# Network Dialog Minimization and Network Dialog Diffing: Two Novel Primitives for Network Security Applications

M. Zubair Rafique
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
zubair.rafique@cs.kuleuven.be

Juan Caballero
IMDEA Software Institute
Madrid, Spain
juan.caballero@imdea.org

Christophe Huygens
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
christophe.huygens@cs.kuleuven.be

Wouter Joosen
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
wouter.joosen@cs.kuleuven.be

## ABSTRACT

In this work, we present two fundamental primitives for network security: network dialog minimization and network dialog diffing. Network dialog minimization (NDM) simplifies an original dialog with respect to a goal, so that the minimized dialog when replayed still achieves the goal, but requires minimal network communication, achieving significant time and bandwidth savings. We present network delta debugging, the first technique to solve NDM. Network dialog diffing compares two dialogs, aligns them, and identifies their common and different parts. We propose a novel dialog diffing technique that aligns two dialogs by finding a mapping that maximizes similarity.

We have applied our techniques to 5 applications. We apply our dialog minimization approach for: building drive-by download milkers for 9 exploit kits, integrating them in a infrastructure that has collected over 14,000 malware samples running from a single machine; efficiently measuring the percentage of popular sites that allow cookie replay, finding that 31% do not destroy the server-side state when a user logs out and that 17% provide cookies that live over a month; simplifying a cumbersome user interface, saving our institution 3 hours of time per year and employee; and finding a new vulnerability in a SIP server. We apply our dialog diffing approach for clustering benign (F-Measure = 100%) and malicious (F-Measure = 87.6%) dialogs.

## Categories and Subject Descriptors

C.2.2 [**Computer Systems Organization**]: Network Protocols; D.4.6 [**Operating Systems**]: Security and Protection

## General Terms

Security

## Keywords

Network dialog minimization, network dialog diffing, network security, network delta debugging.

## 1. INTRODUCTION

A network dialog comprises the exchange of network traffic between peers to achieve a goal, which can be benign such as visiting your favorite website or downloading a file from a remote server, but also malicious such as exploiting a networked application. For example, a simple goal like visiting the root page of Amazon.com produces a complex network dialog that involves multiple remote servers and requires establishing as many as 41 TCP connections and exchanging up to 330 HTTP messages, used among others to download the HTML content, figures from CDNs, perform analytics, and display advertisements from ad networks.

Two fundamental primitives for network security are replaying a network dialog previously captured in a network trace and comparing network dialogs. Dialog replay enables applications such as testing which server versions are vulnerable to a exploit [11], evaluating intrusion detection systems [18], building interactive honeypots [23], and active botnet infiltration [5, 8]. And, comparing network dialogs enables clustering malware samples based on their network behavior [38] and building C&C signatures [16, 31].

Many tools have been proposed to replay traffic from a trace [7, 18, 28, 35] and prior work proposes techniques for replaying a network dialog regardless of the protocols involved [11, 27]. However, a network trace may contain much traffic unrelated to the replay goal. For example, a trace obtained at the network's boundary may capture an attack on an internal server that an analyst wants to replay. But, it likely contains much traffic (e.g., Gigabytes) from other hosts not attacked, and traffic from the attacked server unrelated to the attack, e.g., because it happens before or after the attack. Before replaying the attack an analyst may want to remove all unrelated traffic to better understand the attack, and also for efficiency since the attack traffic may be a small part of the full trace and the attack may need to be replayed many times (e.g., against different server versions).

In this work, we introduce the problems of *network dialog minimization* (NDM) and *network dialog diffing*. Network dialog minimization is the problem of given an *original dialog* that satisfies a goal, producing a *minimized dialog* comprising the smallest subset of the original dialog that when replayed still achieves the same goal as the original dialog. In essence, the minimized dialog provides a shortcut to the goal, removing all connections and messages in the original dialog unrelated to the goal. A minimized dialog enables understanding what parts of a dialog really matter for the goal e.g., determining which messages and fields are really required to exploit a network server without expensive code analysis. Furthermore, it provides huge reductions (up to 71 times) in time and band-

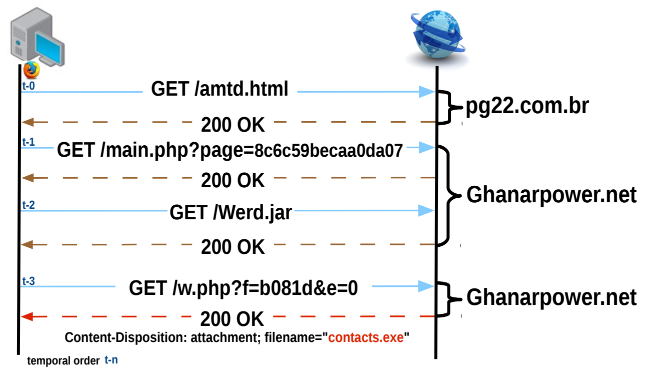


**Figure 1: A drive-by download network dialog.**

width to achieve the goal, which is fundamental when replaying a dialog repeatedly over time, e.g., in active botnet infiltration [5, 8].

In this work, we propose *network delta debugging* the first technique to automatically minimize a network dialog. Our insight is that NDM is a generalization of the problem of minimizing an input that crashes a program, for which delta debugging is a well-known solution [39]. Our network delta debugging technique generalizes delta debugging so that it can be applied to remote applications, allows application-specific goals beyond crashing an application, and takes advantage of the hierarchical structure of a network dialog.

Network dialog diffing is the problem of given two dialogs, identifying how similar they are, how to align them, and how to identify their common and different parts. In this work we propose a novel technique for dialog diffing, which aligns two network dialogs by finding a mapping between the dialogs that maximizes similarity. Once aligned, it identifies which elements (e.g., connections, messages) have been inserted, removed, and modified. We also define a dialog similarity metric that captures how similar two dialogs are.

We have applied our dialog minimization and diffing techniques to 5 applications. First, we apply NDM to build drive-by download milkers, which periodically replay a minimized drive-by download dialog to collect the malware an exploit server distributes over time. We have built milkers for 9 exploit kits and have integrated them into an infrastructure that has collected over 14,000 malware samples. A milker achieves a 34x reduction in replay time and their use enables reducing our malware collection infrastructure from three to a single host. Second, we apply NDM for measuring the current status of well-known cookie replay issues in popular websites [1, 4, 14, 17, 30]. The use of NDM achieves a 71x reduction in replay time, saving over 20 hours of processing each day. Our measurements show that despite years of research on the topic 31% of the top 100 Alexa domains do not destroy the server-side state when a user clicks the logout link. Thus, the user remains effectively logged in. For 17% of all Alexa top 100 sites their cookie can be replayed for more than a month, creating a very large window for session hijacking. Third, we use NDM for vulnerability analysis, finding a new vulnerability in the OpenSBC SIP server. Fourth, we apply NDM for simplifying an unnecessarily cumbersome proprietary web interface to move up and down a windows sunblind in an automated building, converting it to a simple command-line tool. The simplified interface saves up to 3 hours of time a year per employee, and $10,000 that the building vendor would charge for modifying the proprietary interface. Finally, we evaluate our dialog diffing approach for clustering similar dialogs. Our dialog clustering achieves 100% precision and 100% recall on 60 benign dialogs from the top 30 Alexa sites, and 100% precision with 78% recall when clustering 91 malware dialogs from 6 families.

**Contributions:**

- We introduce the problem of network dialog minimization (NDM) and present network delta debugging, the first technique to solve NDM.

- We propose a novel dialog diffing approach that finds a mapping that maximizes similarity between two network dialogs, and then identifies their common and different elements. We define a dialog similarity metric that leverages the alignment.

