Advancing Cyber Forensics via Record and Replay of User-Browser Interactions

by

Christopher James Neasbitt

(Under the direction of Roberto Perdisci)

Abstract

As the pervasiveness of the Internet has increased over the last several decades so also have the prevalence of web-based cyber attacks. Unfortunately, performing detailed forensic analysis of web-based security incidents is a notoriously challenging task. Forensic analysts must often manually analyze disparate and often ephemeral information sources in an attempt to achieve a detailed view a security incident. This task is further aggravated by the shear volume of data generated by modern computer systems.

Much of the difficulty associated with investigating web-based security incidents can be mitigated by utilizing systems which can replay such incidents from their recorded artifacts. Systems which can partially or fully replay security incidents can produce insights which would require a human analyst numerous hours of error prone manual analysis to replicate. To this end we present two novel Record and Replay systems, ClickMiner and WebCapsule, which are designed to aid in the forensic analysis of web security incidents.

ClickMiner aims to automatically reconstruct user-browser interactions by replaying archived web traffic traces via an instrumented browser. This ability is useful in a number of web security scenarios including the postmortem analysis of user facing web attacks. Our evaluation demon-

strates that ClickMiner can reconstruct between $\simeq$ 82% and $\simeq$ 90% of user-browser interactions with false positives between 0.74% and 1.16% outperforming previous reconstruction algorithms.

WebCapsule is a forensic engine for web browsers which strives to record all non-deterministic inputs into the web rendering engine embedded in many popular browsers. WebCapsule enables the replay and analysis of past potentially harmful web browsing sessions in a controlled isolated environment. We evaluate WebCapsule on phishing attack instances and popular websites to demonstrate that both dangerous and benign web browsing sessions can be recorded and fully replayed while incurring reasonable overhead.

INDEX WORDS: Web Security, Cyber Forensics, Record and Replay, Web Forensics, User-browser Interactions

Advancing Cyber Forensics via Record and Replay of User-Browser Interactions

by

Christopher James Neasbitt

B.S., Valdosta State University, 2006

A Dissertation Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2015

©2015

Christopher James Neasbitt

All Rights Reserved

Advancing Cyber Forensics via Record and Replay of User-Browser Interactions

by

Christopher James Neasbitt

Approved:

Major Professor: Roberto Perdisci

Committee: Kang Li
Laksmish Ramaswamy

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
December 2015

Acknowledgments

I would like to acknowledge my family and friends for their continual support, advice, and encouragement without which this work would not have been possible. I would also like to thank Roberto Perdisci for his mentorship and guidance.

Contents

Acknowledgments **iv**

List of Figures **vii**

List of Tables **ix**

1 Introduction and Literature Review **1**

1.1 Introduction 1

1.2 Literature Review 3

2 ClickMiner: Towards Forensic Reconstruction of User-Browser Interactions from Network Traces **15**

2.1 Abstract 15

2.2 Introduction 16

2.3 Problem Formulation 19

2.4 Referrer-based Inference (RCI) 21

2.5 ClickMiner System Overview 24

2.6 ClickMiner System Details 26

2.7 Click Graph Analysis 34

2.8 Evaluation 37

2.9 Case Studies 44

2.10 Limitations and Future Work 48
2.11 Related Work 49
2.12 Conclusion 51
2.13 Acknowledgments 51

3 WebCapsule: Towards a Lightweight Forensic Engine for Web Browsers 52

3.1 Abstract 52
3.2 Introduction 53
3.3 Problem Definition and Goals 58
3.4 Approach and Challenges 61
3.5 Recording: Design and Implementation 63
3.6 Replay: Design and Implementation 72
3.7 Evaluation 78
3.8 Related Work 87
3.9 Conclusion 88

4 Summary 90

Bibliography 92

List of Figures

2.1 Example of referrer graph pruning (the length of the edges reflects the referrer delay). 22

2.2 ClickMiner system overview. 24

2.3 ClickMiner's process (simplified). 25

2.4 Examples of augmented click graph. 36

2.5 Trade-off between true and false positives. 39

2.6 Reconstructed click graph (case study 1) 45

3.1 High-level overview of WebCapsule's record and replay capabilities. Non-deterministic inputs to the embedded web rendering engine are recorded, and can be fully replayed in an isolated forensic analysis environment where no new external user inputs or network transactions are received. 56

3.2 Overview of WebCapsule's instrumentation shims. 57

3.3 Extending DevTools: instrumentation example (from `Inspector Instrumentation.idl`). 65

3.4 Simplified view of WebCapsule's platform wrapper in *record mode*. `PlatformImpl` is the actual implementation of the current underlying system platform. 67

3.5 WebCapsule's instrumentation of V8's `Math.random()` implementation (from `v8/src/math.js`). Function calls starting with '`%`' are used to call C++ functions internal to V8 from JavaScript code. 70

3.6 Simplified view of how WebCapsule records asynchronous network transactions. . 71

3.7 Simplified view of WebCapsule's replay strategy. 73

List of Tables

2.1 ClickMiner results - Group1 (no-cache) 41

2.2 ClickMiner results - Group2 (cache) 42

3.1 Functionality Tests 80

3.2 Replay correctness for phishing attack pages. Measurements performed over 112 phishing traces collected by 6 users. 82

3.3 Performance test results. Overhead computed during recording of browsing activities on popular websites. 83

3.4 Record and replay tests on popular websites. Test are marked as follows: ✔ = successful test; ★ successful test with some divergence; ✖ test with problems; ✳ test with problems that we later fixed. Multiple symbols indicate different results depending on the type of browsing activity on the site. 86

Chapter 1

Introduction and Literature Review

1.1 Introduction

As the pervasiveness of the Internet has increased over the last several decades so too have the scope and prevalence of web-based cyber attacks. As such the ability to perform accurate and timely forensic analysis in response to web-based security incidents is increasingly critical. With detailed insight into these incidents security researchers can develop stronger defenses against future attacks. Unfortunately, analyzing real-world web attacks that directly target users remains an extremely challenging and time-consuming task.

The state-of-the-art methods for reconstructing web-based security incidents generally follow two approaches. The first relies on analyzing the system artifacts of a web browser's activity produced during a security incident [39, 31]. These artifacts include the browser's history, cache files, and system logs. This approach often relies on a forensic analyst manually correlating disparate information sources often lacking adequate detail to reconstruct a precise view of the incident. The second approach leverages full network packet traces, which may provide some indications of how an incident unfolded. However, the complexity of modern web pages results in a large *semantic gap* between the web traffic and the UI events that occurred within the browser [37]. Such a

semantic gap makes it very difficult to precisely reconstruct what a victim actually saw, how the victim was ultimately attacked, and what if any information was consequently leaked.

Forensic analysts need tools mitigate the challenges which hinder the analysis of web-based security incidents. The primary goal of these tools should be to present an analyst with a semantically rich view of the security incident under investigation while requiring as little manual intervention as possible. Accordingly these tools should have the capability to automatically extract pertinent evidence from large often uncorrelated data sets. These tools should also provide functionality to automatically reconstruct a partial or complete view of the incident from the collected evidence. Alleviating an analyst of the burden of manual evidence collection and correlation as well as incident reconstruction can minimize the potential for error introduced by human analysis. Tools with the above mentioned properties can significantly reduce the cost while improving the accuracy of investigations of web-based security incidents.

Record and Replay systems provide an avenue to construct the next generation of forensic analysis tools which meet the preceding criteria. Record and Replay systems are those which capture and deterministically reproduce the execution of some system. These systems have proven useful in a variety of applications [20, 63, 13, 57]. In this dissertation we present two novel Record and Replay systems, ClickMiner (Chapter 2) and WebCapsule (Chapter 3), designed to aid in the forensic investigation of web-based security incidents.

ClickMiner aims to automatically reconstruct user-browser interactions from archived web traffic. This ability is useful in a number of web security scenarios such as postmortem analysis of user facing web attacks such as phishing or socially engineered malware downloads. ClickMiner reconstructs user-browser interactions by replaying archived (possibly partial) web traffic traces via an instrumented browser. Through our evaluation we demonstrate that ClickMiner can reconstruct between $\simeq$ 82% and $\simeq$ 90% of user-browser interactions with false positives between 0.74% and 1.16% thus outperforming a recently proposed reconstruction approach base solely upon the

analysis HTTP headers [64]. We also provide several case studies detailing how ClickMiner can be used to analyze real-world web security incidents.

WebCapsule is a record and replay forensic engine for web browsers. Our primary goal with WebCapsule is to enable always-on, transparent, and fine-grained recording (and subsequent replay) of web browsing sessions. Always-on recording allows WebCapsule to capture all (unexpected) security incidents. To make always-on recording practical, WebCapsule is designed to minimize performance overhead during recording. Additionally, WebCapsule is designed to be highly portable allowing it to be embedded in a variety of web-rendering applications, and run on a variety of platforms. WebCapsule provides a forensic analyst the ability replay and analyze past potentially harmful and often ephemeral web browsing sessions in a controlled isolated environment. We evaluate WebCapsule on phishing attack instances as well as popular websites to demonstrate that both dangerous and benign web browsing sessions can be recorded and fully replayed while incurring reasonable overhead.

1.2 Literature Review

Record and Replay systems capture and deterministically reproduce the execution of some hardware or software system. These systems have proven useful in a variety of applications ranging from debugging to computer forensics. We can generally distinguish between two types of deterministic Record and Replay systems, those that require specialized hardware support and those that are purely software based. In the following we will review the current state of software only Record and Replay systems or *RR systems*.

The remainder of this chapter is organized in the following manner. In section 1.2.1 we will examine the basic construction and operation of an RR system and define some terms used in the rest of this work. Section 1.2.2 summarizes the different targets of replay. In section 1.2.3 we describe the types of information recorded by RR systems. Section 1.2.4 defines a hierarchy of

replay fidelity. We review some common instrumentation techniques employed in RR systems in Section 1.2.5.

1.2.1 Overview

All RR systems strive to capture and reproduce with some level of fidelity the execution of a target system. This target system, or *replay target*, comprises one of the two primary components of any RR system. The means by which the replay target's execution is recorded/replayed, i.e. the *instrumentation platform*, comprises the second primary component of a RR system.

