic fields in *received* messages by analyzing how the data in a received message is used, and we identify dynamic fields in *sent* messages by analyzing how the data in a sent message was derived. The intuition behind these two techniques is that the values contained in the dynamic fields are derived from auxiliary data, i.e., data that is not part of the program's source code. This auxiliary data is obtained by the program through system calls to the operating system, and includes the system time, data received from the network, the local hostname or IP address, user keystrokes, or configuration data from a file. For example, one type of dynamic field is keyboard input. We can identify dynamic fields containing keyboard input by analyzing if some parts of the messages being sent are derived from data that the user entered through the keyboard.

To understand how to rewrite the dynamic fields, it is not enough to know the type of data contained in those dynamic fields. In addition, we need to know exactly *how* the value in the dynamic fields was generated. The reason we need such knowledge is the large number of different encodings employed by network protocols. For example, if we know that some bytes in a message represent a port number, but we do not know the exact encoding the field uses to represent the port number, we will not know how to rewrite the old value with a new one.

We have implemented Rosetta, a system that takes as input the program binaries and returns the transformation function that the rewriter uses during the modified dialog to rewrite the values in the dynamic fields. Our results show that we are able to identify different types of dynamic fields present in commonly used protocols such as FTP, DNS and ICQ, and that we are able to rewrite the values of those dynamic fields, even when complex encodings are used to represent the data. Thus, our results show that when combining our techniques to automatically create the transformation function with previously proposed techniques to identify field boundaries, we enable automatic protocol replay and automatic NAT rewriting.

In summary, this paper makes the following contributions:

Propose a binary analysis approach to the dialog rewriting problem: To successfully rewrite a previously seen dialog, the original dialog needs to be rewritten assigning new values to the dynamic fields. In this paper, we present a solution based on binary analysis that allows to generate a transformation function that can be used to rewrite the dynamic fields, in some unknown protocol, so that the modified dialog is accepted.

Propose novel techniques to automatically identify dynamic fields: We propose two novel binary analysis techniques to identify dynamic fields. We identify dynamic fields in *received* messages by analyzing how the data in a received message is used, and we identify dynamic fields in *sent* messages by analyzing how the data in a sent message was derived. Our techniques work by monitoring how the parameters and return values of selected system calls are used by the program and allow to identify multiple types of dynamic fields including: cookies, timestamps, IP addresses, ports, hostnames, file names, consistency fields, keyboard input, and configuration data.

Propose a technique to automatically rewrite the dynamic fields: We propose a technique to automatically construct the transformation function, used to rewrite the dynamic field values. The core of the transformation function are the input-output formulas that precisely capture how the value in a dynamic field was encoded from the output of selected system calls. Our technique allows to rewrite the value of the dynamic fields independently of the complex encodings that might be used. We capture how the data sent by a program is derived from the output of selected system calls using Dynamic Program Slicing, Symbolic Execution, and other techniques from the programming languages community.

Design, implement and evaluate our techniques: We design and implement Rosetta, a system that implements our techniques. We evaluate Rosetta using 5 different binaries and 3 different widely used protocols: FTP, DNS and ICQ. Our results show that we are able to create the transformation function to automatically rewrite an FTP standard mode session in under 30 minutes, and the time to apply the transformation function during the editing phase is negligible. Thus, our results show that we could potentially enable any protocol with similar behavior to work in the presence of a NAT.

The remainder of this paper is organized as follows. Section 2 introduces the problem. Then, in Section 3 we present the system architecture. Next, in Sections 4 and 5 we propose techniques for identifying dynamic fields and for capturing how to rewrite the values in the dynamic fields,. In Section 6 we explain how to obtain the transformation function and we evaluate our system in Section 7. Finally, we describe related work in Section 8 and conclude in Section 9.

2 Overview and Problem Definition

2.1 Overview of the Problem

Rewriting a previously seen dialog betwen two entities, so that it is accepted by another entity, is important many applications including the *protocol replay* problem and the *NAT rewriting* problem. We call the entity that performs the dialog rewriting, the *rewriter*.

In both problems, the challenge is that the *dynamic fields* in the *original dialog* need to be rewritten for the *modified dialog* to succeed. Dynamic fields contain data that might change for different sessions or different hosts, such as hostnames, IP addresses, ports, checksums, and timestamps. Intuitively, these dynamic fields need to be identified and rewritten in the *modified dialog*, otherwise, this modified dialog is likely to fail. The problem is extremely challenging when the protocol is unknown or has no publicly available specification.

In protocol replay we only need to replay one side of

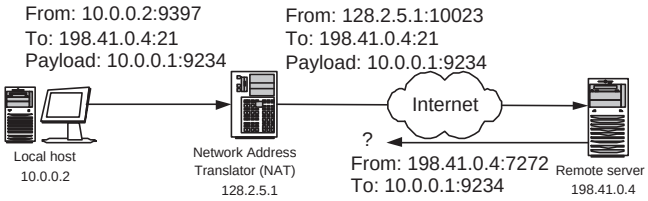

Figure 1: NAT rewriting problem. The connection originated by the remote server cannot reach the local host.

the dialog. For example, when an exploit has been captured using a honeypot and we need to replay it to other hosts running different versions of the same application to verify if they are susceptible to the exploit.

In NAT rewriting we need to modify the dialog between two entities that communicate through some NAT or proxy, using some unknown protocol, Here the rewriter needs to modify both sides of the dialog. For example, imagine the scenario in Figure 1, where a local network connects to the Internet through a NAT that only allows outbound connections to go through, i.e., connections that originated from the local network. A host in the local network is connected to some remote server using a command channel and wants to start a separate data transfer using a new connection, as in FTP in standard mode. The host selects a new port to listen to and sends this newly selected port and its IP address, to the remote server, who needs to start a new connection back to the client using the IP address and port provided by the host.

For the data channel to be successfully established, the NAT needs to be able to do two things: 1) identify the dynamic fields in the message sent by the host (i.e., the IP address and port) so that it knows an inbound connection is coming, 2) to rewrite the dynamic fields with new values. The IP address needs to be rewritten so that it becomes the externally routable address of the NAT and the port needs to be rewritten if it is already being used by the NAT for another connection.

We consider protocol replay and NAT rewriting two different instances of the same problem that we call the *dialog rewriting* problem. We divide the dialog rewriting problem into two stages: the offline *analysis* stage and the online *editing* stage. In the analysis stage, the goal is to generate the *transformation function* that details how to rewrite the dynamic fields. This stage happens offline. Then, in the editing stage, the rewriter uses the transformation function to rewrite the dynamic fields so that the modified dialog can succeed. This stage happens online.