- We present a novel encoding of a network dialog as a tree, which captures the dialog's hierarchical structure of connections, messages, and fields.

- We apply our dialog minimization and diffing techniques to 5 applications: building drive-by download milkers, cookie expiration validation, simplifying user interfaces, vulnerability analysis, and dialog clustering.

## 2. OVERVIEW

A *network dialog* comprises the network traffic exchanged between peers for achieving a goal such as sending an email or installing malware in a victim's host through a drive-by download. At the core of this work is the intuition that a network dialog can be represented as a *network dialog tree* that captures the hierarchical structure of the network dialog, which comprises connections, messages, and fields in messages.

A network dialog tree enables automatically operating on a network dialog, e.g., replaying the network dialog, minimizing it, or comparing two dialogs. Figure 1 shows a simple drive-by download dialog where a victim visits a compromised site (`pg22.com.br`) that redirects the victim to an exploit server at `Ghanarpower.net`, which checks the browser and Java plugin version (GET /main .php), redirects the victim's browser to download a Java exploit (GET /Werd.jar), and if exploitation succeeds redirects the browser to download a malware sample (GET /w.php). Figure 2 shows the corresponding dialog tree, where the first level below the root corresponds to connections, the second level to messages in those connections, and deeper levels to the hierarchical field structure of each message. We detail the network dialog tree in Section 3.

The remainder of this section presents our novel operations on the network dialog tree (Section 2.1) and describes applications of those operations (Section 2.2).

### 2.1 Network Dialog Tree Operations

We propose two novel operations on network dialog trees: dialog minimization and dialog diffing.

**Dialog minimization.** One important use of network dialog trees is to automatically replay a previously captured network dialog. The network dialog can be periodically replayed to the same entities involved in the dialog or to different ones (e.g., by replacing their IP addresses / domain names). We take dialog replay one step further and introduce the problem of *network dialog minimization* (NDM), which is the process of given an *original dialog* that satisfies a *goal*, automatically producing a *minimized dialog* that, when replayed, achieves the same goal as the original dialog but with minimum traffic, i.e., removing all connections and messages not needed to achieve the goal. Further removal of elements from the minimized dialog causes the goal not to be reached.

For example, Figure 3 presents the minimized dialog for the drive-by download original dialog in Figure 1, where the goal is to download the malware sample being distributed. The minimized

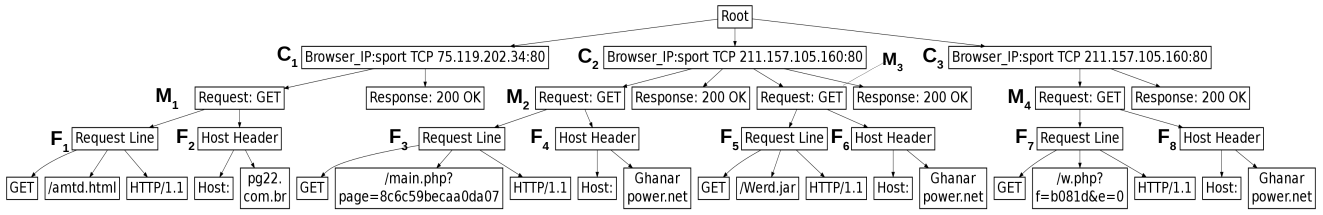


**Figure 2: Network dialog tree for the dialog in Figure 1. For brevity, response nodes are not expanded.**

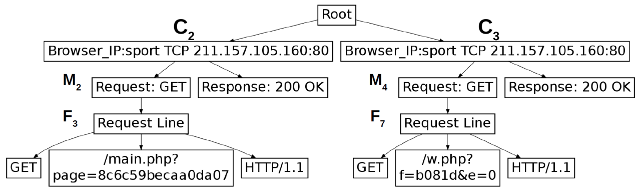


**Figure 3: Minimized drive-by download dialog.**

dialog shows that it is possible to download the malware by simply replaying two of the four requests in the original dialog.

**Dialog diffing.** Two dialogs that achieve exactly the same goal but are taken at different points in time, or from different end hosts, may intuitively be similar but may include a large number of divergences due to non-determinism. For example, a visit to Microsoft's MSN.com root page produces a complex network dialog with 82 connections connections and 177 requests. Visiting MSN.com again one day later produces a similar dialog but with 99 connections and 222 requests instead.

Network dialog diffing is the problem of given two network dialogs automatically identifying how similar they are, which parts are the same across them, and which parts different (i.e., connections and messages added, removed, or modified). Solving the dialog diffing problem implies solving the dialog similarity and dialog alignment problems as well. Our approach first defines similarity metrics between messages, connections, and dialogs. Then, it uses the similarity results to align both dialogs. Once aligned, common and different parts between both dialogs are output.

## 2.2 Dialog Tree Applications

This section introduces the 5 applications we use to demonstrate our techniques.

**Building drive-by download milkers.** A drive-by-download milker is a lightweight program that periodically downloads a malware sample from an exploit server. Its goal is similar to a honeyclient, which given a URL, visits it, gets exploited, and downloads the malware sample being distributed. But, a drive-by download milker does not go through the exploitation process and thus does not require running a heavyweight VM with a vulnerable browser; it simply replays a minimized drive-by download dialog. A drive-by download milker takes as input a minimized drive-by download dialog for a specific exploit kit (e.g., RedKit or CoolExploit), a set of exploit server IP addresses, and a time value specifying how often to replay the dialog. At each iteration it downloads a malware sample from each exploit server.

Drive-by download milkers need to be used in combination with honeyclients since it is not always possible to replay a drive-by download dialog. However, they achieve large performance savings. To combine them, we first run a honeyclient on a URL obtained from an external feed. If the URL leads to exploitation, the honeyclient outputs an original dialog, which our infrastructure immediately tries to minimize using our dialog minimization technique. If minimization succeeds, then it builds a signature on the URL of the exploit server so that next time a URL for the same exploit kit is seen a milker is used instead of a honeyclient. Our infrastructure periodically visits each exploit server every few hours and from different IP addresses. Thus, using a milker instead of a honeyclient quickly adds up in performance gains.

We have used this combination of drive-by download milkers and honeyclients to collect over 14,000 malware samples during one year from a single machine. The malware collection results and their analysis have been introduced in prior work [26]. However, that work did not present the network dialog minimization technique at the core of our milkers. In addition, our prior work only built milkers for two exploit kits (BlackHole 1.x and Phoenix), while this work builds milkers for 9 (Section 6.1).

**Cookie expiration validation.** Most web applications generate an authentication token upon successful validation of a user's credentials (e.g., username and password) and embed it into a cookie that is stored in the user's browser. The user login is typically protected by HTTPS. However, a significant fraction of web applications (37% of the top 100 Alexa sites) use plain HTTP for subsequent connections because of performance and compatibility issues. Since the unencrypted cookie is included in following requests to the server, a network attacker (e.g., on a public WiFi) can capture the cookie and include it in its own connections to hijack the session, impersonating the user [1,4,13,14,17,30]. This enables the adversary to access and change the user's sensitive information and to steal the account by changing the user's password.

To mitigate cookie replay attacks, the web server should expire the authentication token in the cookie after some time forcing the user to re-authenticate afterwards (thus minimizing the time window for an adversary to launch a session-hijacking attack). Here, it is not enough to delete the cookie in the user's browser when she clicks on a logout link. The web application also needs to remove the authentication token on the server-side, otherwise the attacker can still replay it, even if the user has (apparently) logged out.

While these problems have been known for over a decade [14, 30], it is necessary to understand if web applications are addressing it and how fast. However, manually checking cookie expiration times for many web applications is cumbersome because each application's cookie has a different format and may be encrypted. The most general way to test cookie expiration times is to replay the cookie periodically until the server stops accepting it. For this, we use our dialog minimization technique on a valid authenticated dialog for each application under test and automatically replay the minimized dialog at fixed intervals.