RR systems generally operate in two basic modes, *recording* and *replay*. During recording the replay target comprises an input to the instrumentation platform which records all non-deterministic events generated as a result of the the target's execution. In replay mode the instrumentation platform uses the information captured during recording to deterministically re-execute the replay target. The amount and types of non-deterministic information captured during the recording phase are dictated by the desired replay fidelity. This fidelity or *replay granularity* is discussed in further detail in Section 1.2.4.

In order to construct an RR system, the designer must answer several basic questions. The answers to the following questions form the primary constrains on the design and construction of the RR system's instrumentation platform.

- What is the replay target?
- With what granularity should the replay target's execution be replayed?
- What information is required to be captured to enable re-execution of the replay target at the desired granularity level?
- What is the method by which the necessary information can be captured during recording and utilized during replay?

The instrumentation platform of RR systems is often constructed and operates at a level logically in-between that of the replay target and the underlying system. We can envision this type of

instrumentation platform as a wrapper surrounding the replay target. During recording the wrapper is semi-permeable allowing non-deterministic information into the replay target. This information is recorded into some type of wrapper defined storage. During replay the wrapper becomes impermeable isolating the replay target from outside influence. The captured non-deterministic events are then re-injected into the replay target on a timeline matching the target's original execution.

This design offers several advantages in terms development and operation. It is often required that the instrumentation platform must capture and replicate the replay target's interactions with the underlying system. These interactions include accepting keyboard input, accessing a file system, generating network traffic, writing to volatile memory, etc. By interposing the instrumentation platform between the replay target and the underlying system the instrumentation can capture and replicate these system interactions with minimal alteration of the replay target itself. To enable certain replay granularity levels the instrumentation platform must also record and control the re-execution of certain system level primitives. For example, ReVirt [20] implements interrupt recording and re-execution by interposing itself between a guest VM and the underlying host OS. While we describe the replay target and instrumentation platform as conceptually separate entities a portion of the instrumentation platform's functionality is on occasion integrated into the replay target [6, 8, 29, 30, 41, 45].

1.2.2 Replay Targets

Individual applications are the most common replay target for RR systems. The problem of deterministically replaying the execution of applications has been well studied. These RR systems typically vary by types of instrumentation employed and the level of concurrency they can deterministically replay. RR systems have been designed for applications running on numerous platforms including Linux [6, 8, 41, 47, 50, 57], Solaris [45], Java [13, 29, 24], Android [25, 30], and Web [11].

A minority of RR systems are dedicated to replaying whole systems. Several works present RR systems [20, 22, 32] constructed to replay the execution of arbitrary operating systems along with user land applications isolated with a virtual machine. Some RR systems such as [17, 1, 63] consider networks as their replay target. These systems tend to guarantee that some portion of recorded network traffic will be re-injected into the network within certain ordering and timing constrains. Network oriented RR systems are primarily applied to network diagnostics and forensics applications.

OFRewind [63] is one such network oriented RR system designed as an interactive network debugging tool used to aid in diagnosing certain network anomalies within OpenFlow networks. The split flow architecture provided by OpenFlow allows OFRewind record both control and data plane traffic. Control plane traffic is recorded through an OpenFlow controller. Data plane traffic is duplicated at the switch level and distributed among several collection nodes. Both traffic recording and replay can be directed via user defined selection rules.

1.2.3 Sources of Non-determinism

The primary goal of any RR system's recording phase is to record the non-deterministic events generated during the execution of the replay target. RR systems focus on recording non-determinism exclusively for two primary reasons. Only the non-determinisitic events of the replay target can cause subsequent executions of the target to differ from each other. Therefore, to enable some level of replay it is sufficient to record and re-execute only the non-deterministic events from the replay target. Non-deterministic events generally form a relatively small fraction of the total information about a replay target's execution. Therefore, The total amount of information that must be captured in order to implement faithful replay is greatly reduced by recording only the non-determinisitic portions of the execution.

Several different sources of non-determinism can potentially affect a replay target's execution dependent upon both its construction and the system upon which it's executed. User input and non-deterministic system calls are among the most prevalent of these sources.

User Input

Within the context of RR systems any input provided to a replay target from any agent, biological or software, external to the target is defined as user input. This class of non-determinism includes sources such as command line arguments, interactive keyboard input, interaction with GUI elements, etc. User input is often utilized to set the initial state of the replay target as well as dynamically direct its execution. As the replay target is unable to definitively predict the interactions of an external agent nor the timing at which they will occur user input must be directly captured to be replicated.

Non-deterministic System Calls

In contrast to user input, non-deterministic system calls form a class of non-determinism directly requested by the replay target. A system call can be considered non-deterministic from the point of view of a replay target if its result can not be pre-computed before the first such call is made. Non-deterministic system calls can be divided into two categories, *stable* and *volatile*.

Stable non-deterministic system calls are those whose return value changes infrequently if at all. These system calls include requests for system properties such as the operating system version or total system memory. A common space saving technique RR systems employ involves recording the result of a single call to a stable non-deterministic system call during the record phase. During replay mode the single recorded value is simply returned as many times as requested by the replay target.

Volatile non-deterministic systems calls differ from their stable counterparts in that their return values change frequently if not constantly. Certain system calls such as those for current system

time or to a system provided pseudo-random number generator are inherently volatile. However, calls for highly dynamic system properties such as current CPU load or total available memory can also fall into this category. Requests for both local and network resources must be considered volatile as the state of the resource returned can be arbitrarily modified outside of the execution of the replay target. For instance, local files can often be manipulated by numerous programs including the operating system during the replay target's execution. Dynamically generated network resources might change between each consecutive access. Furthermore, network resources may be unreachable at the time they are requested due to network effects.

Concurrency Non-determinism

Certain sources of non-determinism affect concurrent applications exclusively. These sources of non-determinism include pre-emptive thread scheduling and shared memory accesses. Concurrent programs distribute their application logic over multiple independent flows of control, i.e. threads. The thread scheduler is responsible for determining which thread is executing at any point in time. Pre-emptive thread schedulers utilize information about current application work as well as system load to dynamically switch between executing threads at arbitrary points in their execution. Therefore, a pre-emptive thread scheduler comprises an external software component that directly influences the execution of the replay target in a non-deterministic fashion. It has been demonstrated [36, 46] that it is sufficient to capture thread scheduling decisions in order to deterministically replay concurrent applications on uniprocessor machines.

Multiprocessor machines add additional non-determinism to the execution of concurrent applications due to the fact that multiple threads can be executed simultaneously. The pattern of interleaving of simultaneously executing threads can vary between executions of the replay target. Consequently, different interleaving patterns can result in different executions of the replay target. A data race, a race condition between non-atomic variables, is example of non-deterministic event which is often exhibited only under certain thread interleaving patterns. In general, when

multiple threads access or modify shared memory it becomes necessary to enforce the recorded the thread interleaving pattern in order to deterministically re-execute the replay target. However, thread interleaving is not an issue in terms of replay if the individual threads never interact.

Hardware non-determinism

Modern computing hardware introduces difficult to capture sources of non-determinism. CPUs often exhibit irreplicable non-deterministic properties during operation. Out-of-order execution within modern CPU's ensures that program instructions are executed outside of the order specified in the program. CPU caching effects can affect instruction execution timing as well. Memory scrubbing can non-deterministically affect the timing memory access operations. Timing variations on buses and I/O devices can also introduce subtle non-determinism between program runs. As each of these hardware properties are not replayable from software, specialized hardware is often required to eliminate these forms of non-determinism.

1.2.4 Replay Granularity

A prominent characteristic of all software based RR systems is replay granularity. Specifically, replay granularity is the level of fidelity with which the RR system replays the recorded scenario. Application replay granularity can be broadly divided into four categories. Listed in order least to most faithful these categories include *output*, *value*, *concurrency*, and *instruction*. The aforementioned categorization is partially subsumptive. For example, all *value* level granularity systems within certain restrictions are also *output* level granularity replay systems.

Output Granularity

Output level granularity guaranties that the re-execution of the program ultimately produces the same output as during recording. Identical output production includes conditions such as exhibiting identical bugs or reproducing the results of a software exploit. RR systems which implement only

output granularity provide few guaranties regarding how faithfully the sequence of instructions executed during replay matches that of the original execution. For instance, systems such as [6, 41] strive to derive an execution path ultimately reaching some known state without re-executing a recorded scenario exactly.

Value Granularity

Value level granularity guaranties that during re-execution reads and writes are made with the same values to and from memory, either heap or stack, as during recording. These reads and writes include those made to and from variables, function parameters and their return values, etc. Value level granularity is sufficient to replay the execution of non-concurrent programs.

Concurrency Granularity

Concurrency level granularity further increases the fidelity of replay over value granularity by guaranteeing that code blocks, primarily function calls, are re-executed in the same order relative to other executing threads/processes. Correct re-execution of thread schedules, synchronization primitives, and shared memory accesses across multiple threads are of critical importance concurrency level RR systems. Concurrency level RR systems are sufficient to deterministically replay the executions of concurrent applications.

Instruction Granularity

Instruction level granularity guaranties that each machine instruction with its corresponding parameters is re-executed in the same absolute order during replay. This replay granularity level also guaranties that the state of the machine after each instruction is executed is the same between recording and replay. Instruction level replay granularity ultimately implies that underlying system's operation is deterministic. Due to non-programmable non-deterministic effects within hard-

ware such as, CPU cache state, timing variations on devices, etc. this level of replay granularity is not achievable without the assistance of deterministic hardware.

Replay of Network Targets

Network oriented RR systems offer weaker replay guaranties than those offered by application oriented systems. From the point of view of the network oriented each processing node, i.e switches, routers, etc., is essentially a black box of which only a certain set of outputs are visible. This lack of observable information about each node's execution heavily restricts the amount of non-determinism that can be replicated solely at the network level.

1.2.5 Instrumentation Techniques

In the following section we describe some common instrumentation techniques utilized in RR systems. We also present specific examples of how these techniques have been applied in previous RR systems.

Checkpointing

RR systems often record much more of a replay target's execution than strictly required for certain applications such as debugging or forensics. Alternately, a replay target may reach a desired point in its execution, e.g. the exhibition of a bug, only after significant amount of execution has occurred. In these cases a user would desire to skip certain portions of the recording in order to focus on other parts which are more relevant to the task at hand. RR systems which implement checkpointing provide the ability start the re-execution of the replay target at points later than the beginning of the recording. Checkpointing allows the RR system to skip irrelevant portions of a recording without sacrificing replay granularity.