Thus, in dialog rewriting, we need to address two different problems: 1) how to automatically identify the dynamic fields, and 2) how to automatically rewrite the values of the dynamic fields.

Scope of the problem: We can further subdivide the

problem into tasks. During the analysis stage, there are three tasks: to identify the field boundaries, to identify which of those fields are dynamic, and to generate the transformation function that allows to rewrite the values of the dynamic fields. During the editing stage, the rewriter has two tasks: identify the field boundaries, and apply the transformation function to rewrite the dynamic fields with new values.

Both stages need to identify, as a first step, the field boundaries. There have been proposed solutions for the problem of identifying the field boundaries in protocol messages [7, 11, 19]. In this paper, for the sake of scope, we assume that when we need the field boundaries, they can be provided by one of the previous solutions and focus on the remaining tasks, which are challenging enough in themselves.

Besides identifying the field boundaries, the remaining task on the editing stage is for the rewriter to apply the transformation function with the new values for the dynamic fields. Clearly, the challenge is how to obtain such transformation function. Once obtained, applying it is straightforward. Thus, in this paper, we focus on the last two tasks of the analysis stage, which are the most challenging: how to identify the dynamic fields and how to rewrite the values of those dynamic fields. We have built a system called Rosetta that implements those two tasks.

2.2 Problem Definition

We define the dialog rewriting problem as follows. Two entities, called the *initiator* and the *respondent*, engage in some communication over the network using some unknown protocol. We term such communication the *original dialog* and such dialog represents an instance of an application session.

The goal of dialog rewriting is to produce a *modified dialog*, which is accepted by an external entitity, called the *verifier*, which was not involved in the original dialog. The modified dialog occurs between the verifier and a fourth entity that we call the *rewriter*. Note that, in the modified dialog, the rewriter can rewrite the initiator's side of the dialog, or the respondent's side of the dialog (e.g., in protocol replay), or both of them (e.g., in NAT rewriting).

Problem definition: Our problem is, given the program binaries of both the initiator and the respondent, to generate a *transformation function*, which can be used by the rewriter to successfully communicate with the verifier during the modified dialog. Such transformation function is based on the original dialog and states which parts of the original dialog need to be modified by the rewriter, so that the verifier accepts the modified dialog.

2.3 Approach

To obtain the transformation function needed by the rewriter, we have to solve two main problems: 1) identify which fields in the original dialog are dynamic and what type of data they contain, and 2) learn how to rewrite the dynamic fields values.

Identifying the dynamic fields: The intuition we use to identify the dynamic fields is the following: if we understand either how a dynamic field is used or how the value of a dynamic field is derived, then we can usually infer what type of dynamic field it is, that is, the semantics of the dynamic field. Using this intution we propose two different techniques to identify dynamic fields, which depend on the direction of the message: 1) for messages that are *received* by the progam, we analyze how the data in the message is used, and 2) for messages that are *sent* by the program, we analyze how the data in the message was derived.

For received messages, we observe if the data contained in the received message is used in the *parameters* of some selected system calls. Since the goal of the system call and the meaning of its parameters is usually well defined in some specification (e.g., the Windows API documentation), then when previously received data is used in those parameters we can infer its semantics. For example, when the parameters of a system call to start a new connection (e.g., connnect), which are defined to be an IP address and a port, have been derived from some parts of a previously received message, then we can infer that those parts of the received message encoded an IP address and a port.

For sent messages, we capture how the sent data was constructed using the *return values*[1] of selected system calls. Since we know what the values returned by a system call mean, then we can infer the semantics of data that has been derived from those values. For example, one type of dynamic field is keyboard input. We can identify dynamic fields containing keyboard input by analyzing if some parts of the messages being sent are derived from data that the user entered through the keyboard.

Understanding how to rewrite the dynamic fields: To rewrite the dynamic fields it is not enough to know what type of data is stored in the dynamic field. We also need to understand how that type of data needs to be encoded into the dynamic field. For example we might know that a dynamic field contains time data but there exist many different encodings for time. For example, according to the ISO 8601 standard [1], October 4th, 2007 could be represent using months and days as 2007-10-04, but it could be represented using weeks as 2007-W40-4. Clearly, when rewriting the value of a dynamic field with new data, we need to know which format to use, so that the value is not

[1] By return values, we mean all ouput values of a system call, not just the return value usually place in eax. Here, we try to avoid overloading the term output.

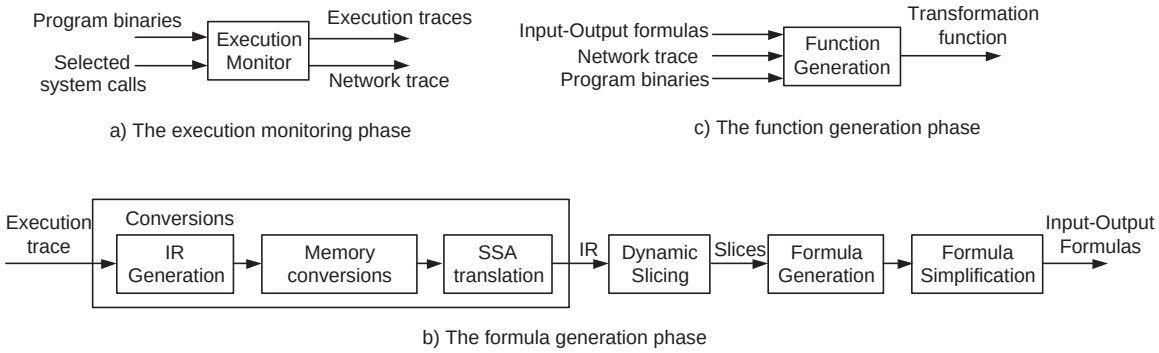

Figure 2: System Architecture for Rosetta.

rejected.

To understand how to rewrite the dynamic fields, we need to precisely capture how the values in the dynamic field were generated. Thus, we are interested in understanding how the different bytes in a message being sent have been derived from the output of selected system calls To address this problem we use data dependency techniques from the programming languages community. In particular, we use Dynamic Program Slicing [3] and Symbolic Execution, which extracts, for a execution history and a variable, an slice that contains all the statements in the execution history that affected the value of that variable. In our case, the variables are the different bytes sent over the network, and the slices contain only the information about how those bytes were generated. Then, we use those slices to generate input-output formulas that are semantically equivalent to the slices, but much smaller in size.