We have applied our approach to the top 100 Alexa sites that do not use full HTTPS communication, measuring how many sites allow cookie replay and for how long (Section 6.2). Our measurements show that 31% of the top 100 Alexa domains still do not destroy the server-side state when a user clicks the logout link and that 17% have cookies the live over a month, creating a large opportunity window for session hijacking.

**Simplifying user interfaces.** Web applications often require the user to go through several manual steps (e.g., traversing multiple webpages, filling forms) to reach an interesting state. Network dialog minimization can be used to produce a shortcut to that state that removes unnecessary steps and which can be automatically replayed. This simplifies application access, e.g., converting a web based application into a command line tool. Simplified user interfaces save user time and also provide benefits for disabled users [21]. For example, one of our institutions recently moved to a brand new fully automated building. This building has no manual switches; tasks like turning on the lights, moving sunblinds, and setting the temperature of an office can only be performed through a web application that interfaces with a building automation system (BAS). A simple task like moving the sunblind down when the sun hits the computer screen, requires accessing three different webpages (login, office selection, value setting) through a cumbersome interface. Unfortunately, the web application and the BAS interface are proprietary and undocumented. Any changes to them requires expensive work by the BAS manufacturer. We have applied our dialog minimization technique to generate minimized dialogs for moving a sunblind up and down, and a command line tool that replays them (Section 6.3). Employees can now perform these tasks more efficiently saving 3 hours per person and year.

**Vulnerability analysis.** Replaying an attack on a vulnerable networked application is a useful capability that enables testing the vulnerability and determining which versions are vulnerable [11]. In addition, minimizing a network attack dialog enables understanding the conditions leading to exploitation without expensive code inspection, and the minimized dialog can be used as an exploit signature [12]. Furthermore, minimization is fundamental to isolate the real attack when the input is a network trace (e.g., captured at an IDS) that contains an attack on a vulnerability but also much other traffic unrelated to the attack. We have applied our network dialog minimization technique to minimize an attack on the OpenSBC SIP server (Section 6.4). The minimized dialog still exploits a patched version of the OpenSBC server unveiling a previously unknown vulnerability, which has been acknowledged by the author and assigned an OSVDB identifier [29].

**Dialog clustering.** Our dialog diffing approach defines a similarity metric between network dialogs that enables clustering multiple similar dialogs at once. We have evaluated dialog clustering on 60 benign and 91 malicious dialogs (Section 6.5). The benign dialogs correspond to visiting the top 30 Alexa sites twice, spacing the visits one day apart. Our clustering groups with perfect accuracy and recall the dialogs from the same site in the same cluster, despite the many changes between both dialogs due to non-determinism. The malicious dialogs are obtained by running malware, capturing their C&C communication. Here, the clustering achieves 100% accuracy and 78% recall. The perfect accuracy enables integrating our clustering as a first step into existing signature generation tools [31, 33].

## 3. NETWORK DIALOG TREE

A network dialog may comprise multiple connections, each with multiple messages on both dialog directions, and each message with its own field hierarchy. The network dialog tree captures this hierarchical structure. The nodes of the tree correspond to connections, messages, or fields depending on the depth in the tree. An edge from a parent node to a child node implies that the child belongs to the parent, e.g., that a message belongs to a specific connection or a field to a specific message. Nodes at the same depth and with the same parent (e.g., all messages in a connection) respect a temporal ordering in which the beginning of a node (e.g., the first packet in a message or the first byte in a field) appeared on the network earlier than the start of the node on its right.

The root represents the complete dialog and is annotated with the list of peers involved in the dialog (at least two but possibly more), each peer characterized by an IP address, a DNS domain, or both.

Nodes at depth 1 correspond to connections. Each connection node has 7 attributes: source and destination peers, source and destination ports, transport protocol, application protocol, and the timestamp of the first packet. We build UDP "connection" nodes by grouping all UDP packets between two peers until no traffic is seen between them for a predefined window of time. The source peer corresponds to the initiator of the connection. Connections are ordered using the timestamp of their first packet.

Nodes at depth 2 correspond to the messages in the connection. Each message node is annotated with the peer that sends the message, which captures its direction. For UDP connections, each packet is considered a separate message. Messages in the same connection are ordered using the timestamp of their first packet.

The nodes at depth $\geq 3$ correspond to fields in the message. Each field node is annotated with the range of bytes it occupies in the message and fields can only have completely disjoint or completely enclosed ranges (i.e., fields cannot partially overlap). Field nodes can be *hierarchical fields* if they have children or *leaf fields* otherwise. The children of a hierarchical field cover disjoint parts of the parent's range. For example, every HTTP GET message in Figure 2 contains a Request-line field that in turn contains 3 other fields: the method, the URI, and the protocol version. Field nodes are ordered using the start offset of the field in the message it belongs to (offset zero is the first byte).

**Node dependencies.** A network dialog tree also captures field dependencies, e.g., a field may be the length or checksum of another field in the same message. Dependencies can also happen across messages, e.g., the value of a Cookie field received from a web server in an HTTP response may be sent in a later HTTP request to the same server. There can also be dependencies across connections, e.g., the Referer and Location HTTP headers indicate that a request was originated by a redirection in a previous response.

**Protocol knowledge.** The network dialog tree is a general abstraction that can be used to represent any network dialog regardless of the protocols involved. However, only the construction of the first level, i.e., extracting the connections nodes, is protocol-independent. Extracting the message and field levels requires knowledge about the application protocol used in the connection. We assume the availability of the grammar of each protocol involved. When the grammar is not available, prior work has shown how to recover it from an implementation of the protocol [6, 10, 24, 37]. In this paper, we focus on HTTP and SIP dialogs since these are the only protocols needed by our applications.

## 4. DIALOG MINIMIZATION

Minimizing a network dialog is an intuitive process: create a *test dialog* by removing some part of the original dialog, replay the test dialog, and check if the goal is still achieved. If the test *passes* (i.e., the goal is still achieved) the part removed is not needed to achieve the goal and thus should not be in the minimized dialog. If the test *fails* (i.e., goal not achieved), it is needed and should be kept in the minimized dialog. The main goals are that the minimized dialog should not contain unnecessary parts and that the number of tests to produce the minimized dialog should be as small as possible.

Our insight is that network dialog minimization (NDM) is a generalized version of the problem of minimizing an input that crashes a program, for which *delta debugging* is a well known solution [39]. The differences are that (1) NDM deals with remote networked applications, which we may not control, rather than a local program; (2) NDM's goal is not necessarily crashing the remote application, but more generally to drive it to some externally visible state; and (3) NDM operates on sequences of connections and messages rather than a file or a single network packet. We detail how we address these differences next.

**Reset button.** The first difference is that in input minimization the analyst controls the program for which the input is being minimized. More specifically, the analyst can restart the program before each test. Thus, the tests are independent because the program starts each test in the same initial state. However, NDM deals with a remote server which we may not control because it may belong to another party, e.g., an exploit server. This is problematic because when we perform a test, the remote server may be left in a state different from its initial state. Thus, the next test starts in a different server state and the tests are not independent. Without independent tests, the minimization may produce an incorrect result. For example, the minimized drive-by download dialog in Figure 3 shows that messages $M_2$ and $M_4$ are both needed to download the malware. If a test consisted only of message $M_2$ and the next test only of message $M_4$, the second test would reach the goal thanks to the server state set by the first test, which would be incorrectly interpreted as only $M_4$ being needed to reach the goal.