Flashback [50] implements the checkpointing of arbitrary processes on uniprocessor machines via 'shadow processes'. Flashback instruments the Linux operating systems to create secondary

copies of a target process at a specified point in its execution, i.e. a checkpoint. This secondary or 'shadow' process then maintains the state of the target process at the checkpoint. A process is reverted to a checkpoint by swapping in its corresponding shadow process for execution.

In [32] the authors implement checkpointing within their 'Time Traveling Virtual Machine' by utilizing copy-on-write versioning for disk blocks and memory pages. In each instance only the differences between any two checkpoints are saved to the redo/undo logs. This technique requires significantly less CPU time and memory compared with copying the complete state of the VM at each checkpoint.

Concurrent Shared Memory Accesses

Several works have focused exclusively on the problem of recording shared memory accesses in concurrent applications. DejaVu [13] utilizes the concept of a global clock to record and deterministically re-execute thread scheduling decisions within Java applications. Recplay [45] utilizes the concept of a Lamport clock [34] to record the partial ordering synchronization events thereby reducing recording overhead. However, this system can only correctly replay programs that are free of data races. LEAP [29] attempts to reduce the recording overhead to recording shared memory accesses via an alternate approach. Instead of directly recording the thread scheduling decisions affecting the replay target the system records the results of those decisions. Specifically, each shared variable tracks the order of thread accesses it sees during execution. The recorded thread accesses are then enforced on a per variable basis during replay via instrumented bytecode.

Static Analysis

Static program analysis is a commonly utilized overhead reduction technique in RR systems. Static analysis allows the identification of certain areas of importance within an application prior to its execution. By selectively instrumenting the identified areas of importance the recording coverage of the replay target can be focused to only a subset of the application's total logic. The amount of

recording overhead eliminated via focused instrumentation correlates to the frequency with which those areas of importance are executed. This is particularly true of highly deterministic, long running replay targets. Areas of particular interest to RR systems include potentially unsynchronized shared memory accesses [29] or the utilization of inter-process communication [8].

Constrained Execution Search

Some RR systems reduce recording overhead by implementing replay via constrained execution search. The principle idea behind this technique is to produce an equivalent re-execution of the replay target matching a desired state. Both time and space requirements of recording are reduced by capturing only partial information about the replay target's execution. This reduction of recording time is shifted to the searching portion of the less resource critical replay phase. During the search candidate executions, either pre-recorded or generated on-the-fly, are compared with a partial recording of the replay target.

ODR [6] leverages the idea that it is sufficient for debugging purposes to produce any execution of a replay target that exhibits the same output as the recording. In this vein, ODR records an execution 'determinant' which uniquely defines a run of the replay target. This determinant is then utilized in a search of recorded executions which satisfy the output conditions of a program run.

PRES [41] implements an iterative replay exploration technique. In this system a partial recording called a 'sketch' of the replay target is captured and used to direct re-execution. The replayer then attempts to reconstruct non-deterministic information not contained in the sketch by iteratively re-executing the replay target. Re-execution attempts that fail to match the sketch are utilized to further constrain the next replay iteration.

Deterministic Multithreading

Several works have eliminated the need to replay non-deterministic thread schedules by enforcing deterministic thread scheduling. Eliminating this source of non-determinism provides several ad-

vantages over recording pre-emptive scheduling decisions. These RR systems no longer require complex scheduling recording and replay logic. Likewise, deterministic thread scheduling removes the space overhead required to store a high frequency source of non-determinism.

In [57] the author's utilize the concept of 'epoch parallelism' to generate deterministic thread schedules. Epoch parallelism is defined as multiple time intervals or epochs of a program's executions running concurrently. Their system, DoublePlay, executes multiple thread and epoch parallel runs of the same program on several processors. As a consequence of feedback between the thread and epoch parallel runs the system generates a 'uniparallel' schedule of the program's execution. This uniparallel schedule is in effect a deterministic serial thread schedule equivalent to a parallel execution of the program.

CoreDet [8] utilizes a deterministic thread scheduling library as a component in its deterministic compiler and runtime system. CoreDet runs concurrent applications utilizing a runtime environment implementing a 'quantum' based scheduling algorithm. The main idea behind quantum based scheduling is to split scheduling into rounds composed of two finite logical timeslices or 'quanta'. During the 'parallel' quanta all threads are scheduled arbitrarily and concurrently but are restricted from performing any kind of inter-thread communication. During the 'serial' quanta only one thread is executed at any instant effectively serializing the application's execution however, inter-thread communication is permitted.

Chapter 2

ClickMiner: Towards Forensic Reconstruction of User-Browser Interactions from Network Traces

2.1 Abstract

Recent advances in network traffic capturing techniques have made it feasible to record full traffic traces, often for extended periods of time. Among the applications enabled by full traffic captures, being able to automatically reconstruct user-browser interactions from archived web traffic traces would be helpful in a number of scenarios, such as aiding the forensic analysis of network security incidents.

Unfortunately, the modern web is becoming increasingly complex, serving highly dynamic pages that make heavy use of scripting languages, a variety of browser plugins, and asynchronous content requests. Consequently, the semantic gap between user-browser interactions and the net-

Neasbitt, C., Perdisci, R., Li, K. and Nelms, T. 2014. ClickMiner. Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14. (2014), 1244-1255.

work traces has grown significantly, making it challenging to analyze the web traffic produced by even a single user.

In this paper, we propose ClickMiner, a novel system that aims to automatically reconstruct user-browser interactions from network traces. Through a user study involving 21 participants, we collected real user browsing traces to evaluate our approach. We show that, on average, ClickMiner can correctly reconstruct between $\simeq 82\%$ and $\simeq 90\%$ of user-browser interactions with false positives between 0.74% and 1.16%, and that it outperforms reconstruction algorithms based solely on referrer-based approaches. We also present a number of case studies that aim to demonstrate how ClickMiner can aid the forensic analysis of malware downloads triggered by social engineering attacks.

2.2 Introduction

Recent advances in network traffic capturing techniques have made it feasible to record full traffic traces, often for extended periods of time (e.g., a sliding window of several days). This ability is important to enable fine-grained, off-line traffic inspection, for example to allow for a detailed postmortem analysis of security breaches or other significant network events or anomalies.

The ability to perform detailed traffic inspection is orthogonal to host-based event recording and forensic analysis, and can be especially useful in those cases in which host-based instrumentation introduces high overhead, is inconvenient, or is simply not practically feasible (e.g., in networks with permissive bring-your-own-device policies).

Among the applications enabled by full traffic captures, being able to automatically reconstruct user-browser interactions from archived web traffic traces would be useful in a number of scenarios. For example, malware infections through social engineering attacks [54] specifically target the browser and its user. Therefore, it is important to enable an after-the-fact analysis of such events, traveling back in time to study what actions the user performed *before* she arrived to the social

engineering attack page. This may in turn allow us to better understand *browsing behavior* patterns related to such attacks, and suggest new defense mechanisms.

Besides computer forensics, reconstructing user-browser interactions from traffic traces may benefit other interesting applications, such as web-usage mining [51], whose main goal is to understand how users interact with web pages to enhance their browsing experience or to improve on personalized advertisement strategies [10, 42].

Unfortunately, the modern web is becoming increasingly complex, serving highly dynamic pages that make heavy use of scripting languages, a variety of browser plugins, and asynchronous content requests. Visiting a single web page often requires the browser to generate numerous HTTP requests to fetch all objects necessary to render it correctly. Consequently, the *semantic gap* between user-browser interactions and the packets captured by network traces has grown significantly, making it challenging to analyze the traffic produced by even a single user's browsing sessions to reconstruct what activities the user performed.

Research Goals and Approach. In this paper, we aim to answer the following research questions:

- Is it at all possible to *accurately infer user-browser interactions from full packet network traces*?
- With what precision can we *reconstruct the sequence of web pages explicitly requested (or "clicked") by the user*, while filtering out automatically requested objects due to browser rendering and plugins?
- Can we infer *what element in a web page was "clicked" by the user* to reach the next desired resource?

To answer these research questions and measure the reconstruction accuracy, we propose ClickMiner, a novel system for reconstructing user-browser interactions from network traces. We also compare ClickMiner to a "naive" referrer-based click inference (RCI) approach based on ReSurf [64], a recently proposed web traffic analysis system. For both ClickMiner and RCI, we assume to be given a (possibly partial) recording of the web traffic generated by a user within a

given time window of interest. Our main goal is to reconstruct the sequence of "clicks" that the user performed within the browser. Notice that we focus only on user-browser interactions that cause the browser to initiate an HTTP request for a new web page, and that our definition of *click* goes beyond mouse clicks, and includes other events such as typing an URL directly, pressing `Enter` on the keyboard to follow a link, etc. (see Section 2.3).

The RCI approach is based on first constructing the *referrer graph* as in ReSurf [64], where a node represents an HTTP request, and two nodes are linked together by a directed edge according to their `Referer` request header fields. Then, this graph is pruned using a number of heuristics to filter out requests that were (likely) automatically generated by the browser during rendering (see Section 2.4.1).

Our ClickMiner system, instead, takes a very different approach: it reconstructs user-browser interactions by *actively replaying* the recorded HTTP traffic within an *instrumented browser*. At a high level, given the HTTP request-response pair (i.e. HTTP exchange) (q_0, r_0) related to an initial page in the trace, we force the browser to issue request q_0, and we feed it back response r_0 from the trace. Through the natural rendering of response r_0, the browser will issue another number of HTTP requests to retrieve page components. ClickMiner serves all the related responses to the browser from the trace. Essentially, the browser *consumes the traffic trace* until it reaches a *resting state*, whereby some action is needed for the browser to continue to consume the remaining HTTP traffic. At this point, we consider the next yet to be consumed HTTP exchange in the trace as a *candidate* user-browser interaction, feed it to the browser, and continue until all HTTP traffic has been consumed. In practice, the replay process used by ClickMiner is much more involved; we therefore defer the details to Section 2.6.