The input-output formulas precisely capture, how the value of the dynamic field was generated in that execution. Even if the input-output formulas only encode a single execution, the fact that fields do not usually hold two different types of values, or two different encodings of the same value, means that these input-output formulas can normally be used to replay different values. There is one exception though with variable-length dynamic fields. Our input-output formulas only allow byte-by-byte rewriting of the dynamic fields. But sometimes, the new value that we want to encode into a variable-length dynamic field might have a larger length than the original value did. In this case, such byte-by-byte rewriting will fail. Thus, for these cases we further analyze the program binary using static analysis techniques, to identify the part of the code that performs the encoding, and add this knowledge to the transformation function.

3 System Architecture

Our system, Rosetta, implements the last two steps in the offline analysis stage: identifying the dynamic fields and understanding how the values in those dynamic fields are generated. Figure 2 shows the system architecture for Rosetta. It is comprised of three phases: the *execution monitoring* phase, the *formula generation* phase, and the *function generation* phase.

The execution monitoring phase takes as input the program binaries of the initiator and the respondent, and the set of selected system calls. It executes both binaries inside the execution monitor, while they generate the original dialog. The output of the execution monitoring phase are the execution traces that capture all the processing done by each of the binaries during the dialog, and the network trace that contains all messages exchanged during the dialog. We present the execution monitoring phase in section 3.1.

The formula generation phase is the core of the system. It is repeated for each message in the network trace in the same order that they were sent. It takes as input the execution trace of the originator of the current message being processed and outputs an input-output formula that captures how that message was generated from the output of the selected system calls. We present the formula generation phase in section 5.

The function generation phase takes as input the input-output formulas generated during the formula generation phase, the program binaries, and the network trace, and produces the transformation function, which will be used by the rewriter during the editing stage. We present the function generation phase in section 6.

3.1 The Execution Monitoring phase

In this section we briefly describe the execution monitor that we use to monitor the program binaries of the initiator and the respondent during the execution of the original

dialog. In addition we describe the different groups of system calls that we currently suport.

The execution monitor: We implement the execution monitor using an emulator [2]. Executing a program inside an emulator allows us to monitor the internal execution of the program and the input/output operations it performs. The execution trace generated by the execution monitor contains all the instructions executed by the program or any library that the program uses, including system libraries and dynamically loaded libraries such as dll's. In addition to the instructions themselves, the emulator also collects, for each instruction, the content of the operands at the time the instruction is executed.

Selected system calls: To detect the dynamic fields we monitor the parameters and return values of selected system calls. We note to the reader that we are lax in our use of the term system calls throughout this paper, since we denote calls to the Windows API as system calls where they are really only an interface to the real system calls. The specific system calls to be monitored can be defined by the user. We currently provide support for six large groups of systems calls: File system calls to create files, open files or read file information; Network system calls to receive data, open sockets, bind ports, and start connections; Keyboard system calls to receive data written through keyboard; Time system calls to get the system or local time; Registry system calls to open Windows registry keys, or read and write the values; and Host identifier system calls that return data that identifies the host, such as the IP address or the hostname.

4 Identifying the Dynamic Fields

In this section we introduce the different dynamic fields that we want to identify, and how to identify them. We use two different techniques to identify dynamic fields: 1) we identify dynamic fields in *received* messages by analyzing how the data in a received message is used, and 2) we identify dynamic fields in *sent* messages by analyzing how the data in a sent message was derived.

For received messages, we observe if the data contained in the received message is used in the *parameters* of some selected system calls. Since the goal of the system call and the meaning of its parameters is usually well defined in some specification (e.g., the Windows API documentation), then when received data is used in those parameters we can infer its semantics. For sent messages, we capture how the sent data was constructed using the *return values* of selected system calls. Since we know what the values returned by a system call mean, then we can infer the semantics of the sent data that has been derived from those values.

Next, we present the different dynamic fields we consider, and which of these two techniques we use to identify them.

Direction fields: Direction fields are fields that store information about the location of another field in the message, called the target field. The most common direction fields are length fields, whose value encodes the length of a target field. Other types of direction fields are pointer fields and counter fields. There have been previously proposed techniques to identify the direction fields in a message [7, 11]. In this paper, we use some of those techniques, which identify direction fields by monitoring how the binary parses a received message [7].

Timestamps: Timestamps are fields that contain time data. We identify timestamps by analyzing if some parts of a message being sent by the program have been derived from the output of system calls that request the local or system time (e.g., Windows GetLocalTime and GetSystemTime).

Cookies: Cookies are data present in messages in both directions of the communication, such as session identifiers or data echoed back to the sender. We identify cookies by analyzing if some parts of a message being sent by the program have been derived from data previously received over the network, which includes the case when the data is directly copied and also when it is modified.

Local host identifiers: Local host identifiers identify the local host sending the information. Typical host identifiers are the IP address and the hostname. This local information can be obtained from the operating system through system calls (e.g., gethostname), or from configuration data, which we describe later. We identify local host identifiers by analyzing if some data in a message being sent has been derived using the values returned by such system calls.

Connection endpoints: Some protocols, such as FTP or voice over IP protocols, use data channels that are dynamically created using the control channel. The connection endpoint information for these data channels is usually created during run time. The usual transaction is that one of the parties selects the new connection endpoint, usually consisting of the $\langle addres, port \rangle$ pair, and tells the other party to establish a connection back to that endpoint. We identify connection endpoints by analyzing how the parameters of the system calls used by the program to start new connections (e.g., connect), and bind new listening ports (e.g., bind) have been derived from a previously received message. If such parameters (i.e., addresses and ports) depend on some previously received network data, then we can infer that the received network data encoded some connection endpoint.

Consistency fields: Consistency fields are fields used to check if the data has been erronously modified during transmission. Examples of consistency fields are checksum and hash fields. Consistency fields need to be

checked when a message is received and need to be constructed when a message is sent. We identify consistency fields in the following way: if we find that some bytes of a message being sent have dependencies on all other bytes in the same message, or certain fields, then we flag those bytes as a consistency field.

Keyboard input: Messages often include data provided by the user via the keyboard, such as the filename in a FTP download, the domain name in a DNS query or the user name and password in an ICQ login session. We identify keyboard input by analyzing if any part of a sent message has been derived from data obtained from the keyboard.

File names: We identify file names present in received messages, rather than in sent messages as presented in the above paragraph, by analyzing if the parameters of system calls used to open files (e.g., open) or used to get file properties (e.g., NtQueryInformationFile) have been derived from data previously received over the network.

Configuration data: Configuration data is data provided by the user in a non-interactive way. For example, protocol configuration parameters such as the number of times to retry a connection. We identify configuration data by analyzing if some parts of a sent message have been derived from data read from file. As an special case, we also consider if data has been derived from values stored in the Windows registry.