The solution to this problem is to define an application-specific *reset button* that the minimization process can use to reset the remote server to its initial state before each test. We propose one reset button implementation, applicable to many network server types, but other implementations are possible.

The intuition behind our reset button is that a network server handles multiple simultaneous clients by keeping a separate state machine for each. Thus, if the replayer uses a previously unused IP address for every test, then at the start of a new test every endpoint will be in its initial state. For each level of the tree with $n$ nodes, the minimization may generate up to $n^2+3n$ tests, so that our reset button may require a large IP address pool. In addition, the remote application may limit the geographical location of IP addresses it serves. To identify this case, the minimization first conducts a geographical distribution test (GDT) where it replays the full original dialog from one IP address in each region. If some regions are blocked, the reset button is modified to utilize only IP addresses in the allowed regions.

To implement the reset button we use a commercial Virtual Private Network (VPN) that offers exit points in more than 50 countries, each with a pool of IP addresses, totaling more than 45,000 IPs. Each test changes the VPN observable IP. In our applications we never run out of IP addresses. For very large trees, NDM supports limiting the minimization to a certain tree depth.

**Goal function.** The second difference is that NDM minimizes the original dialog with respect to a goal that may not be crashing the remote network application. For example, in the drive-by download dialog in Figure 1 the goal is downloading a malware binary from the exploit server. NDM takes as input a boolean *goal function* that given the responses received from the peers during the test, returns true if the goal was achieved in the test, false otherwise. The implementation of the goal function is dialog-specific. In some applications it is enough to check the received responses, e.g., whether a response from an exploit server contains an executable. Other goal functions may be more complex, e.g., for vulnerability analysis it may check if the application crashed through local monitoring or by sending a probe to see if the application still responds.

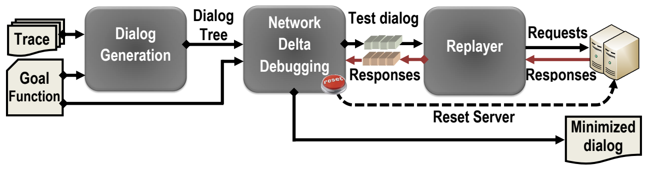


**Figure 4: Network dialog minimization architecture.**

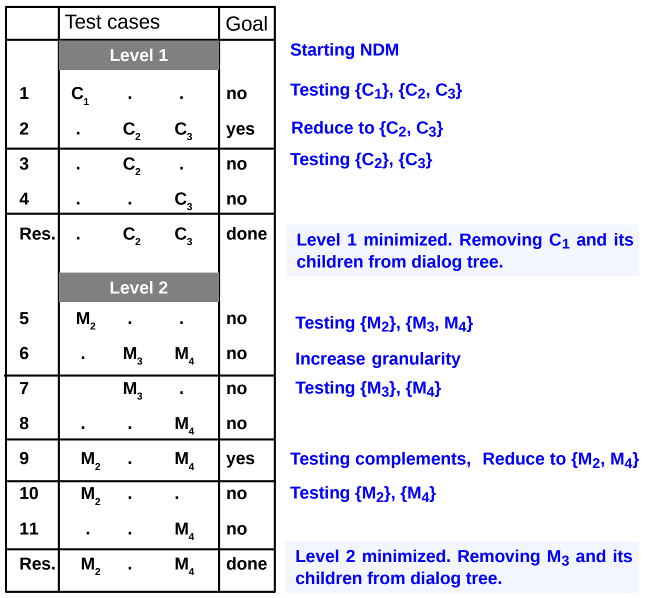
**Dialog tree.** The third difference is that NDM does not deal with minimizing a single input, but rather a sequence of network connections and messages. Our insight is that by encoding a network dialog as a tree we can use a hierarchical version of delta debugging to minimize the dialog [25]. Capturing the hierarchical structure of the network dialog enables applying delta debugging (ddmin) at each level of the tree. Intuitively, this greatly speeds up minimization because only properly-formatted inputs are produced and because if a test removes the subtree rooted at one node and the test passes, then the complete subtree requires no more tests.

**Architecture overview.** Figure 4 shows the architecture of NDM, which comprises 3 modules: *dialog generation*, *network delta debugging*, and *replayer*. The dialog generation module runs once. It takes as input a network trace and the goal function. It builds a network dialog tree from the traffic in the trace and uses the goal function to remove any connections and messages after the goal was reached, since they are unrelated to the goal. The core of NDM is the network delta debugging module, which takes as input the original dialog tree, the goal function, and the reset button. It implements an iterative process. At each iteration it produces a test network dialog by removing some elements from the original dialog. The test dialog is passed to the replayer module, which replays it to the remote peers and passes the responses back to the network delta debugging module. The responses are checked with the goal function to determine if the test passed or failed, and a new test dialog is produced for the next iteration. This iterative process eventually outputs the minimized dialog.

**Network delta debugging.** Network delta debugging applies ddmin at each level of the network dialog tree from the root to the leaves, using the goal function to determine if a test passed and the reset button to restart the server state after each test. In a nutshell, ddmin executes the following 3 steps:

1. Reduce to subset: Split the current configuration into $n$ partitions. Test each partition. If a partition reaches the goal, treat it as the current configuration and go to 1.
2. Reduce to complement: Test the complement of each partition. If any reaches the goal, treat it as the current configuration and go to 1.
3. Increase granularity: Split the current configuration into $2n$ partitions, where $n$ is the current number of partitions, and go to 1. If the configuration cannot be split further, the algorithm outputs the current configuration as the minimized dialog.

Figure 5 details the minimization process for the dialog tree in Figure 2. First, it applies ddmin to the connections (level 1). Here, ddmin splits the connections into 2 subsets and tests each subset separately (tests §1–2). The first test contains connection $C_1$ (i.e., all messages in $C_1$) and fails to reach the goal. The second test contains $\{C_2, C_3\}$ and this time the goal is reached. The current configuration is set to $\{C_2\}, \{C_3\}$; the subtree rooted at $C_1$ will no longer be tested. Next, it reduces the current configuration into 2 subsets, testing $C_2$ and $C_3$ individually (§3–4), both failing to reach the goal. Since the complements have been tested and the granularity cannot be increased, level 1 minimization outputs $C_2$ and $C_3$. Next, ddmin is applied to the messages (level 2) of the remaining



| | Test cases | | | Goal | |
|---|---|---|---|---|---|
| | Level 1 | | | | Starting NDM |
| 1 | $C_1$ | . | . | no | Testing {$C_1$}, {$C_2$, $C_3$} |
| 2 | . | $C_2$ | $C_3$ | yes | Reduce to {$C_2$, $C_3$} |
| 3 | . | $C_2$ | . | no | Testing {$C_2$}, {$C_3$} |
| 4 | . | . | $C_3$ | no | |
| Res. | . | $C_2$ | $C_3$ | done | Level 1 minimized. Removing $C_1$ and its children from dialog tree. |
| | Level 2 | | | | |
| 5 | $M_2$ | . | . | no | Testing {$M_2$}, {$M_3$, $M_4$} |
| 6 | . | $M_3$ | $M_4$ | no | Increase granularity |
| 7 | | $M_3$ | . | no | Testing {$M_3$}, {$M_4$} |
| 8 | . | . | $M_4$ | no | |
| 9 | $M_2$ | . | $M_4$ | yes | Testing complements, Reduce to {$M_2$, $M_4$} |
| 10 | $M_2$ | . | . | no | Testing {$M_2$}, {$M_4$} |
| 11 | . | . | $M_4$ | no | |
| Res. | $M_2$ | . | $M_4$ | done | Level 2 minimized. Removing $M_3$ and its children from dialog tree. |

**Figure 5: Minimization of the dialog in Figure 2.**

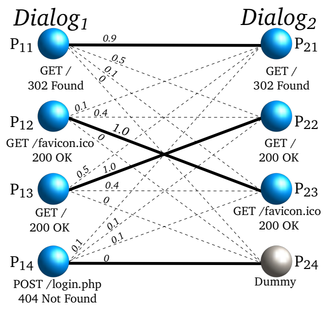
connections. The messages are split into 2 subsets, both failing to reach the goal (§5–6). Here, ddmin increases the granularity to $2n$, halving the message subsets. Thus, $M_3$ and $M_4$ are tested separately, both failing (§7–8). Note that $M_2$ had already been tested in §5. Next, the complement of $M_3$ is tested reaching the goal (§9), so the current configuration is reduced to $\{M_2\}, \{M_4\}$, but both fail (§10–11). At this point, level 2 minimization finishes determining that messages $M_2$ in $C_2$ and $M_4$ in $C_3$ are required to reach the goal. We leave level 3 minimization as an exercise to the reader. Figure 3 shows the resulting minimized dialog.