Contributions. We make the following contributions:

- We propose ClickMiner, a novel system dedicated to automatically reconstructing user-browser interactions from full packet captures. Our system can benefit a number of important

tasks, such as aiding the forensic analysis of security incidents involving the browser (e.g., social engineering attacks).

- Through a user study involving 21 participants, we collected 24 different traffic traces from real user browsing sessions. Using these traces, we evaluate the two aforementioned approaches. We show that ClickMiner can reconstruct in average between $\simeq 82\%$ and $\simeq 90\%$ of user-browser interactions with false positives between 0.74% and 1.16%, and that it outperforms RCI.
- We report a case study involving a real social engineering-based malware download attack, and show that ClickMiner was able to reconstruct the *full chain of user-browser interactions* that led the user to the malware download. In particular, ClickMiner reduced the amount of information to be analyzed by a forensic analyst from 316 HTTP exchanges to only 6 nodes in a *click graph*.
- To make our results reproducible and foster further research, we released the source code of our ClickMiner software and all the browsing traces collected through our user study at `http://clickminer.nis.cs.uga.edu`.

2.3 Problem Formulation

Goal. Our main goal is to reconstruct the user-browser interactions that occur during a user's browsing session, given only a full packet capture of the network traffic generated by the browser. We focus exclusively on interactions that cause the browser to initiate a request for a *new web page* (e.g., a click on a hyperlink). As an application example, we are interested in helping a forensic analyst understand what a user's browsing behavior was during a time window *preceding* (and including) a social engineering or phishing attack, or other relevant security incidents and anomalies.

Assumptions. We assume the recorded traffic we are interested in replaying was generated by a *non-compromised* browser. Namely, we assume that the browser's code itself had not (yet) been "hijacked". This includes the replay of traffic generated during social engineering and phishing attacks, as well as traffic generated *before* a browser vulnerability is actually exploited (e.g., via a drive-by attack).

Definition of User-Browser Interaction. In the remainder of the paper, we will often interchangeably use the terms "user-browser interaction" and "click". Our definition of click is intentionally lax, and broader than a mouse click. A click in our definition includes the following events: a mouse click on a page element that initiates a request for a new page; pressing `Enter` while the focus is on a page element such as a hyperlink, thus initiating a new page request; typing a new URL directly in the browser's address bar, or equivalently, clicking on a bookmarked link.

Reconstruction Output. As an example, assume that a user visits a web page p, and that within this page there exists a "clickable" DOM element e (notice that if e resides in a page frame, we still consider it as an element within the context of the outmost frame p). As the user clicks on e, the browser initiates a request q for a new web page, as well as several other automatic requests due to the rendering of the new web page (e.g., to load images, frames, etc.). Our objective is to reconstruct this information by relying only on the recorded network traffic, and thus to log the tuple (p, e, q). As we will discuss later in the paper, in some cases the exact DOM element e may not be reliably identified. Therefore, identifying and logging the tuple $(p, null, q)$ is also considered a satisfactory output, though less preferable.

In some cases, p may not exist. For example, if the user types an URL directly on the address bar or clicks on a bookmarked link, we aim to automatically identify the resulting request q, and simply log $(null, null, q)$.

HTTPS traffic. One may think that because many web services are transitioning to HTTPS, most recorded web traffic is going to be encrypted, and both the ReSurf-based RCI approach and ClickMiner will become less useful in the near future. However, it is important to notice that

many modern enterprise networks already deploy web proxies that allow for SSL man-in-the-middle [17] (SSL-MITM) to enable the inspection of all HTTPS traffic (e.g., to enforce traffic filtering policies). Therefore, enterprise networks can already easily record the content of most HTTPS communications (sensible traffic capturing policies would avoid SSL-MITM for banking applications or other known sensitive activities).

Cache. Because of the effect of the browser's cache, some requests resulting from user-browser interactions may not be directly recorded in the network traces. In this case, we use a "best effort" approach and attempt to infer and log the request q resulting from the interaction by leveraging "artifacts" of q that may be observed in the network trace.

Traffic Traces. Naturally, if packets are captured from the network edge (e.g., at a web proxy or router level), the recorded network traces may contain a mix of traffic generated by many network users. However, it is not difficult to isolate the traffic generated by a specific user's browsing session by simply partitioning the network traces according to the user's source IP address, transport and application protocol, user-agent string, and time window of interest.

2.4 Referrer-based Inference (RCI)

In this section, we briefly explain how we perform referrer-based click inference (RCI, for short). Our RCI algorithm is based on ReSurf's approach [64]. Please, refer to [64] for further details on the construction and pruning of the referrer-based graph.

2.4.1 Building the Referrer Graph

Let $l = \{(q_i, r_i)\}_{i=1...n}$ be the list of all HTTP exchanges reassembled from the packet capture, where q_i and r_i represent the i-th HTTP request and related response, respectively. We build a directed acyclic graph $\mathcal{G} = (V, E)$ where each node in V represents an HTTP exchange in l. Assume $w = (q_w, r_w)$ and $y = (q_y, r_y)$ are two nodes in such graph. A directed edge $(w \rightarrow y) \in E$ exists if

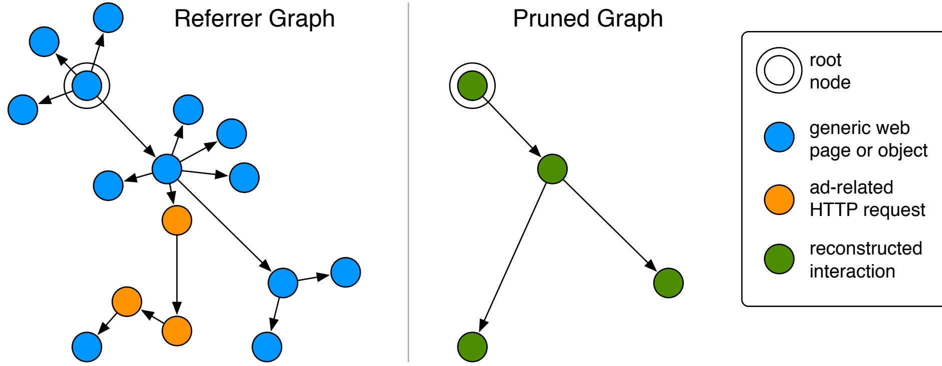

Figure 2.1: Example of referrer graph pruning (the length of the edges reflects the referrer delay).

the absolute URL related to w's response r_w is referred in the `Referer` (or `Location`) header field of q_y.

Clearly, not all nodes in $\mathcal{G}$ are generated due to user-browser interactions. In practice, most of the nodes in the referrer graph are related to automatic requests issued by the browser during page rendering. In order to infer user clicks, we would like to derive a pruned graph $\mathcal{G}'$ that only contains nodes that are directly related to user-browser interactions. To this end, we use a number of heuristics (as in [64]) that allow for identifying and pruning away nodes that are likely due to automatically generated requests. After applying these pruning heuristics to obtain $\mathcal{G}'$, which typically contains many fewer nodes and smaller connected subgraphs than $\mathcal{G}$, we classify all nodes $v \in \mathcal{G}'$ as directly related to user-browser interactions. Figure 2.1, shows a visual example of the results of the pruning process.

Pruning by Referrer Delay: Let $v_i = (q_i, r_i)$ and $v_j = (q_j, r_j)$ be two nodes in graph $\mathcal{G}$ linked by a directed edge $(v_i \rightarrow v_j)$. Also, let t_{r_i} be the timestamp of the last packet related to response r_i (as recorded in the network trace), t_{q_j} be the timestamp of the first packed related to q_j, and $\delta_{i,j} = t_{r_i} - t_{q_j}$. We call $\delta_{i,j}$ the *referrer delay*.

For most automatically generated requests, the referrer delay effectively approximates the *rendering time*, i.e., the time between when an HTTP response is processed by the browser and the time in which the requests issued by the browser due to the rendering cause page components to be loaded. As such, a large majority of "automatic" HTTP exchanges in $\mathcal{G}$ should exhibit a small referrer delay. By contrast, those requests generated due to an explicit user-browser interaction generally exhibit a longer referrer delay, because human reaction time is typically much slower than the time needed by the browser to parse an HTML file and issue subsequent page component queries. We take advantage of these differences to filter out many of the (highly likely) automatically generated requests by setting a tunable threshold θ_δ on the referrer delay, and removing all edges $(v_i \rightarrow v_j)$ and their "destination nodes" (e.g., node v_j) for which the referrer delay $\delta_{i,j} < \theta_\delta$.

Pruning Asynchronous Requests: Dynamic asynchronous requests (e.g., via AJAX), though automatically generated by the browser, may result in large referrer delays. Consequently, we may not be able to filter them out by simply leveraging the referrer delay. Fortunately, in many cases (but not always) asynchronous requests are signaled by the presence of the `X-Requested-With: XMLHttpRequest` in the HTTP request header. We therefore take advantage of this to simply filter out such asynchronous requests. The remaining asynchronous requests may not be as easily identified, and could therefore generate *false positives* (i.e., HTTP queries being classified as resulting from a direct user-browser interaction while they are not).

Pruning Ads and Social Widgets: Modern web pages typically host a significant number of components related to advertisements and social network widgets (e.g., "like" buttons). Rendering a page containing social widgets or ad-related components usually entails a complex chain of automatic requests. For example, ad requests typically go through several *advertisement networks*, until the ad's object is eventually fetched and rendered.

To identify and prune away these types of automatic requests, we first label those nodes in the referrer graph that are related to social network widgets or ads. To this end, we leverage the EasyList and Fanboy's Social rule sets for AdBlock Plus [3], a popular browser plugin.

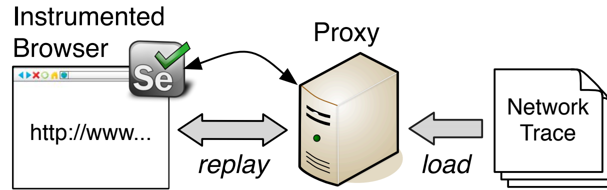

Figure 2.2: ClickMiner system overview.

2.5 ClickMiner System Overview

In this section, we give an overview of how ClickMiner works. A simplified view of the traffic replay process described below is provided in Figures 2.2 and 2.3.