To summarize, in this section we have shown how we can identify dynamic fields using one of two techniques: analyzing how the data in a received message is used to derive the parameters of selected system calls, or analyzing how the data in a sent message has been derived using the return values of selected system calls, where analyzing means extracting the dependencies between those values. In Section 5 we show the techniques we use to extract the dependencies.

5 Formula Generation

In this section we describe the formula generation phase. The formula generation phase generates the input-output formulas, which precisely capture how a value was generated. We can use the formulas to capture two types of dependencies: 1) how the different bytes in a message being sent have been derived from the output of selected system calls, and 2) how the input parameters to selected system calls have been derived from data previously received over the network.

The second case is only used to identify dynamic fields in received messages, as explained in Section 4 but not to understand how the data in the dynamic field was encoded. Since both cases are instances of the same technique, in this section we will only describe the first case. Thus, the dependencies we describe in this section, capture how the bytes in a message being sent have been derived from the output of selected system calls.

The formula generation comprises three steps. First, we convert the execution trace into our intermediate representation (IR) and apply additional SSA and memory conversions to prepare the IR in the format expected by the following steps. Then, we apply a Dynamic Program Slicing algorithm which extracts, for each variable, all statements that affected the value of the variable, where the variables are in this case the bytes sent over the network. These statements form a slice, which is different for each variable and contains all the needed dependencies. Note that, we consider each byte in a message sent over the network independently and each slice contains all statements that affected how a single byte was derived. Finally, we create a formula by combining all the statements in a slice together and simplify such formula using different techniques. The output of this simplification is the input-output formula, which captures how a byte in a sent message was constructed.

5.1 Conversions

5.1.1 IR generation

We convert the instructions logged in the execution trace to an intermediate representation (IR). The advantage of using an intermediate representation is that it allows us to perform subsequent steps over the simpler IR statements, instead of the hundreds of cumbersome x86 instructions. The translation from an x86 instruction to our IR is designed to correctly model the semantics of the original x86 instruction, including making otherwise implicit side effects explicit. For example, we correctly model instructions that set the `eflags` register, prefixes that allow single instruction loops (e.g., `rep`), instructions with hidden operands (e.g. $imul\,\%ebx$ which also operates on eax and edx), and instructions that behave differently depending on the operands (e.g., shifts).

Our IR is shown in Table 1. It has assignments ($r := v$), binary operations ($r := u \square_b v$), unary operations ($r := \square_u v$), loading a value from memory into a register ($r_1 := *(r_2)$), storing a value ($*(r_1) := r_2$), direct jumps to a known target label (jmp ℓ), indirect jumps to a computed value stored in a register (ijmp r), and conditional jumps (if r then jmp ℓ_1 else jmp ℓ_2).

When converting the execution trace to the IR, we need to mark the output of the selected system calls so we can trace how the sent messages are derived from these values. For each occurrence of one of the selected system calls, we mark the return values of the system call as *symbolic*. A symbolic variable is a variable with a special name that is never assigned a concrete value throughout the IR.

Then, for each instruction that operates on a symbolic variable, we propagate this symbolic marking to

Instructions	i	::=	$*(r_1) := r_2 \mid r_1 := *(r_2) \mid r := v \mid r := u \square_b v$ $\mid r := \square_u v \mid$ `label` $l_i \mid$ `jmp` $\ell \mid$ `ijmp` r $\mid$ `if` r `jmp` ℓ_1 `else jmp` ℓ_2
Operations	$\square_b$	::=	$+, -, *, /, \ll, \gg, \&, \mid, \oplus, ==, !=, <, \leq$ (Binary operations)
	$\square_u$	::=	$\neg, !$ (unary operations)
Operands	v	::=	n (an integer literal) $\mid r$ (a register) $\mid \ell$ (a label)
Reg. Types	τ	::=	reg64 | reg32 | reg16 | reg8 | reg1 (number of bits)

Table 1: Our RISC-like assembly IR. We convert x86 assembly instructions into this IR.

the destination. For example, given the instruction $add\%eax, \%ecx$, which adds the contents of eax and ecx placing the result in ecx, where eax is symbolic and ecx is concrete, then the value of ecx after the instruction becomes symbolic. Thus, symbolic variables represent values that depend on the output of the selected system calls.

After we have translated the execution trace into the IR, we convert the IR into Static Single Assignment form (SSA), a representation in which every variable is assigned exactly once [23]. Since SSA is a well-known technique, we do not describe it here in detail.

5.1.2 Memory conversions

One of the advantages of dynamic analysis over static analysis is that we know the exact memory addresses that the program accessed during execution, since dynamic regions have already been allocated when they are used and the content of registers are available when indirect addressing modes are used.

Still, there are certain issues related to memory that we need to address. In particular, the simplification modules can reason about memory accesses but this reasoning is expensive. We have found out that if we do not preprocess our IR to simplify the memory reasoning, then the simplications usually run out of memory.

Memory accesses with different memory sizes: The x86 architecture allows memory accesses of different sizes (i.e., one, two, or four bytes in 32-bit architecture). This makes it difficult to reason about data dependencies that involve memory. We address this problem by converting all memory accesses in the IR to work on a single byte level.

Modeling memory locations as registers: For all memory accesses that use a concrete memory index, that is, when the value of the memory index does not depend on the selected system calls, we model the memory access as a read or write to a temporary register that represents the specific memory location. This reduces the number of array accesses, which are expensive to reason about.

Memory indices that depend on the input: The above conversion can only happen when the indices are concrete. When the indices depend on the input we need to perform some additional processing.

If a memory read or write has a symbolic index, i.e., the memory address has been derived from a symbolic variable, we know that values derived from this value have a dependency on the symbolic index. Ideally, in this case we would perform range analysis on the symbolic index to determine which other memory locations could be accessed using the symbolic index. However, in our current implementation, we restric the symbolic variable to be the same one that appeared in the execution. We plan to incorporte detailed range analysis in the near future.

To summarize, in this section first we have presented two memory related optimizations that we have found to significantly increase the performance of the simplification step. Without these optimizations the simplification phase becomes much more expensive. Then, we have presented how to deal with memory accesses that operate on symbolic indices. The IR that results after these conversions have been applied, is the input to the Dynamic Program Slicing algorithm, which we present next.

5.2 Dynamic Program Slicing

The problem of finding dependencies in a program has been extensively studied in the programming languages community. The goal is to find all the statements in a program that affect a variable occurrence. One proposed solution is Dynamic Program Slicing [3], the dynamic analysis counterpart to Program Slicing [30]. Dynamic Program Slicing can be defined as follows: Given a particular execution history $hist$ of program P, and a variable var, the dynamic slice of P with respect to $hist$ and var is the set of all statements in $hist$ whose execution affected the value of var, as observed at the end of the execution.