## 5. DIALOG DIFFING

Our dialog diffing approach first computes similarity scores between messages in both dialogs. Then, it aligns the dialogs by finding an optimal mapping between their messages. Once aligned, common and different parts are output and the dialog similarity score is computed. The two dialogs are compared on their structure and message content. Similar dialogs are identified despite changes in endpoints (domains, IPs), message reordering, and different number of connections and messages in both dialogs. Endpoints often change with malware since C&C domains and IPs are frequently updated, and also with benign dialogs if CDNs or load balancers are involved. Many HTTP dialogs contain asynchronous connections producing alternative message and connection orderings, which do not affect the dialog semantics. Furthermore, while different number of messages make the dialogs less similar, it is still possible to flag similarity if the spurious messages are few.

**RRP similarity.** Rather than comparing individual messages, our approach compares HTTP request-response pairs (RRPs). RRP similarity uses 11 features:

1. Request type: same HTTP method in both requests.
2. URL path: same path in both requests' URL.
3. URL filename: same filename in both requests' URL.
4. URL parameters: Jaccard index of the sets of parameter names (without values) in the URLs of both requests.
5. Referer: same value of the Referer header in both requests.
6. Response type: same response code and response status in both responses.



(a) Dialog alignment

| | $P_{21}$ | $P_{22}$ | $P_{23}$ | $P_{24}$ |
|---|---|---|---|---|
| $P_{11}$ | 0.9 | 0.5 | 0.1 | 0 |
| $P_{12}$ | 0.1 | 0.4 | 1.0 | 0 |
| $P_{13}$ | 0.5 | 1.0 | 0.4 | 0 |
| $P_{14}$ | 0.1 | 0.1 | 0.1 | 0 |

(b) Similarity matrix

**Figure 6: Example of dialog alignment (a) and similarity matrix (b) for two dialogs with 4 and 3 messages respectively.**

7. Server header: same value in both responses.
8. Location header: same value in both responses.
9. Content-Type header: same value in both responses.
10. Content length: computes $1 - \frac{||L_1 - L_2||}{max(L_1,\ L_2)}$ where $L_1, L_2$ are the content lengths for both responses (after joining chunks and decompressing, if needed).
11. Response headers: Jaccard index between the set of response headers (without values).

Features 1–3, 5–9 are boolean, while 4, 10, and 11 output a value in [0,1]. RRP similarity simply computes the average of the RRP features: $RRPsim(RRP_1, RRP_2) = \frac{1}{11}\sum_i f_i(RRP_1, RRP_2)$.

**Message alignment.** The first step to align the messages in both dialogs is building an $N \times N$ matrix, where $N$ is the maximum number of nodes in either dialog, and a row captures the similarity between an RRP in dialog 1 and all RRPs in dialog 2. If a dialog has less messages than the other, it is padded with dummy RRPs, and the similarity of any RRP with a dummy RRP is zero. This matrix can be represented as a complete bipartite graph $G = (V_1 \cup V_2, E)$ where $v_i \in V_1$ corresponds to an RRP in dialog 1 and $v_j \in V_2$ to an RRP in dialog 2. The bipartite graph is complete because there is an edge between each RRP in one dialog and all RRPs in the other dialog ($v_i \in V_1, v_j \in V_2$) weighted by their RRP similarity score. Figure 6 shows the bipartite graph (a) and similarity matrix (b) for two dialogs with 4 and 3 messages respectively, the latter padded with one dummy RRP ($N = 4$).

Our insight is that aligning the messages in both dialogs corresponds to the *assignment problem* in graph theory, which finds the mapping in a weighted bipartite graph that maximizes the total weight of the mapping. The Hungarian algorithm [22] solves this problem in polynomial time. The solid lines in Figure 6a correspond to the alignment output by the Hungarian algorithm.

The reader may realize that our approach aligns messages without considering connection structure. This is to handle misaligned connections, e.g., a connection with 4 messages in one dialog may align with 2 connections with 2 messages each in the other dialog.

**Dialog diffing.** For the diffing, RRPs aligned with similarity 1.0 are considered identical. RRPs aligned with high similarity ($s \geq 0.7$) are considered changed. RRPs aligned with low similarity ($s < 0.7$) are considered new. Threshold selection is explained in Section 6.5. In Figure 6, the diffing outputs two identical RRPs, one changed, and one new.

**Dialog similarity.** The dialog similarity is computed by summing the weights of all alignment edges and dividing by the maximum number of nodes in a dialog:

$$sim(D_1, D_2) = \frac{1}{N}\sum_{i=1}^{N} w_i \qquad (1)$$

In Figure 6, $sim(D_1, D_2) = 2.9/4 = 0.725$.

| Dialog Generation | | | | | Network Delta Debuging | | | | | | | Milking Time | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Exploit Kit | Pre-filtering Nodes | Pre-filtering C:M:F | Filtered C:M:F | Filtered IPs | L1 C:M:F | L2 C:M:F | L3 C:M:F | Tree Nodes | IPs used | GDT Pref. | Time (sec.) | Honeyclient (sec.) | Milker (sec.) |
| BlackHole 1.x | 73 | 6:6:60 | 5:5:50 | 2 | 2:2:22 | 2:2:22* | 2:2:6 | 11 | 33 | ✓ | 157.0 | 53.5 | 11.1 |
| CoolExploit | 646 | 18:58:569 | 5:5:49 | 2 | 1:1:7 | 1:1:7* | 1:1:3 | 6 | 15 | ✗ | 42.5 | 118.8 | 1.2 |
| CritiXPack | 192 | 4:19:168 | 2:7:62 | 2 | 1:4:33 | 1:1:7 | 1:1:3 | 6 | 17 | ✗ | 49.0 | 46.8 | 1.3 |
| Eleonore | 936 | 12:76:848 | 8:66:736 | 2 | 1:1:8 | 1:1:8* | 1:1:4 | 7 | 27 | ✗ | 215.8 | 238.0 | 3.3 |
| Phoenix | 132 | 12:12:107 | 7:7:73 | 1 | 1:1:7 | 1:1:7* | 1:1:3 | 6 | 15 | ✗ | 24.2 | 20.8 | 2.2 |
| ProPack | 137 | 10:12:114 | 6:6:57 | 2 | 1:1:7 | 1:1:7* | 1:1:3 | 6 | 15 | ✗ | 37.3 | 94.1 | 0.4 |
| RedKit | 154 | 8:17:128 | 2:6:57 | 1 | 2:6:57 | 2:2:19 | 2:2:10 | 15 | 71 | ✓ | 250.4 | 25.9 | 0.5 |
| Serenity | 54 | 5:5:43 | 5:5:43 | 1 | 2:2:15 | 2:2:15* | 2:2:6 | 11 | 28 | ✗ | 79.7 | 96.2 | 0.8 |
| Unknown | 79 | 5:7:66 | 5:7:66 | 2 | 1:2:14 | 1:1:7 | 1:1:3 | 6 | 18 | ✗ | 51.0 | 29.6 | 0.4 |

**Table 1: Summary of network dialog minimization of drive-by downloads dialogs.**

## 6. EVALUATION

In this section we evaluate our dialog minimization and dialog diffing techniques for 5 different applications.