In-Browser Replay: At a high level, ClickMiner works as follows (see Figure 2.3). Given a network trace containing full packet captures (1), we first reassemble all HTTP flows and the related HTTP exchanges. These flows are then loaded into a proxy application. We instrument a browser via a *browser driver* plugin implemented on top of Selenium WebDriver [4]. We chose to leverage Selenium WebDriver because it is compatible with most major browsers, making it easier to replay the traffic on different browser versions, e.g., chosen according to the user-agent string recorded in the network traces.

The instrumented browser is configured to use the proxy application as its HTTP proxy, and is responsible to replay the network traffic and reconstruct the user-browser interactions that took place during traffic capture. To initiate the reconstruction process, the browser driver will ask the proxy for the URL of the first HTTP request recorded in the trace that contains clickable elements within the body of its response (2). Then, the browser is instructed to load this URL as if a user typed it on the address bar. As the browser renders the web page (3), all related automatic object requests (e.g., to load images, frames, etc.) are sent to the proxy, which retrieves the responses from the recorded HTTP flows and passes them back to the browser (notice that the proxy is not

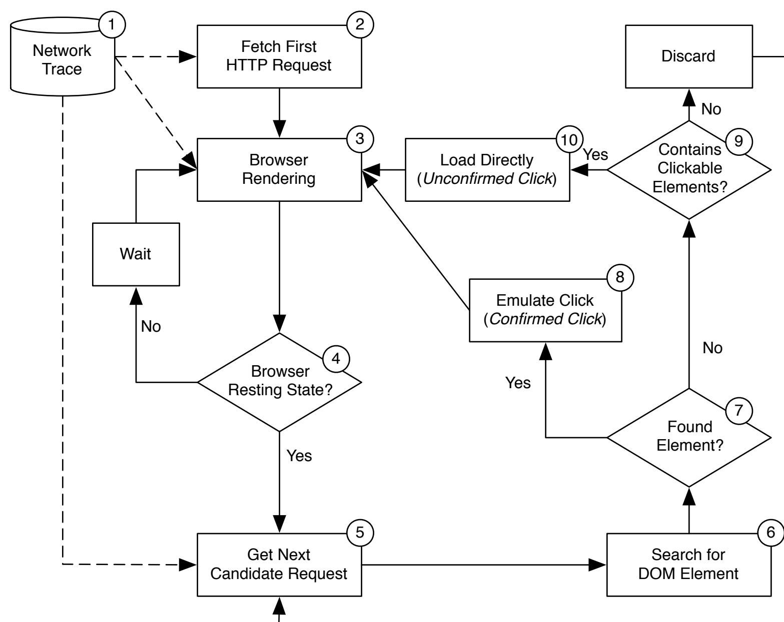

Figure 2.3: ClickMiner's process (simplified).

allowed to retrieve content from other than the recorded network trace). This could be seen as forcing all browser requests to be served from a cache.

Once the browser fetches all necessary objects and renders the current page, it will reach a *resting state* (4), whereby no further previously recorded HTTP requests would be issued by the browser without explicit user interaction (see Section 2.6.3 for a more accurate definition of resting state). At this point, the browser driver queries the proxy for the next HTTP request in the network trace that has not yet been "consumed" by the browser (5). The returned request represents a *candidate click*. To verify if this request was actually caused by a user click, the browser driver inspects the DOM of the pages currently rendered by the browser (6). If a *clickable element* is found (7), whose attributes (e.g., the `href` attribute of an <a> tag) match the URL of the candidate click, we consider this to be a *confirmed click*, which reports information including the URL of the page currently rendered in the browser (i.e., the URL in the address bar), the path in the current

page DOM of the found clickable element, and the URL of the next Web resource that would be requested as a consequence of clicking on that element. In Section 2.6 we will explain in details how (and why) our URL matching process uses an *approximate matching* approach to compensate for dynamically generated URLs.

If a confirmed click is found, our browser driver (virtually) clicks on the related element (8), thus emulating a user interaction and allowing the browser to fetch the next web page to be rendered from the proxy. If no element is found that can be clicked, and after applying a number of other checks (9), the plugin assumes the user had typed the next URL directly into the address bar (10), and therefore instructs the browser to directly load the page and log the event as an *unconfirmed* user-browser interaction. ClickMiner then continues to search for other user-browser interactions, until all HTTP traffic in the recorded trace is consumed.

Challenges: Intuitively, if all web content was "static" and encoded using well-formed HTML, the process outlined above would perfectly reconstruct all user's clicks. However, modern dynamic webpage construction as well as browser design pose a number of challenges to reconstructing user-browser interactions. In the following sections we discuss how ClickMiner copes with these challenges.

2.6 ClickMiner System Details

2.6.1 In-Browser Traffic Replay

Let $l = \{(q_i, r_i)\}_{i=1..n}$ be the ordered list of HTTP exchanges in the recorded trace, where the pairs are sorted according to the timestamp associated with the requests $\{q_i\}$. To bootstrap the in-browser traffic replay process, we scan the list l searching for the first pair (q_j, r_j) whose response contains HTML content. Essentially, we look for the first "usable" page in the trace that can be rendered by the browser that may contain clickable elements.

Then, our browser driver instructs the browser to replay request q_j. This request is received by ClickMiner's proxy, which responds by serving r_j (see Section 2.6.2). As the browser renders r_j, it may issue a set of subsequent requests $\{q_r\}$, which are sent to the proxy and served accordingly from the trace.

2.6.2 The Role of ClickMiner's Proxy

ClickMiner's proxy (see Figure 2.2) is mainly responsible for reassembling TCP flows and extracting HTTP exchanges from previously recoded network traces, and for serving the recorded web content upon request to the instrumented browser. When the browser (which is setup to use our proxy application as its *HTTP proxy*) sends an HTTP request q', the proxy extracts the corresponding absolute URL u' from q', and searches the list of exchanges, l, to find the first occurrence of an exchange whose query URL matches u'. Let (q_k, r_k) be such a pair. At this point, the proxy retrieves the content of response r_k from the trace, sends it to the browser, and marks the pair (q_k, r_k) as *consumed*.

(Approximate) Matching of Requests and Responses. Due to the highly dynamic nature of modern web pages, during replay some HTTP requests issued by the browser may not match any of the originally recorded traces. This may occur for example when URL parameters encode data that is either system- or time-dependent, or is randomly generated. To attempt to satisfy these requests, the proxy applies an *approximate matching* algorithm. Let q_u be the request issued by the browser during replay. We measure the similarity between q_u and all not-yet-consumed requests in the trace under the same domain name (or IP address) as q_u. Let q_a be one of such yet to be consumed requests. To compute the similarity between q_u and q_a, we measure the similarity between their URL path strings, the similarity of the set of parameter names, and the number of matching parameter values. We also consider q_u and q_a to be more similar when the timestamp of q_a is closer to the timestamp of the last request that was successfully consumed from the network trace. We then combine all these similarity scores, and match q_u with the most similar q_a in the

trace. In the eventuality that the browser issues an HTTP request that does not match any not-yet-consumed request in the trace, the proxy will simply respond with a `404 Not Found`.

It is possible (though we found to be rare in practice) that due to approximate matching a request may be mistakenly consumed, potentially causing ClickMiner to miss a "click" later on during trace replay. The effect of this type of inappropriate match is akin to missing clicks due to the effect of the browser cache. Section 2.6.6 details this scenario as well as ClickMiner's recovery mechanisms.

2.6.3 Browser Resting State

Since our main goal is to detect user-browser interactions, we need to distinguish between HTTP requests automatically issued by the browser due to the rendering process, and requests that would otherwise not realize themselves, if not through an explicit (broadly defined) *user click* event. To this end, we aim to detect at what point in time, after loading a page, user intervention is needed for the browser to request a new web page or object that was previously stored in the packet capture. That is, we aim to detect at what point in time the browser reaches a *resting state*.

Typically, after the browser has rendered a page, including loading all page components such as images, ads, and embedded objects, the browser will stop issuing HTTP requests. In this case, we can easily detect that the browser reached a resting state. But the simple scenario outlined above does not take into account the fact that in the case of dynamic pages, the browser may periodically issue a "page refresh" request (e.g., due to a "meta refresh" tag) or other asynchronous requests (e.g., via AJAX), which make it more difficult to decide whether the browser has reached a resting state or not.

As a concrete example, the browser may be rendering a page that uses asynchronous requests to periodically update a component of the page with some real-time news feed. However, these asynchronous requests are of little interest for the purpose of mining subsequent user-browser interactions, and therefore we would like to find a way to determine whether the user had in fact

moved on to request another page, rather than waiting indefinitely on a page that periodically updates its content.

To detect whether the browser has reached a resting state, even in the presence of dynamic content, we proceed as follows. We set a polling interval t_{poll} (10s, in our experiments), after which the browser driver checks whether the instrumented browser has issued any successful HTTP requests since the last poll. That is, we verify whether during the last polling interval, the browser has loaded any components successfully served from the recorded trace. In fact, going back to our news feed example, if during recording the user had moved on to another page, during replay we will reach a point in which the page rendered in the instrumented browser has consumed all automatic requests (and related responses) recorded in the trace, and now issues asynchronous requests that do not exist in the recorded trace. This signals that the browser may have reached a resting state. In addition, at every poll point the driver checks whether there was any change in the page DOM, with respect to the previous interval. In practice, the driver computes a hash of the current DOM, and compares it with the previous version.

Accordingly, we apply these three rules.

(R1) If no DOM change and no successful requests are observed for a time equal to $2 \cdot t_{poll}$, we decide that the browser has reached a resting state.

(R2) It is possible, though, that the DOM of a page may change continuously (e.g., via JavaScript) to realize some highly dynamic visual effect, while the browser still fits our definition of resting state, i.e., the need of (emulated) user intervention to load other content recorded in the trace. Therefore, even if the DOM keeps changing, we decide that we reached a resting state if the instrumented browser has not issued any new successful (i.e., served from the trace) HTTP request in the last $4 \cdot t_{poll}$ intervals.

(R3) It is possible for a page to take a rather large amount of time to reach a resting state based upon (R1) and (R2) alone, given a significantly long trace. Therefore, we put an upper limit

on the number of polling attempts we make before a browser window is considered to be in a resting state. We conservatively set the upper limit to $25 \cdot t_{poll}$ in our experiments.