The difference between Program Slicing and Dynamic Program Slicing is that the latter works only on a specific execution, so it only contains the dependencies for that run. The reason we use Dynamic Program Slicing versus Program Slicing is that the static slices produced by Program Slicing are often too large due to memory aliasing.

Dynamic data slices contain only data dependencies while full dynamic slices contain both data and control dependencies. Control dependencies capture the different decisions taken by the program to follow the specific

execution path (i.e., the conditions of all the conditional jumps taken during the execution). Data dependencies capture how values were derived during that execution. In this paper we only include data dependencies in our slices. The intuition behind this decision is that control dependencies are not needed to understand how a specific value was derived. On the other hand they capture whether different values in the execution would follow a different execution path. We argue why understanding a single execution is often enough for our purposes in Section 6.

In our problem, the execution history is the execution trace, converted to our IR representation and having been through the SSA and memory conversions. The variables are the different bytes in a message being sent. We obtain a slice for each variable, where each slice contains all the statements whose execution affected the value of one variable.

The generated slices contain all the information about how each byte of a sent message was constructed, from the output of the selected system calls, during the original dialog. Each slice is much smaller than the complete IR, around 0.2% of the total size on average and below 1.5% of the total size for the largest ones. Still, the slices are too large for understanding how each variable was generated and in Section 5.3 we explain how we further simplify the dependencies by generating input-output formulas.

We have implemented a slicing algorithm proposed by Zhang et al [32], 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 IR and allows generating multiple slices (e.g., one per byte in the message being sent) in a single pass. This algorithm has the advantage that it does not need to generate the dynamic dependence graph first. Our experience has shown that generating the dynamic dependence graph is too expensive for large traces as previously observed by Venkatesh et al. [29].

5.3 Formula Generation and Simplification

The formula generation takes as input the slices generated by the slicing algorithm, i.e., one slice per byte in the message being analyzed, and outputs a set of input-output formulas that capture for each byte in the message, the exact way it was derived from the output of the selected system calls. It comprises two steps.

First, we perform symbolic execution [18] in the slice, which generates a preliminary formula that combines all the statements in the slice. The problem with using this preliminary formula is its size. There are two main reasons why this formula is large: redundant instructions and additional statements introduced by our SSA and memory conversions. For example, the following two instructions: "add $0x1,%eax" followed by "add $0x2,%eax", are clearly identical to "add $0x3,%eax". Removing this redundant instructions, in their IR representation, makes the formula simpler but semantically equivalent. In addition, our SSA and memory conversions insert redundant statements, because doing so makes the conversions themselves easier and less prone to implementation errors. This is a common technique used in compilers, where sequential optimizations might introduce redundant statements, which are removed at the end by some simplification stage such as dead code elimination.

Thus, the second step simplifies this preliminary formula to remove those inefficiencies and outputs the input-output formula. We use two simplification modules. One is a decision procedure, called STP [15], which provides a simplification interface. To be able to use this interface, we have modules to convert our expressions to STP's input language and back to our IR representation. The second simplification module we have developed ourselves with additional simplifications not yet available in STP such as expression reordering in some commutative binary operations, which allows further constant folding. The overall effect of the simplification is a large reduction in the size of the formula.

6 The Function Generation Phase

The last step in the analysis stage is to generate the transformation function that the rewriter will use during the editing stage. So far, we have obtained the input-output formulas for the different messages in the original dialog. The input-output formulas precisely capture, how the dynamic fields were derived from the output of the selected system calls during the original dialog.

But, if the output of the selected system calls was different than in the original dialog, then the execution could have gone through a different path. In this case, the input-output formula may be different than the one we have generated. One example is with variable-length dynamic fields. Note that our input-output formulas generated in the previous step only allow byte-by-byte substitution of the dynamic fields. The reason being again that the input-output formulas only capture what happened in one execution, the one that originated the original dialog, where the variable-length dynamic fields had some specific length. It could happen that the rewriter needs to rewrite a variable-length dynamic field with a new value that is longer than the value in the original dialog. Then, there would be some bytes in the new value of the variable-length dynamic field that we would not have enough information to rewrite by using information only from one previous execution.

Thus, in this step, we enhance the dynamic analysis done in the previous step with static analysis to capture additional information not present in previous executions. In

Program	Version	Type	Size
Bind	9.3.4	DNS server	224kB
Microsoft ftp	5.1.2600	FTP client	40kB
Micros. nslookup	5.1.2600	DNS client	70kB
FileZilla	0.9.23	FTP server	570kB
TinyICQ	1.2	ICQ client	11kB

Table 2: Different program binaries used in our evaluation. All binaries are for Windows. The size represents the main executable if there are several.

particular, in this case, we perform limited static program slicing, where we statically identify statements which may be needed to compute the variable of interest and then convert this static slice into a program. This program will then be able to handle variable-length dynamic fields because it could include loops as well. The actual technique how to perform such static slicing and ouptput the slice as a program is similar to the techniques we have developed for a different application, vulnerability signature generation [5]. Due to scope and space limit, we refer interested reader to this paper for details.

7 Evaluation

In this section we present our evaluation results. We have evaluated our system using 5 different binaries and three different protocols as shown in Table 2. All binaries are for Windows and have been run on a Windows XP environment. They range in size from the 11kB of the TinyICQ client to the 570kB of the FileZilla FTP server. Both the Bind DNS server and the FileZilla FTP server are widely deployed implementations, which we use to study our techniques on real size binaries. For all the experiments we use the same set of 44 selected system calls, which can be grouped into the six categories that were introduced in Section 3.1: Network, File, Time, Registry, Keyboard, and Host identifiers.

The protocols under study include diverse dynamic fields such as session identifiers, IP addresses, ports, user names, passwords, and configuration data. The results show that we can correctly identify the dynamic fields and generate the corresponding transformation function.

7.1 DNS Dialog

In this experiment we study a DNS query/response session. We show the input-output formulas extracted for each byte of the DNS response. The DNS client is the nslookup binary shipped with Windows XP, while the DNS server is the open source Bind server.

Identifying the dynamic fields: Figure 3 shows a summary of the dynamic fields identified in the DNS response sent by the Bind server. The numbers above the fields denote the offset with respect to the start of the response and the labels below the fields indicate the field type. There are three types of dynamic fields in the DNS response: cookies, configuration data and direction fields. We label those fields as Cookie, Config and Direction respectively. As explained, the field boundaries and the direction fields in the dialog are extracted by previously proposed techniques [7,11,19]. The remaining dynamic fields are cookies and configuration data.