### 6.1 Building Drive-by Download Milkers

Our drive-by download malware collection infrastructure runs a honeyclient on any suspicious URL received from external feeds. If the honeyclient is exploited, it outputs a network trace. The infrastructure then applies NDM on the honeyclient dialog, to check if a milker can replace the honeyclient for increased performance.

**Goal.** The goal function checks if the body of a HTTP response contains an executable file (after joining chunks and decompressing, if needed) since in the drive-by download dialogs examined the binary was not encrypted. If the exploit server drops more than one executable, the goal function can be set to output true after the first executable is seen, or after a specified number of binaries is received using a conjunction of constraints.

**Results.** Table 1 details the dialog minimization for the 9 exploit kits for which we built milkers. The left part of Table 1 summarizes the honeyclient dialogs (responses not included). It shows the number of nodes in the dialog tree built from the honeyclient network trace; the split of those nodes into connections, messages, and fields (`C:M:F`); the remaining nodes when messages and connections after the goal are filtered (e.g., C&C communication after exploitation); and the number of endpoints (IPs) in the filtered dialog. Filtering reduces the number of messages by a factor of 1.8 on average and up to a factor of 11 (CoolExploit). The filtered dialogs comprise 2–8 connections, 5–66 messages, and 1–2 peers.

The middle part of Table 1 summarizes the 3-level minimization on the filtered dialogs. It shows the connections, messages, and fields left after each minimization level (L1–L3); the number of nodes in the minimized dialog; the number of IPs (i.e., tests) used in the minimization; whether the exploit kit distributed samples geographically; and the minimization runtime. For 6 kits L2 minimization is not run (marked with asterisk) because connections remaining after L1 minimization contain at most one request each. The minimized dialogs for 6 kits contain a single request, i.e., the malware can be directly downloaded. For BlackHole 1.x and RedKit two messages are left, indicating that the first request sets server state. For Serenity, 2 connections are required because it drops 2 malware samples.

L3 minimization results show that the majority of HTTP headers are not needed to download the malware. For 7 kits, only the Request-Line (HTTP method, URI, and protocol version) is required. In addition, Redkit requires Referer and User-agent headers in both requests and Eleonore the Host header. This shows that NDM reveals which fields are checked by the server. The 3-level minimization reduces the nodes in the original tree by a factor of 39 on average, and up to a factor of 162 for Eleonore. The geographical distribution test (GDT) shows that 7 kits drop the malware to visitors from any country. BlackHole and Redkit did not serve malware for IPs in China. After GDT, only 27 IPs (i.e., tests) were needed on average to minimize a dialog. The Time column shows the minimization runtime in seconds, when run on a 32-bit 3.3GHz host with 4 cores and 4GB RAM. On average, 3-level minimization took 1.7 minutes, the slowest being RedKit with 4.2 minutes. Tree size is the dominant runtime factor, but latency and whether the server keeps state (i.e., L2 minimization needed) also matter.

There were 2 other exploit kits (BlackHole 2.x and Neutrino) for which minimization failed due to replay protection through dynamically generated URLs. For these exploit kits we used honeyclients.

**Savings.** The right side of Table 1 shows the milking time when using a honeyclient and a milker. Using a milker is 34 times faster than using a honeyclient. These values only include the time for network communication. The overall savings are larger because the honeyclient requires an additional 10 seconds to load the snapshot and transfer the URL, and runs for 4 minutes after initialization to allow exploitation to complete. Furthermore, each VM uses 512 MB, so the number of VMs that can simultaneously run on a host is memory constrained. In practice, milkers enabled us to reduce our milking infrastructure from 3 hosts down to a single host that runs all milkers and honeyclients.

### 6.2 Cookie Expiration Validation

In this experiment, we measure how many of the top 100 Alexa websites are affected by cookie replay attacks, leading to session hijacking, and the expiration time of their cookies. As preparation, we create a user account in each of the websites. Then, we obtain the original network dialog by logging into the website using those credentials, accessing the user profile, and logging out.

**Goal.** To determine if the user logged in successfully, the goal function checks if a server response includes the username string; all evaluated sites include it in the web page sent to the user after successful login.

**Minimization and replay.** 2-level minimization is applied to each dialog. Then, the minimized dialog is automatically replayed every 30 minutes, each time checking if the goal is reached. If a replay does not reach the goal, the replayer outputs the cookie lifetime.

**Results.** Table 2 summarizes the cookie replay experiments. Of the top 100 Alexa websites, 53 are not vulnerable because they either use full HTTPS (42) or have no user login (11). For other 10 we were unable to create a user account due to stricter requirements, e.g., having a cell phone from a specific region. Cookies can be replayed for 37 of the top 100 Alexa sites[1]. All 37 sites destroy the

[1]Some of those (e.g., `youtube.com`, `linkedin.com`) can be accessed fully through HTTPS, but by default use HTTP.

|  | Cookie Replays |  |  |  |  |  | No Replay |  |  |
|---|---|---|---|---|---|---|---|---|---|
| Total Websites | Replay login | Replay remote | Replay logout | Duration > 2 days | Duration > 1 week | Duration > 1 month | Comp. HTTPS | No login | Others |
| 100 | 37 | 36 | 31 | 27 | 22 | 17 | 42 | 11 | 10 |

**Table 2: Summary of cookie replay results for Alexa Top 100.**

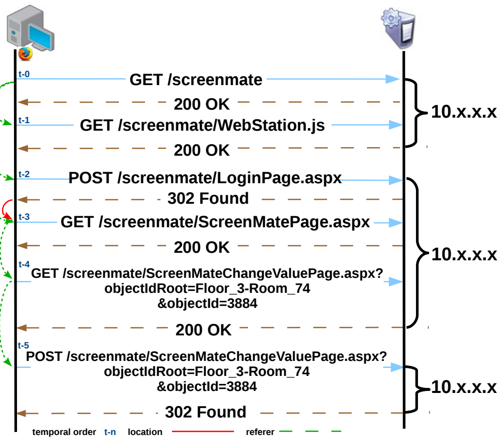

t-0
GET /screenmate
200 OK
t-1
GET /screenmate/WebStation.js
200 OK
10.x.x.x
t-2
POST /screenmate/LoginPage.aspx
302 Found
t-3
GET /screenmate/ScreenMatePage.aspx
200 OK
t-4
GET /screenmate/ScreenMateChangeValuePage.aspx?
objectIdRoot=Floor_3-Room_74
&objectId=3884
200 OK
10.x.x.x
t-5
POST /screenmate/ScreenMateChangeValuePage.aspx?
objectIdRoot=Floor_3-Room_74
&objectId=3884
302 Found
10.x.x.x
temporal order t-n
location
referer

**Figure 7: Building Automation Dialog.**

client-side cookie when the user clicks the logout link. However, only 6 sites destroy the server state after the user clicks the logout link. For the other 31 sites, cookies can still be replayed after the user believes he has logged out. One site (vk.com) does not allow cookie replay from an IP address different from the one used to login. Of the 37 sites for which replay works, for 27 the cookie still replays after 2 days, for 22 after a week, and for 17 after a month.

Our experiment flags two issues. First, 31% of popular sites do not destroy the server-side state when a user clicks the logout link. Thus, the user is effectively still logged in. In addition, for 17% of the sites the cookie replays for more than a month, creating a very large window in which session hijacking is possible.