The time threshold values are empirically chosen to strike a good trade-off between reconstruction accuracy and efficiency. The rules are applied (per window) to all windows (i.e., pages) concurrently open on the instrumented browser.

2.6.4 Reconstructing Interactions

Assume the browser has reached a resting state. At this point, our browser driver sends a control message to the proxy to request the next unconsumed request, q, in the network trace. Let u be the URL of the request q returned by the proxy. We refer to u as a *candidate interaction* URL. To verify whether q was due to a user-browser interaction or not, we proceed as follows. We inspect the DOM of all currently open windows, and match any element that includes u as one of its attribute values. For example, u may match the `href` value of an <a> tag, the `src` value of an <`img`> tag, etc. Then, we filter out all non-clickable matches. For example, we discard a match if u matches a `src` attribute value, as this means that u was automatically requested by the browser (e.g., to render an image), rather than being caused by a user-browser interaction. In addition, we discard matches to non-clickable tags, such as <`span`>, <`p`>, <`pre`>, etc.

***Confirmed* vs. *unconfirmed* interactions**. If a valid (clickable) DOM element e that matches u is found, we label it as a *confirmed user-browser interaction*, and log the page and DOM element related to the interaction. Notice that if e exists within a page frame, we still consider it as an element of the outmost page frame p, and therefore we log the tuple (p,e,q). In case no element is found, we load u in a new browser window. We then check whether the related loaded page itself contains any clickable HTML elements, and if so mark it as a possible, *unconfirmed user-browser interaction*; otherwise we disregard u (no interaction).

Emulating user-browser interactions. Once a determination is made that an HTTP request was (highly likely) generated via user interaction, ClickMiner emulates that interaction via our browser plugin. Because it is difficult to infer from the traffic alone if the user had forced a page open in a new window/tab, we open each new page resulting from an emulated interaction on a separate window, and track the browser state and DOM concurrently for all open pages.

Challenges. There are a number of practical complications that make it difficult to locate the URL *u* within the DOM. For example, *u* may have been dynamically generated via JavaScript as a product of the user clicking on a DOM element with an `onclick` or `onmousedown` attribute. In addition, *u* may have been requested as a result of the user's interaction with an embedded object, such as a Flash object. Lastly, *u* may have been requested from a page satisfied from the browser cache, making the page containing *u* unsearchable as it may not have been recorded in the trace. These scenarios are addressed in Sections 2.6.5 and 2.6.6.

Another challenge comes from the plethora of plugins and scripting languages, which often issue asynchronous content requests. For example, consider a browsing session in which the user visits `www.google.com` and types a search keyword. As the user types, asynchronous requests will be sent to the server to update the page content with the (partial) search results. While the packet traces capture all requests, without knowledge of the semantics behind the asynchronous requests it is difficult to simulate the actions required in order to force the browser to make those same requests during replay. In turn, this means that we cannot easily replay those requests within the browser. As such, the effects of those requests (i.e, displaying of dynamically updated search results) will not occur during replay. We may therefore miss finding the element in the DOM (i.e., the search result link) on which the user eventually clicked. In the following sections we discuss how ClickMiner copes with some of these challenges.

2.6.5 Dynamic and Embedded Content

The algorithm described in Section 2.6.4 for finding user-browser interactions is not able, by itself, to reconstruct events for which the actual click is handled outside of the context of the page's DOM. For example, a click event may be handled via a JavaScript (JS) that dynamically compiles the URL of the next page to be loaded and triggers a `document.location` change. Correctly replaying all possible such cases is extremely challenging. What we describe in this section is a *best effort* approach to close some of the semantic gaps present in the network traces, thus further aiding the work of a forensic analyst.

JavaScript Mediated Clicks: One possible way to reconstruct clicks handled by JS would be to apply program analysis techniques to all scripts contained in the currently open pages. However, JS code embedded in highly dynamic pages can be very complex [49], and is often obfuscated to protect it from straightforward reverse engineering. Therefore, instead of leveraging program analysis techniques, we take a lightweight "network-oriented" approach to *approximately* reconstruct the clicks. The idea is to identify all clickable elements in a page that subscribe to events such as `onclick`, `ondoubleclick`, `onmousedown`, etc., and "test" them to check whether any of these elements leads to a particular URL. In practice, we proceed with emulating clicks to each one of these elements in the current page DOM under analysis. If, as the result of a particular click, the browser requests the next expected URL (i.e., the URL of the candidate interaction currently being considered), our proxy will simply serve the response. However, if the click causes the browser to request an URL that does not match the expected URL, the proxy will serve a `204 No Content` response, thus preventing the browser from inadvertently loading an "undesired" page. Still, there exists a challenge due to the fact that clicking on certain elements may occlude parts of the page, thus in a way changing the "state" of the page itself. However, our emulated clicks are based on programmatically "activating" DOM elements, and in practice the visual state of the page is often irrelevant.

Embedded Object Mediated Clicks: Common third-party browser plugins such as Flash, Silverlight, or Java, allow complex content to be rendered and controlled outside of the scope of a page's DOM. As Flash is arguably the most common plugin, ClickMiner attempts to detect clicks on Flash objects (it is worth recalling that we are only concerned with clicks that cause the browser to load a new page). We do so by analyzing the `FLASHVARS` parameter typically passed to embedded Flash objects. For example, Flash-based ads often extract the destination URL to be loaded upon an ad click from `FLASHVARS`, to enable efficient ad reuse on multiple pages. Hence, we attempt to detect clicks on Flash objects by parsing the variables passed via `FLASHVARS` and searching for a URL that (approximately) matches a candidate interaction's URL.

2.6.6 The Effect of the Browser Cache

ClickMiner's reconstruction capabilities are limited to the information contained within the recorded HTTP flows. This presents a problem if a click was satisfied from the browser's cache. ClickMiner implements a best effort approach to reconstruct interactions dependent upon missing traffic.

Assume that a user visited a page A, where she clicked on an element which took her to page B, then clicked on an element of B that brought her to a new page C, and finally clicked on an element of C to go to page D. Suppose that the request for B was satisfied directly from the cache. In this simplified example, ClickMiner would first process page A, retrieving it from the trace and loading it into the browser. After the browser reaches a resting state, the proxy would suggest the next *candidate interaction* URL. Since no record of the request to page B exists within the trace, the first interaction will be missed. Instead, ClickMiner's proxy will suggest the URL of page C, C_{url}. Consequently, ClickMiner inspects A's DOM searching for C_{url}, without finding it. At this point, ClickMiner loads C_{url} anyway in a new browser window, and marks C_{url} as an *unconfirmed click*. Finally, ClickMiner would retrieve D's URL, D_{url}, as the next candidate interaction, inspect C's DOM to (likely) match the element of C pointing to D_{url}, emulate a click on that element, and mark D_{url} as a *confirmed click*.

In general, though user-browser interactions that are not reflected in the network trace cannot be reconstructed with certainty, ClickMiner may still be able to infer them, as we discuss in Section 2.7.2. More importantly, missing one interaction does not jeopardize the reconstruction of other interactions that can be recovered from farther along the trace.

Other sources of missing traffic: Packet loss, corrupted packets, and encrypted traffic represent other possible sources of traffic missing from the traces. The effect of such missing information on ClickMiner's results is similar to the effect of the browser cache.

2.7 Click Graph Analysis

In this section, we describe how we can analyze the results of ClickMiner by building a *click graph* (Section 2.7.1). Furthermore, we explain how we can combine ClickMiner's click graph with the referrer graph (see Section 2.4.1) to infer additional user-browser interaction that may have been missed during the in-browser replay process (Section 2.7.2).

2.7.1 Building the Click Graph

We represent a user-browser interaction reconstructed by ClickMiner as a tuple (p, e, q), where q is an HTTP request, p is a web page URL, and e is a DOM element in the (rendered) page p that when clicked caused the browser to issue a request for q. Let $m = \{(p_i, e_i, q_i)\}_{i=1...n}$ be the list of all user-browser interactions *mined* by ClickMiner via in-browser traffic replay, as explained in Section 2.5 and Section 2.6. We build a directed acyclic graph $C = (V, E)$, where each node in V represents a "click tuple" from the list m. A directed edge $((p_w, e_w, q_w) \rightarrow (p_y, e_y, q_y)) \in E$ exists if page p_y was reached as a consequence of request q_w, which in turn was issued by emulating a user click on an element e_w of page p_w, for example.

Notice that the list m of interactions reconstructed by ClickMiner includes both *confirmed* and *unconfirmed* clicks (defined in Section 2.6.4). Remember that an *unconfirmed* click U is a

reconstructed interaction for which ClickMiner was not able to find (i.e., confirm) the existence of a related clickable DOM element. Formally, an unconfirmed click can be represented by a node $U = (p_u, e_u, q_u)$, where p_u and e_u are unknown values (in practice, p_u and e_u are *null*). However, in some cases the page of an unconfirmed click can be inferred from the referrer header field of q_u, as discussed below in Section 2.7.2.

The click graph C may contain a node $R_j = (null, null, q_{r_j})$ (or more than one) related to a request q_{r_j} that may have been issued directly, without clicking on a page element. For example, we would have such a "root node" if the user types the URL of a page directly in the browser address bar, or clicks on a bookmarked link. In general we call R_j a root node if during traffic replay ClickMiner was not able to find any page and DOM element that would lead to the node's HTTP request q_{r_j}, and if q_{r_j} did not carry a `Referer` field. Thus, a node can be considered as a "root" if it was mined as an unconfirmed interaction, and if no referrer is carried in the related HTTP request.

2.7.2 Augmented Click Inference

Ideally, the click graph C will include the entire sequence of clicks a user made during a browsing session. However, there are some situations in which ClickMiner may fail to detect a user-browser interaction (see Sections 2.6.5 and 2.6.6), thereby causing the generation of an incomplete click graph. To recover from some of the missing clicks, we augment the click graph produced by ClickMiner with information extracted from the HTTP referrers in the network trace, as follows.