Cookies are fields that have been directly copied from the received DNS query. Cookies include the session identifier present on the first two bytes of the DNS response and all the fields containing the original DNS query (offsets 12 through 28). Configuration fields are fields that contain data that has been derived from file. Here, all fields have been derived from the server's database file, which contains all the DNS records. There are numerous configuration data fields in the DNS response such as, among others, the Time-to-Live (TTL), and the email address to send complains (*hostmaster*).

The fields labeled as Fixed do not show any dependencies to the selected system calls. The remaining case, labeled Net, is a cookie field but that it has not been directly copied from the DNS query and thus we mark it differently. This byte is part of the Flags field and shows a dependency on some bytes of the received DNS query, but this dependency is not a direct copy from the DNS query and thus we mark it differently. We give more details about this byte in the following paragraphs. Here we only note the surprising fact that both bytes belonging to the Flags field are marked with different field types. This happens because the Flags field contains subfields smaller than one byte and those subfields might have been derived from different sources.

Understanding how to rewrite the fields: Here we show input-output formulas for some of the bytes belonging to the dynamic fields in the DNS response and illustrate how they are used to rewrite the value. The session identifier (ID) field has been directly copied from the DNS query to the response, without any modification. Thus, the input-output formulas for the two bytes of that field are:

```
OUTPUT_0:reg8 = INPUT_18514142_0:reg8;
OUTPUT_1:reg8 = INPUT_18514142_1:reg8;
```

where for each INPUT variable the first suffix is an input stream identifier and the second one an offset in that stream. In this case, the suffix 18514142 identifies the DNS query and the suffix 0 indicates that it has been derived from the first byte of that message. It is by observing these input-output formulas that we learn that these two bytes are part of a Cookie field.

Not all input-output formulas are this concise. For example the input-output formula for byte 2, the first byte of the Flags field, is the following:

```
(0xf:reg8 & (0xf:reg8 &
  (INPUT_18514142_2:reg8 &
```

```
    0xf8:reg8) >> 3:reg8
  )) << 3:reg8
|
 (0x8f:reg8 & (0x4:reg8
   |  (0x80:reg8
     | (0x1:reg8 &
         (0x8f:reg8 &
           INPUT_18514142_2:reg8)
  ))))
```

This input-output formula indicates that this byte is only dependant on the second byte of the DNS query, and exactly how it was derived from this value. The second byte of the query is part of the Flags field and the different constants that appear in the input-output formula are the masks used to extract the different flag fields. For example, the constant 0x8f near the bottom, zeros out the opcode field from the query, since it needs to have a new value in the response.

The input-output formulas for the configuration data fields are very large and thus we are not able to include them here. For example, the input-output formula for the first byte of the Serial Number field, contains 462,501 elements, which can be classified as binary operations, constants, casts, and variables. Though the input-output formula is complex, it only contains 10 unique input variables, representing bytes 103 through 112 of the database file, which contain the serial number as a string. The reason why these input-output formulas are so large is that most configuration data is encoded as integers in the DNS response, but the data is read from file as a string. The needed string to integer conversions generate large input-output formulas. Note that, large input-output formulas are difficult to interpret by a human, but they can be effectively used for rewriting the dynamic field values. Thus they are added to the transformation function.

Replaying the DNS response: The set of input-output formulas for the DNS query and response get included into the transformation function, which allows the rewriter to rewrite the dynamic fields in the modified dialog. For example, the rewriter impersonating the DNS server would know how to rewrite the ID field by just copying the one received in the DNS query from the client. The rewriter could as well choose to rewrite the configuration data, with a different DNS record to convince the client to connect to a server different than the one it requested. Thus, by combining previously proposed techniques to identify field boundaries, and the techniques presented in this paper to automatically generate the transformation function we have shown that we enable automatic protocol replay.

7.2 FTP Dialog

In this experiment we analyze an FTP session. Here, we use the Microsoft command line ftp client to start the control channel with the Filezilla server, then we log in as the anonymous user (providing no password) and query a directory listing. The directory listing triggers a new data channel to be established, where the client sends its IP address and a new port to the server, who connects back to the client using those parameters. This is known as FTP standard mode. The dialog is shown in Figure 4, where each line represents a different message and the first letter represents the originator of the message, where S stands for the server and C stands for the client.

S: 220 FileZilla Server version 0.9.23 beta
C: USER anonymous
S: 331 Password required for anonymous
C: PASS
S: 230 Logged on
C: SYST
S: 215 UNIX emulated by FileZilla
C: PORT 10,0,0,1,149,167
S: 200 Port command successful
C: LIST
S: 150 Opening data channel for directory list.
S: 226 Transfer OK

Figure 4: FTP dialog. Each line represent a message. The first letter denotes the originator of the message, where S stands for the server and C for the client.

Identifying the dynamic fields: On the client side of the dialog we identify two types of dynamic fields: keyboard input and host identifier fields. Keyboard input is data provided interactively by the user using the keyboard. The first message sent by the client contains the USER command and the user name typed by the user during login. We properly identify such dynamic field and mark it with the Keyboard type.

Additionally, on the client side we also detect two different host identifier fields, the IP address and the port. We detect these fields because they are derived from the output of the getsockname function, one of our selected system calls in the Network group. The process followed by the client is the following. First, the client binds a new port letting the kernel select the port number to bind. Then, the program calls the getsockname function to obtain the IP address and port that the kernel selected in the previous bind call. Finally, it encodes the IP address and port and puts them in the output buffer to be sent to the server.

On the server side we detect one cookie field and we also identify the IP address and port being sent by the client. The cookie field appears in the third message, which echoes back to the client the *anonymous* user name previously sent by the client. We are also able to identify the IP address and port from the server side by monitoring how the parameters to the connect system call are derived from previously received network data.