On average a minimized dialog replays in 0.6 seconds compared to 42.6 seconds for the original, a 71 times reduction. Each site is visited 48 times a day, so minimization saves over 20 hours of processing each day.

## 6.3 Simplifying User Interfaces

In this experiment we simplify the cumbersome proprietary web interface to move a sunblind up/down in our brand new automated building, into a simpler command line tool. To begin, we capture two original dialogs corresponding to a user going through the web interface to move the sunblind down and up, respectively. Figure 7 shows the sunblind-up dialog, which requires 6 HTTP request-response pairs during which the user visits the service (an additional 2 unsuccessful requests are not included), logs in with a username and password; selects the office, sunblind item, and event type; and finally rises the sunblind.

**Goal.** The goal function checks whether the server redirects the user to a webpage that indicates that the command was successful.

**Minimization.** The minimized dialog comprises a single POST request to the web service, with the request line, the session cookie, and a payload defining the sunblind-up event. The replayer is able to replay this dialog over time without updating the cookie value, which indicates that the cookie does not have any expiration time.

We have built a 10-line C wrapper for the replayer that takes a parameter to indicate if the sunblind-up or sunblind-down dialog should be replayed. On average, a user moves the blind twice a day and each use of our tool saves 22 seconds. This amounts to significant savings of 3 hours per person each year. More importantly, we save $10,000 that the building vendor would charge for simplifying their proprietary web interface.

## 6.4 Vulnerability Analysis

Dialog minimization and replay is a useful capability in vulnerability analysis. In this experiment, we use our dialog minimization technique to replay a previously published attack on the OpenSBC SIP server [32]. Surprisingly, the minimization finds another, previously unknown, attack on the OpenSBC server.

To obtain the original dialog we deploy a test OpenSBC server, configured in UDP stateful proxying mode without authentication, and use SIPp [34] to simulate both a SIP user agent client (UAC) and a SIP user agent server (UAS). Then, we use the available attack tool against the test server, producing an attack trace.

**Goal.** To check server availability after each test, the goal function sends a benign request to the server, verifying it still responds.

**New attack.** The 3-level minimization outputs that a single 74-byte SIP INVITE request is needed to crash the server (Figure 8). However, the minimized input does not contain the Via header, whose flawed processing created the original vulnerability. Replaying the minimized input against a version of OpenSBC that patches the original vulnerability, still crashes the server. We have reported the new vulnerability to the OpenSBC author, who has confirmed that it corresponds to a null-pointer dereference in the parser. The vulnerability has been assigned an OSVDB identifier [29]. This example illustrates that beyond performance savings when replaying, the minimized dialog is also useful for capturing the essential constraints to reach and exercise a vulnerability, without expensive code analysis. It can also be used as an exploit signature.

## 6.5 Dialog Clustering

This experiment describes the use of our dialog similarity metrics for clustering benign and malicious dialogs.

**Datasets.** Our benign dialog dataset contains 60 dialogs obtained by visiting twice the root page of each Alexa Top 30 website, one day apart. Each pair of dialogs achieves the same goal but contains changes introduced by non-determinism and the time difference such as message reordering, additional messages, and small content modifications. Four sites simply redirect a user to the HTTPS version, which produces short dialogs with 1–2 RRPs. On average each benign dialog comprises 27.8 connections and 66.3 messages, the largest being a dialog for 163.com with 177 connections and 355 messages. These numbers highlight how complex web application dialogs can be.

The malicious dialog dataset comprises 91 dialogs obtained by executing 91 labeled malware binaries from 6 families (cleaman, qakbot.ae, malagent, spyeye, zbot, and zeroaccess), once each, in a contained environment that allows only HTTP C&C traffic to reach the Internet. On average, each malware family dialog comprises 2.0 connections and 2.6 messages, the largest being a dialog for spyeye with 4.5 connections and 6.0 messages.

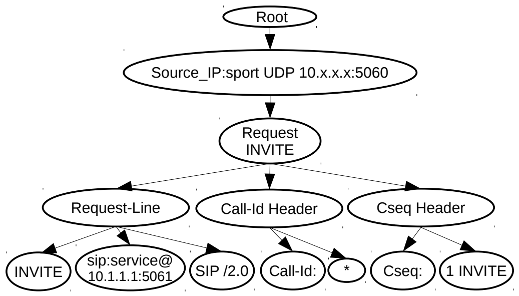


**Figure 8: Minimized SIP INVITE request revealing previously unknown vulnerability on the OpenSBC SIP server.**

**Similarity matrix.** The first step is building the similarity matrix by computing the similarity between each pair of dialogs. The average similarity between two dialogs from the same website is 0.96 and the minimum 0.80, which shows that our metric correctly captures their similarity despite changes due to non-determinism. The average similarity for dialogs from different sites is 0.16 and the maximum is 0.73 (between `facebook` and `twitter` which have only 1 RRP), which shows that the dialog similarity metric properly differentiates between similar and different dialogs. For the malicious dialogs the average similarity between dialogs from the same family is 0.74 and the minimum 0.41. The average similarity for dialogs from different families is 0.25 and the maximum is 0.73 (between `zeroaccess` and `zbot`).

**Clustering algorithms.** We use two different clustering algorithms: partitioning around mediods (PAM) [20] and aggressive. PAM takes as input the similarity matrix and the number $k$ of clusters to output, so we run it with different $k$ values, selecting the one which maximizes the silhouette width, an internal measure of clustering quality [20]. Our aggressive clustering starts with zero clusters and iterates on the list of dialogs. For each dialog, it checks if its similarity is larger than 0.8 with any dialog already in a cluster. If the comparison holds only for dialogs in the same cluster, it adds the dialog to that cluster. If it holds for dialogs in multiple clusters, it merges those clusters and adds the current dialog to the merged cluster. Otherwise, it creates a new cluster for it.

To select the 0.8 threshold we measured the clustering accuracy on a subset of the dataset for each threshold value between [0,1] with a step of 0.1 [15]. A threshold of 0.8 achieved best results.

**Clustering results.** Table 3 shows the clustering results for benign and malicious dialogs using both algorithms. For the benign dialogs, both algorithms output 30 clusters, each with the 2 dialogs for the same website, achieving perfect precision and accuracy. For the malware dialogs, the precision is perfect but the recall is 64.8%-78.0% respectively. This is because the same malware family is split into multiple clusters. There are two main reasons for this. First, zbot is a malware kit and zeroaccess an affiliate program. In both cases owners/affiliates may configure the malware differently, which creates differences captured by our metric. In addition, our metric fails to detect similarity for short dialogs with highly polymorphic requests. This is why signature generation tools use traffic clustering only as a first step before analyzing other information like content and endpoints [31, 33]. Given its perfect accuracy, our dialog clustering could be integrated as a first step in those tools.