Assume $U = (p_u, e_u, q_u)$, with $p_u = null$ and $e_u = null$, is an unconfirmed interaction mined by ClickMiner, and let ref_u be the referrer carried by q_u. In this case, we can infer that p_u should be equal to ref_u. Therefore, we can replace U with $U' = (\text{ref}_u, null, q_u)$. At this point, if the click graph C contains a node $Y = (p_y, e_y, q_y)$ where q_y's URL equals ref_u, we can draw an edge $Y \rightarrow U'$. In other words, we inferred that the page p_u was missing in the click graph C, and we were able to derive it by leveraging referrer information, as shown in the example Figure 2.4(a).

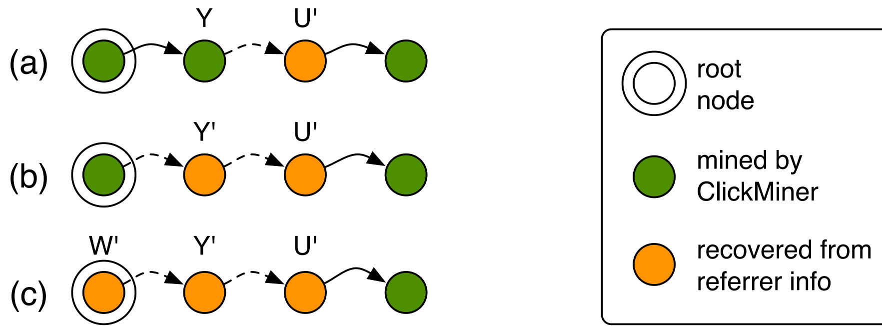

Figure 2.4: Examples of augmented click graph.

Assume now that ClickMiner failed to mine the node Y. That is, $Y = (p_y, e_y, q_y)$ is missing from the click graph C. Also, suppose that the network trace contains an exchange (q_y, r_y), for which q_y's URL equals ref_u. In this case, we can further infer that the click graph C should have contained a node $Y' = (p_y, e_y, q_y)$, where p_y and e_y are unknown. We can repeat the process described above until the last inferred node can be connected to an existing node in C, as shown in Figure 2.4(b), or a "root node" $W' = (null, null, q_w)$ is recovered from referrer information, whereby q_w carries no referrer, as shown in Figure 2.4(c).

We can also apply some of the pruning techniques used for RCI (Section 2.4.1) to the augmented click graph, thus further refining it. For instance, we can apply ad and social widgets pruning to the augmented click graph in a manner similar to that described in Section 2.4.1. In addition, the referrer delay pruning can also be applied to the augmented click graph, allowing for a direct comparison between ClickMiner and RCI, which we discuss in Section 2.8 (see Figure 2.5).

Differences w.r.t. RCI. It is worth noting that, unlike the RCI method (Section 2.4), ClickMiner selectively leverages referrer information *only to fill in some gaps* between interactions in the click graph that were mined via in-browser traffic replay. On the other hand, RCI "naively" uses the entire set of referrers found in the traffic traces.

2.8 Evaluation

To evaluate and compare ClickMiner and RCI, we conducted a user study[1] involving 21 different participants. All user browsing traces we collected during this study are available at `http://clickminer.nis.cs.uga.edu`, along with our prototype implementations of RCI and ClickMiner.

All our experiments were conducted with Firefox. However, it is important to notice that our implementation of ClickMiner is based on *Selenium-WebDriver* [4], which in turn is compatible with most major browsers, including some mobile versions. Therefore, with only minor adjustments ClickMiner could be used to replay web traffic within other browsers, for example selected according to the user-agent string that appears in the recorded network traces.

2.8.1 Recording User-Browser Interactions

For our user study we recruited 21 subjects among the undergraduate and graduate students, and staff members at the University of Georgia. Each subject was asked to *freely browse* websites of their choosing, within only few "privacy-preserving" restrictions (e.g., we prohibited the subjects from logging into any site containing personally identifiable information, such as GMail, Facebook, online banking sites, etc.). Participants were assigned a browsing time slot of about 20 minutes, during which they visited a large variety of sites, including many highly dynamic ones such as `amazon.com`, `youtube.com`, etc.

Each user interacted with an instrumented Firefox browser that allowed us to record most UI-level user-browser interactions. For example, every time the user performed a mouse click within a page, we recorded a tuple $(t_e, \text{page}_e, \text{elem}_e, \text{dst}_e)$ containing the timestamp of the click event e, the URL of the main page where the click happened, the DOM element that was clicked, and the destination page URL. We also recorded key-press events. At the same time, we recorded the

[1] The user study was approved by our university's IRB.

related full packet traces, and a video of all UI events (essentially, a video of the entire Desktop). A few users offered to record more than one 20-minutes browsing session, and overall we collected 24 different browsing traces that we used in our experiments.

We split the users into two roughly equal groups: Group1 and Group2. The users in Group1 were assigned a browser whose page caching functionalities were completely disabled, whereas users in Group2 used a browser with default caching settings. In practice, each user in Group1 was assigned a "fresh" virtual machine (VM) image with a fresh instrumented no-cache browser instance. On the other hand, users in Group2 shared the same browser instance. Namely the second user in Group2 was assigned the browser instance previously used by the first user in Group2. Similarly, the third user was assigned the browser instance previously used by the first and second users in Group2, etc. We did this to enable the evaluation of ClickMiner and RCI with and without a "warmed up" browser cache.

Table 2.1 and 2.2 report the number of HTTP requests and recorded user-browser interaction for each trace (a complete description of the table content is given in Section 2.8.2).

2.8.2 Reconstructed User-Browser Interactions

RCI Results

To evaluate the RCI method and compare it to ClickMiner, we first built a referrer graph (see Section 2.4.1) from each of the network traces recorded during our user study, and then pruned the graphs as explained in Section 2.4.1. Assume that $v_i = (q_i, r_i)$ is a node in the pruned RCI graph generated from a network trace. We compare the URL of request q_i with the set of user-browser interactions recorded during the related user browsing session. For example, if q_i matches the destination dst_e of a click event e, we consider q_i as a *true positive*, and mark event e as "consumed" to avoid matching the same recorded interaction more than once. On the other hand, if q_i does not match any of the recorded user-browser interactions, we label it as a *false positive*.

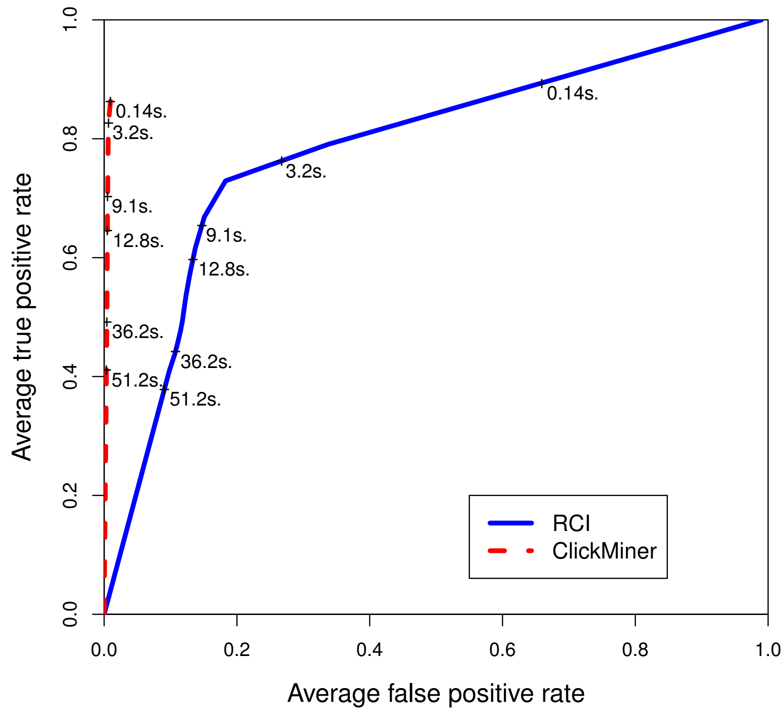

Figure 2.5: Trade-off between true and false positives.

The blue solid curve in Figure 2.5 shows the trade-off between the average true and false positive rates obtained by varying the referrer delay threshold, θ_δ. Per each trace, the true positive rate is computed as the fraction of user-browser interactions recorded during a user's browsing session that are represented in the pruned referrer graph, whereas the false positive rate is the fraction of graph nodes that do not match any recorded interaction. The average is computed across all 24 available user browsing traces.

ClickMiner Results

To evaluate ClickMiner, for each user browser session, we performed in-browser traffic replay as explained in Section 2.6. Then, we built the click graph, as explained in Section 2.7. Given the click graph, we match each node in the graph with the user-browser interactions recorded

during our user study, in a way similar to what we discussed in Section 2.8.2 for the RCI method. Tables 2.1 and 2.2 report the results obtained using ClickMiner with the augmented click graph (see Section 2.7.2), but without any referrer delay pruning. On the other hand, the dashed red line in Figure 2.5 reports the average trade-off (across all traces) between true and false positives obtained by pruning the augmented click graph at different referrer delay thresholds, so that we can directly compare the results to those produced by RCI.

In all tables, the *Trace Number* is simply a trace identifier; *HTTP Requests* is the number of requests in the captured network traces; *Recorded Clicks* is the number of user-browser interactions recorded via the instrumented browser used by the participants of our user study; *Mined Clicks* indicates the number of user-browser interactions inferred as explained in Section 2.7.2, averaged across five runs per trace; *Matching Clicks* indicates the number of mined clicks that match a recorded click. Finally, *TPR* is the true positive rate, i.e., the percentage of recorded interactions that match an interaction inferred by our system, whereas *FPR* is the false positive rate, i.e., the number of mined clicks that do not match any recorded interaction.

It is worth noting that, to produce the results in Table 2.1 and 2.2, we replay each user browsing trace for five times, and compute the average and standard deviation for the click mining results obtained per each run. We do this because every time a traffic trace is replayed within the browser, the rendering of highly dynamic pages may introduce slight changes, such as the automatic generation of some new HTTP requests via JavaScript[2]

By comparing the average true and false positive rates reported in Tables 2.1 and 2.2, we can see that the browser cache certainly has an impact on the ability of ClickMiner to reconstruct user-browser interactions. For example, when the cache is disabled (Table 2.1), ClickMiner achieves more than 90% TRP in 7 out of 13 traces, and 100% in 3 traces. On the other hand, only 3 out of 11 traces recorded with the cached enabled can achieve more than 90% TRP. Nonetheless, even in the

[2]To reduce execution time, in our current evaluation we did not turn on ClickMiner's system components describe in Section 2.6.5 that deal explicitly with dynamic and embedded content.