Understanding how to rewrite the fields: The input-

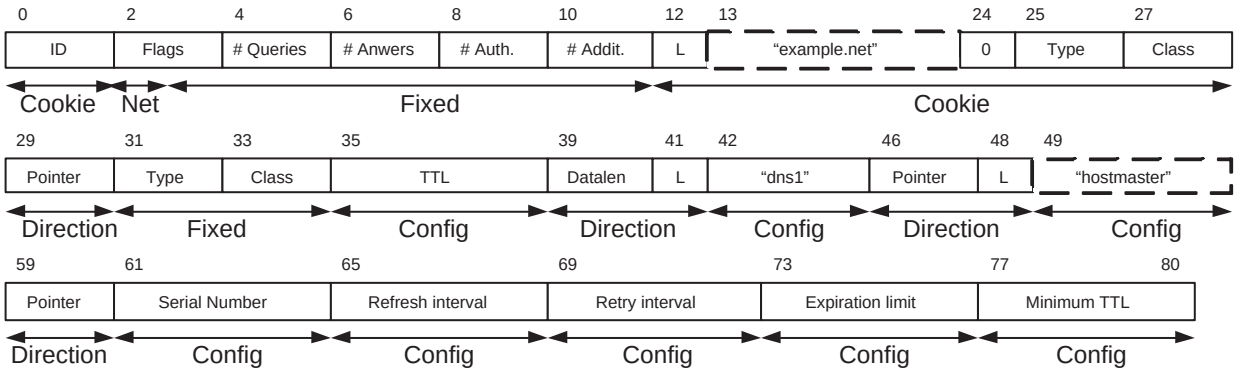

Figure 3: Direction fields identified in a DNS response.

output formulas for the IP address and port are only generated on the client side, even if those dynamic fields are detected on both sides, since only the client constructs these fields before sending the PORT command. The input-output formulas corresponding to the IP address and port encoding range in size from 14 to 74 elements. The input-output formulas indicate that all the commas included in the PORT command are concrete values with no dependencies to the output of the getsockname system call. The input-output formulas also indicate that the port is derived from bytes 4 and 5 of the getsockname parameters, and the IP address is derived from bytes 6 through 9 of the parameters. The first four bytes of the getsockname parameters are the socket descriptor and the address family, which are not used to derive the encoded port and IP address.

Replaying the FTP session: When the rewriter wants to replay the client side of the dialog, it might want to rewrite the address or port number that the server will connect to, maybe even redirecting it to a different host. In this case, the IP address and port get encoded as a variable-length string. Thus, our input-output formulas are not enough to build the transformation function, and we need to include some additional information about the encoding. We obtain such information from static analysis.

In particular, from the dynamic information, we derive that we only need to perform static slicing of the binary on the sections between the getsockname and the send calls, which significantly reduces the amount of code to analyze. In this case the static slice shows that the encoding gets done by one call to the vsprintf function in the msvcrt.dll library (the name of the function is obtained using IDA Pro) with the following format string “PORT %d,%d,%d,%d,%d,%d” where the first four parameters are the four bytes of the IP address and the last two are the port bytes. All these six bytes are derived from the output of the getsockname sytem call. This information is then encoded in the transformation function, so that when it wants to rewrite the IP address or the port, the rewriter just needs to perform a similar call to vsprintf with that format string and the new values. Thus, by combining previously proposed techniques to identify field boundaries, and the techniques presented in this paper to automatically generate the transformation function we have shown that we enable automatic NAT rewriting.

Steps in the formula generation phase	Run time
IR generation	3min 58sec
SSA generation	12min 53sec
Memory conversions	43.5sec
Slicing (46 slices)	23sec
Formula Generation (46 slices)	14sec
Formula Simplification (46 slices)	28sec

Table 3: Run time for the different steps in the formula generation phase using the FileZilla execution trace. The time in the last 3 steps is the total for all 46 slices.

Run time: Here we provide some performance numbers to give the reader an idea about the scalability of our approach. Table 3 shows the running time for the FileZilla execution trace durig the formula generation phase. We only show the run time for the Filezilla execution trace, because it is the largest of our execution traces being around 3GB. The results show that we can run the complete formula generation phase in under 20 minutes with the SSA translation taking about 66% of the whole execution time. The reason SSA is slower than the other steps is that our implementation is general and has not been optimized for execution traces. Notice that, the last three steps show the total time for the 46 slices that were computed.

The total time for the complete analysis stage, which includes the execution monitoring and the function generation phases in addition to the formula generation phase, is under 30 minutes. Note that, this is the time to generate the transformation function. The online editing stage takes negligible time to apply the transformation function with the new values for the dynamic fields.

7.3 ICQ Dialog

In this experiment we analyze an ICQ login session. This time we only have access to the client binary, thus we will be only able to replay that side of the original dialog. Here, the client application asks the user for the user name and password and tries to login into the server.

Identifying the dynamic fields: We correctly identify two keyboard input dynamic fields in the login request, which correspond to the user name and the password.

Understanding how to rewrite the fields: The input-output formulas show that the user name is being directly copied into the login request message at offsets 14 through 22. Interestingly, a modified version of the password appears at offsets 27 through 31 in the sent message The resulting input-output formulas are:

```
OUTPUT_27 = 0xf3:reg8 ^ INPUT_123_9:reg8;
OUTPUT_28 = 0x26:reg8 ^ INPUT_123_10:reg8;
OUTPUT_29 = 0x81:reg8 ^ INPUT_123_11:reg8;
OUTPUT_30 = 0xc4:reg8 ^ INPUT_123_12:reg8;
OUTPUT_31 = 0x39:reg8 ^ INPUT_123_13:reg8;
```

This shows that the password is being XORed against a constant string. This constant string is usually called the roast array in ICQ, and the process is known as roasting the password.

Replaying the ICQ session: In this case, the rewriter might want to login using some different account information it has access to. Here, we cannot directly use the input-output formulas, since they would allow us to replay only passwords up to 5 characters. Since the roast array is larger than 5 characters, we would not know the right constant to XOR against the sixth character of the new password. Thus, we extract the whole roast array from the TinyICQ binary by using static analysis techniques, and add this information to the transformation function so that the rewriter can use this array to rewrite the password.

8 Related Work

We group previous work into four categories.

Dialog rewriting: Leita et al. [20] and Cui et al [12] and propose heuristics to identify dynamic fields from network traces with application to the replay problem. Our work differs in that we use binary analysis since binaries are richer in protocol semantics than network traces, which mostly contain syntactic information. Newsome et al. [25] introduce a binary analysis technique to replay complete executions. Our work differs in that our goal is to replay the protocol messages and that our technique does not need information about the state of the verifier.

Protocol reverse engineering: Previous work on protocol reverse engineering has focused on identifying the field structure in protocol messages. Leita06 et al. [19] and Cui et al [11] propose to use variations between different sessions to extract the protocol message format strictly from network traces. Caballero et al. [7] present dynamic binary analysis techniques to extract the field boundaries from a given message. Lim et al [21] use static analysis to extract the format from the output by a program. Their approach needs the user to identify the output functions and their corresponding parameters. Our works builds on previous techniques to identify the field boundaries in protocol messages [7]. We extend those techniques to deal with protocol field semantics.

There has also been previous work on identifying traffic that uses the same application-layer protocol [22], and on analyzing how to extract the application-level structure in network data [17] There has been also research on languages and parsers that simplify the specification of network protocols [4, 26].