## 7. RELATED WORK

**Traffic replay.** A number of tools have been proposed to replay traffic from a network trace [7, 18, 28, 35]. NDM differs from traffic replay in that the aim is not cloning the original dialog on a new setting, but rather to deliberately produce a different, minimized, dialog that achieves the same goal with minimal traffic. Other works automatically identify and adjust state-dependant fields, e.g., cookies and IP addresses, in the traffic to be replayed, regardless of its protocol, using network-based [11] and dynamic binary analysis [27] approaches. Our goal is not replaying any protocol, but rather minimizing a dialog or comparing two dialogs. The minimization achieves significant performance savings and enables understanding the minimum set of constraints to achieve a goal, while dialog diffing is a very different problem. Our approach assumes the protocol grammar is available, which is true in all our applications. If unknown, protocol reverse-engineering techniques can be used to recover its grammar [6, 10, 24, 37], state-machine [9], and application session structure [19]. Also related is ShieldGen [12], which generates vulnerability data patches by modifying an attack and replaying it to an oracle. A key difference with this work is that NDM operates on a complete dialog rather than a packet.

| Dataset | Algor. | Clusters | Precision | Recall | F-Measure |
|---|---|---|---|---|---|
| Alexa | PAM | 30 | 100% | 100% | 100% |
| Malware | PAM | 10 | 100% | 64.8% | 78.6% |
| Alexa | Agg. | 30 | 100% | 100% | 100% |
| Malware | Agg. | 12 | 100% | 78.0% | 87.6% |

**Table 3: Dialog clustering results.**

**Drive-by download malware collection.** Honeyclients are a popular approach for collecting malware distributed through drive-by downloads [36]. We develop drive-by download milkers that periodically replay a previous dialog for the same goal. Drive-by download milkers are more lightweight but less flexible than honeyclients, being specific to an exploit kit. Our drive-by download infrastructure [26] uses a combination of honeyclients and drive-by download milkers to achieve both flexibility and efficiency.

**Cookie replay.** Session hijacking through cookie replay has been known for more than a decade [14, 30]. Attacks on WiFi have been demonstrated [17] and tools are available to exploit it [1, 4]. It has also been mentioned that cookie replay still works after a user logs out [4, 17]. A large number of techniques have also been proposed to prevent session hijacking attacks [2, 3, 13, 14, 30]. Our work does not discover any new attacks. Instead, we show how dialog minimization and replaying tools provide an efficient and convenient way to automate the periodic visiting of a large number of sites to determine for how long cookie replays works, measuring the current state of these issues.

**Traffic clustering.** A number of works perform malware clustering based on their traffic [16, 31, 33, 38]. Most of these works cluster traffic at the packet level [16, 31, 33]. Most related is Nemean [38], which generates semantics-aware network signatures, clustering similar connections and sessions using request/response types and byte distribution features. Our work is different in that we propose a dialog alignment technique that finds a mapping between dialogs that maximizes similarity.

## 8. CONCLUSION

In this work, we have introduced the problems of network dialog minimization and network dialog diffing. We have proposed network delta debugging, the first technique to solve network dialog minimization. We have also proposed a novel technique for network dialog diffing, which aligns two dialogs by finding a mapping between them that maximizes similarity.

We have demonstrated our techniques for 5 applications: building drive-by download milkers for 9 exploit kits, measuring cookie replay on popular websites, simplifying cumbersome proprietary web interfaces, vulnerability analysis, and clustering benign and malicious dialogs.

## 9. ACKNOWLEDGMENTS.

This research was started while M. Zubair Rafique was an intern at the IMDEA Software Institute. This research was partially funded by the Research Fund KU Leuven, by the EU FP7 project NESSoS (FP7-ICT No.256980), and by the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE). This research was also partially supported by the Spanish Government through Grant TIN2012-39391-C04-01 and a Juan de la Cierva Fellowship for Juan Caballero. All opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsors.

## 10. REFERENCES

[1] Firesheep. `http://codebutler.com/firesheep`.

[2] B. Adida. Sessionlock: Securing web sessions against eavesdropping. In *International World Wide Web Conference*, 2008.

[3] A. Bittau, M. Hamburg, M. Handley, D. Mazieres, and D. Boneh. The case for ubiquitous transport-level encryption. In *USENIX Security Symposium*, 2010.

[4] S. Bowne. Cookie re-use in office 365 and other web services, 2013. `http://samsclass.info/123/proj10/cookie-reuse.htm#steps`.

[5] J. Caballero, C. Grier, C. Kreibich, and V. Paxson. Measuring pay-per-install: The commoditization of malware distribution. In *USENIX Security Symposium*, 2011.

[6] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM Conference on Computer and Communications Security*, 2007.

[7] Y.-C. Cheng, U. Hölzle, N. Cardwell, S. Savage, and G. M. Voelker. Monkey see, monkey do: A tool for tcp tracing and replaying. In *USENIX Annual Technical Conference*, 2004.

[8] C. Y. Cho, J. Caballero, C. Grier, V. Paxson, and D. Song. Insights from the inside: A view of botnet management from infiltration. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2010.

[9] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, 2009.

[10] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol description generation from network traces. In *USENIX Security Symposium*, 2007.

[11] W. Cui, V. Paxson, N. C. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *Network and Distributed System Security Symposium*, 2006.

[12] W. Cui, M. Peinado, H. J. Wang, and M. Locasto. shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *IEEE Symposium on Security and Privacy*, 2007.

[13] I. Dacosta, S. Chakradeo, P. Traynor, and M. Ahamad. One-time cookies: Preventing session hijacking attacks with disposable credentials. *ACM Transactions on Internet Technology*, 12(1), 2012.

[14] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and don'ts of client authentication on the web. In *USENIX Security Symposium*, 2001.

[15] R. Gras, E. Suzuki, F. Guillet, and F. Spagnolo. *Statistical Implicative Analysis*. Springer, 2008.

[16] G. Gu, R. Perdisci, J. Zhang, and W. Lee. Botminer: Clustering analysis of network traffic for protocol and structure independent botnet detection. In *USENIX Security Symposium*, 2008.

[17] Hamster, 2007. `http://blog.erratasec.com/2007/08/sidejacking-with-hamster_05.html#.UwdUz4ZDuKk`.

[18] S.-S. Hong and S. F. Wu. On interactive internet traffic replay. In *International Symposium on Recent Advances in Intrusion Detection*, 2006.

[19] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi-automated discovery of application session structure. In *Internet Measurement Conference*, 2006.

[20] L. Kaufman and P. J. Rousseeuw. *Finding Groups In Data: An Introduction To Cluster Analysis*, volume 344. John Wiley & Sons, 2009.

[21] S. Keates. Designing user interfaces for ordinary users in extraordinary circumstances: A keyboard-only web-based application for use in airports. *Universal Access in the Information Society*, 12(2), 2013.

[22] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2), 1955.

[23] C. Leita, K. Mermoud, and M. Dacier. scriptgen: An automated script generation tool for honeyd. In *Annual Computer Security Applications Conference*, 2005.

[24] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed System Security Symposium*, 2008.

[25] G. Misherghi and Z. Su. Hdd: Hierarchical delta debugging. In *International Conference on Software Engineering*, 2006.

[26] A. Nappa, M. Z. Rafique, and J. Caballero. Driving in the cloud: An analysis of drive-by download operations and abuse reporting. In *SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2013.

[27] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *ACM Conference on Computer and Communications Security*, 2006.

[28] Ostinato. `http://goo.gl/oo5rmn`.

[29] OSVDB. opensipstack opensbc.exe null pointer dereference remote dos, 2012. `http://osvdb.org/86607`.

[30] J. S. Park and R. Sandhu. Secure cookies on the web. *Internet Computing, IEEE*, 4(4), 2000.

[31] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Symposium on Networked System Design and Implementation*, 2010.

[32] M. Z. Rafique, M. A. Akbar, and M. Farooq. Evaluating dos attacks against sip-based voip systems. In *Globecom*, 2009.

[33] M. Z. Rafique and J. Caballero. firma: malware clustering and network signature generation with mixed network behaviors. In *International Symposium on Recent Advances in Intrusion Detection*, 2013.

[34] SIPp. `http://sipp.sourceforge.net/`.

[35] Tcpreplay. `http://tcpreplay.synfin.net/`.

[36] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Network and Distributed System Security Symposium*, 2006.

[37] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Network and Distributed System Security Symposium*, 2008.

[38] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *USENIX Security Symposium*, 2005.

[39] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2), 2002.