Dynamic Taint Analysis: Another technique that could be used to identify the inputs involved in generating an output is dynamic taint analysis [8–10, 24, 28]. Compared to dynamic taint analysis, our input-output formulas capture not only which inputs were involved in the generation of the output but also capture exactly how those inputs were used to compute the output.

Dynamic slicing Our work takes advantage of previous research on dynamic slicing [3]. Previous dynamic slicing applications have focused on debugging program source code but there have been other applications as well [31]. We use a precise slicing algorithm proposed by Zhang et al. [32] in this work. Venkatesh et al. [29] showed that precise slicing algorithms generate slices that can be orders of magnitude smaller than imprecise dynamic slices.

9 Conclusion

In this paper we have proposed a binary analysis approach to the dialog rewriting problem. For the dialog rewriting to be successful, the values of the dynamic fields in the original dialog usually need to be rewritten with new values. Our solution automatically generates a transformation function, which can be used by the rewriter to rewrite the values of the dynamic fields during the modified dialog.

We have proposed two novel techniques to identify the dynamic fields present in the original dialog by 1) analyzing how the data in sent messages has been derived from the output of selected system calls, and 2) analyzing how the parameters from selected system calls have been derived from data previously received over the network. Our results show that we can successfully identify fields such as IP addresses, ports, cookies, user names, passwords, and configuration data.

We have also proposed a technique to generate input-output formulas which precisely capture how the data sent over the network was generated during the original dialog. Our techniques builds on data dependency analysis techniques such as Dynamic Program Slicing. The input-output formulas form the core of the transformation function because they normally capture how the values of the dynamic fields have to be encoded. In addition, our techniques allow a deep understanding on the protocol semantics useful for problems such as protocol reverse engineering, building application layer gateways, or intrusion detection systems.

We have implemented Rosetta, a system that supports our techniques. We have evaluated Rosetta using 5 binaries and 3 widely used protocols: FTP, DNS and ICQ. Our results indicated that we can successfully generate a transformation function that can be used to rewrite the dynamic fields. Thus, in this paper we have shown that when combining our techniques to automatically create the transformation function with previously proposed techniques to identify field boundaries, we enable automatic protocol replay and automatic NAT rewriting.

References

[1] ISO 8601 Date and Time Standard. http://isotc.iso.org.

[2] Qemu: Open Source Processor Emulator. http://fabrice.bellard.free.fr/qemu/.

[3] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. *ACM SIGPLAN Notices*, 25(6), White Plains, NY, June 1990.

[4] N. Borisov, D. J. Brumley, H. J. Wang, and C. Guo. Generic Application-Level Protocol Analyzer and Its Language. *Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2007.

[5] D. Brumley, Z. Liang, J. Newsome, and D. Song. Automatic Generation of Multi-Path Vulnerability Signatures. http://www.cs.cmu.edu/ dbrumley/bitblaze, Carnegie Mellon University Technical Report.

[6] J. Caballero, S. Venkataraman, P. Poosankam, M. G. Kang, D. Song, and A. Blum. FiG: Automatic Fingerprint Generation. *Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2007.

[7] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. *ACM Conference on Computer and Communications Security,* Alexandria, VA, Oct. 2007.

[8] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime Via Whole System Simulation. *USENIX Security Symposium*, San Diego, CA, Aug. 2004.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. *Symposium on Operating Systems Principles*, Brighton, United Kingdom, Oct. 2005.

[10] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural Support for Protecting Control Data. *ACM Transactions on Architecture and Code Optimization,* Dec. 2006.

[11] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic Protocol Description Generation from Network Traces. *USENIX Security Symposium*, Boston, MA, Aug. 2007.

[12] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-Independent Adaptive Replay of Application Dialog. *Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2006.

[13] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. *USENIX Security Symposium*, Vancouver, Canada, July 2006.

[14] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-Peer Communication Across Network Address Translators. *USENIX Annual Technical Conference*, Anaheim, CA, Apr. 2005.

[15] V. Ganesh and D. Dill. A Decision Procedure for Bit-Vectors and Arrays. *Proceedings of the Computer Aided Verification Conference*, July 2007.

[16] M. Holdrege and P. Srisuresh. Protocol Complications with the IP Network Address Translator. RFC 3027, Jan. 2001.

[17] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-Automated Discovery of Application Session Structure. *Internet Measurement Conference*, Rio de Janeiro, Brazil, Oct. 2006.

[18] J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19, 1976.

[19] C. Leita, M. Dacier, and F. Massicotte. Automatic Handling of Protocol Dependencies and Reaction to 0-Day Attacks with ScriptGen Based Honeypots. *International Symposium on Recent Advances in Intrusion Detection*, Hamburg, Germany, Sept. 2006.

[20] C. Leita, K. Mermoud, and M. Dacier. ScriptGen: An Automated Script Generation Tool for Honeyd. *Annual Computer Security Applications Conference*, Tucson, AZ, Dec. 2005.

[21] J. Lim, T. Reps, and B. Liblit. Extracting Output Formats from Executables. *Working Conference on Reverse Engineering*, Benevento, Italy, Oct. 2006.

[22] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected Means of Protocol Inference. *Internet Measurement Conference*, Rio de Janeiro, Brazil, Oct. 2006.

[23] S.S. Muchnick. *Advanced Compiler Design and Implementation.* Academic Press, 1997.

[24] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2005.

[25] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic Protocol Replay By Binary Analysis. *ACM Conference on Computer and Communications Security,* Alexandria, VA, Oct. 2006.

[26] R. Pang, V. Paxson, R. Sommer, and L. Peterson. Binpac: A Yacc for Writing Application Protocol Parsers. *Internet Measurement Conference*, Rio de Janeiro, Brazil, Oct. 2006.

[27] N. Schear, C. Kintana, Q. Zhang, and A. Vahdat. Glavlit: Preventing Exfiltration At Wire Speed. *ACM Workshop on Hot Topics in Networks*, Nov. 2006.

[28] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution Via Dynamic Information Flow Tracking. *International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct. 2004.

[29] G. Venkatesh. Experimental Results from Dynamic Slicing of C Programs. *ACM Transactions on Programming Languages and Systems*, volume 17, 2, New York, NY, USA, 2003.

[30] M. Weiser. Program Slicing. *International Conference on Software Engineering*, Mar. 1981.

[31] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A Brief Survey of Program Slicing. *SIGSOFT Software Engineering Notes*, 30(2), New York, NY, USA, 2005.

[32] X. Zhang, R. Gupta, and Y. Zhang. Precise Dynamic Slicing Algorithms. *International Conference on Software Engineering*, May 2003.