Table 2.1: ClickMiner results - Group1 (no-cache)

Trace Number	HTTP Requests	Recorded Clicks	Mined Clicks avg (stddev)	Matching Clicks avg (stddev)	TPR	FPR
1	3925	21	50.80 (0.40)	20.00 (0.00)	95.24%	0.79%
2	1114	25	39.00 (0.00)	25.00 (0.00)	100.00%	1.29%
3	2884	16	41.00 (0.00)	13.00 (0.00)	81.25%	0.98%
4	1030	10	16.00 (0.00)	10.00 (0.00)	100.00%	0.59%
5	3405	23	46.20 (0.75)	22.80 (0.40)	99.13%	0.69%
6	3800	21	51.60 (0.80)	19.00 (0.00)	90.48%	0.86%
7	4891	11	30.20 (0.40)	11.00 (0.00)	100.00%	0.39%
11	9247	37	75.00 (2.61)	32.20 (0.75)	87.03%	0.46%
14	6508	32	50.00 (1.10)	28.00 (0.00)	87.50%	0.34%
16	1167	32	28.60 (0.49)	22.00 (0.00)	68.75%	0.58%
18	4073	20	76.60 (1.50)	17.20 (0.40)	86.00%	1.47%
22	5005	23	51.40 (0.80)	21.00 (0.00)	91.30%	0.61%
23	722	14	15.00 (0.00)	11.00 (0.00)	78.57%	0.56%
Average	3674.69	21.92	43.95	19.40	89.63%	0.74%
Stddev	2350.46	7.88	18.21	6.60	9.58	0.34

cache enabled case, ClickMiner in average can retrieve more than 81% of user-browser interactions with less than 1.16% of false positives. For comparison, using the *non-augmented* click graph (i.e., without augmenting ClickMiner's output with referrer information), we obtained 70.47% TPR at 1.72% FPR, for the traces in Group2.

We also analyzed the percentage of confirmed and unconfirmed user-browser interactions (see Section 2.6.4) that ClickMiner was able to discover. On average, ClickMiner was able to reconstruct the actual DOM element clicked by the user for around 46% of the true positive interactions in Group1, and 48% in Group2.

ClickMiner vs. RCI Notice that the dashed red curve (ClickMiner) in Figure 2.5 does not reach the top-right corner because, unlike the RCI method, the number of user-browser interactions returned by ClickMiner is bound by what can be discovered through the in-browser replay process. Augmenting the click graph only fills some "gaps", as explained in Section 2.7. Nonetheless, at

Table 2.2: ClickMiner results - Group2 (cache)

Trace Number	HTTP Requests	Recorded Clicks	Mined Clicks avg (stddev)	Matching Clicks avg (stddev)	TPR	FPR
8	4786	28	64.40 (0.80)	21.00 (0.00)	75.00%	0.91%
9	2212	19	42.80 (1.60)	14.00 (0.00)	73.68%	1.35%
10	1639	15	23.20 (0.40)	15.00 (0.00)	100.00%	0.50%
12	1219	10	15.60 (0.49)	7.00 (0.00)	70.00%	0.71%
13	1250	15	17.00 (0.00)	13.00 (0.00)	86.67%	0.32%
15	500	34	34.20 (0.40)	28.00 (0.00)	82.35%	1.33%
17	4682	25	63.00 (0.00)	19.00 (0.00)	76.00%	0.94%
19	2239	21	38.00 (1.26)	19.20 (0.40)	91.43%	0.85%
20	3980	21	117.00 (1.26)	19.00 (0.00)	90.48%	2.48%
21	2312	18	60.60 (0.49)	16.00 (0.00)	88.89%	1.93%
24	943	22	28.40 (0.49)	14.40 (0.49)	65.45%	1.52%
Average	2342.00	20.73	45.84	16.87	81.81%	1.16%
Stddev	1428.86	6.33	28.11	5.10	10.61	0.64

low false positive rates, ClickMiner clearly outperforms RCI. For example, we would have to tolerate more than 20% FPR, for RCI to reach a TPR higher than what obtained with ClickMiner.

Execution Time A bottleneck in ClickMiner's performance stems from the use of a graphical web browser to replay HTTP traffic. GUI based web browsers generate a significant amount of I/O between the display device related to rendering their interface as well as all web pages loaded. While ClickMiner allows for visualizing the entire traffic replay on the browser, to perform our experiments we run the browser in *headless mode* (i.e. without a graphical interface), in which all I/O is simulated in memory via `Xvfb` [2], thus reducing processing time.

We performed our experiments with ClickMiner on an off-the-shelf desktop machine with an Intel Core i7-870 CPU and 8GB of RAM. Overall, ClickMiner required an execution time between approximately two to five times the length of the browsing session. The median execution time for the traces was about 76 minutes for traces in Group1 and 34 minutes for Group2. After inspection, we noticed that ClickMiner's execution time is dominated by inspecting the DOM of each relevant

web page rendered by the browser searching for the elements that were clicked by the users. The lower time needed to process traces in Group2 is likely due to the effect of the cache, because it lowers the number of HTTP requests to be considered as a possible interaction. In our future work we will investigate further optimizations that would allow us to reduce ClickMiner's execution time.

Discussion

While faced by several challenges due to highly dynamic content and caching, in practice ClickMiner is able to automatically reconstruct a large fraction of user-browser interactions with low false positives. We therefore believe our system can be a valuable aid to the forensic analysis of web traffic traces.

In the following, we discuss some common causes of false negatives and false positives that we identified through an analysis of the browsing traces used in our evaluation, which may inspire further improvements in future work. Furthermore, we discuss potential advantages that RCI may have, compared to ClickMiner.

Common causes of false positives. False positives are primarily represented by unconfirmed user-browser interactions reconstructed by ClickMiner as a result of HTTP requests that were not properly "consumed" during in-browser replay. For instance, assume the HTTP exchange (q, r) represents a web advertisement loaded through a JavaScript-driven request, and that response r contains clickable HTML content (the ad itself). Also, assume the user who generated the trace never actually clicked on the ad during her browsing session. It is possible that during ClickMiner's in-browser reply, request q will not be made, because the JavaScript running on the rendered page that was supposed to load the ad happens to dynamically construct a significantly different URL to be requested (e.g., for a different ad). Therefore q remains "unconsumed". At a certain point during trace replay, q may be considered as the next "candidate click", because it is an

outstanding, unconsumed request in the trace. As no DOM element can be found relating to q's URL, ClickMiner will render r in a new window, and classify q as an unconfirmed click.

Common causes of false negatives. An example of "missed interactions" that is prevalent within our user study involves Google searches. The search results page is built dynamically as the user types, via asynchronous requests. ClickMiner is not able to infer all these UI events from the network traces, and therefore it is difficult to have the browser replay the same asynchronous requests. Consequently, the "replayed" DOM will be different from the one on which the user originally clicked (see related discussion in Section 2.6.4), and we will not be able to find the DOM element needed to mine a confirmed interaction.

Another source of false negatives is represented by exchanges that are simply missing from the trace, because of the caching effect. In addition, while more rare, it may also happen that an exchange is erroneously consumed during replay, as a consequence of ClickMiner's approximate URL matching process (see Section 2.6.2). This may "steal" the HTTP exchange from a correct match to an actual user interaction.

Advantages of RCI While ClickMiner outperforms RCI in terms of reconstruction accuracy, the RCI method has the advantage that no in-browser traffic replay is necessary, and that it can work on incomplete traffic traces that only record the header of HTTP requests, rather than requiring the full request and response content. Also, the RCI method is more efficient than ClickMiner's in-browser traffic replay process. As expected, this represents a natural trade-off between efficiency and reconstruction accuracy.

2.9 Case Studies

We now describe the application of ClickMiner to real-world examples of security incidents involving a malware download.

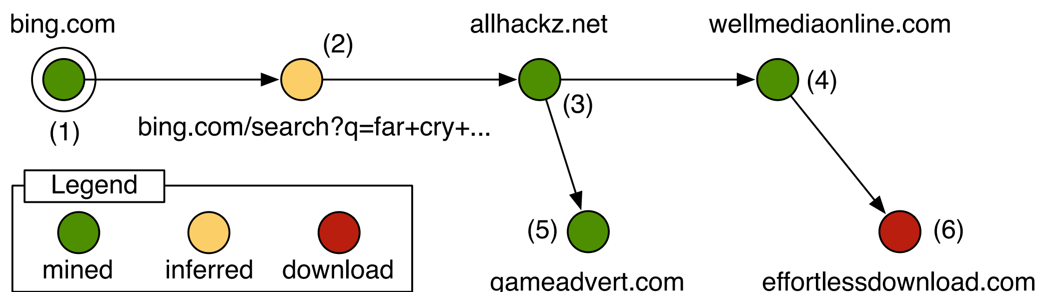

Figure 2.6: Reconstructed click graph (case study 1)

2.9.1 Case Study 1

During our user study, we were able to separately collect a network trace for the following browsing session. A user visited the popular search engine `bing.com`. Next, she performed a search using the terms "`far cry 3 hackz tools crack`". From the search results, the user clicked on a link to a page hosted on the site `allhackz[dot]net`. Then, she clicked on a download button, resulting in the browser opening two pages, hosted at `gameadvert[dot]com` and `wellmediaonline[dot]com`, respectively. Lastly, from `wellmediaonline[dot]com` the user's browser was redirected to `effortlessdownload[dot]com`, from which an executable file was ultimately downloaded[3]. We submitted the downloaded file to `VirusTotal.com`, where it was detected as a malware by 23 out of 51 AVs.

Summary of Results: By running ClickMiner on the network trace recorded during the user-generated browsing session described above, we were able to reconstruct the *full chain of user-browser interactions* that led the user to the malware download. In particular, ClickMiner reduced the amount of information to be analyzed by a forensic analyst *from 316 HTTP exchanges to only 6 click graph nodes*, as shown in Figure 2.6. Additionally, ClickMiner was able to correctly identify what element the user clicked on to initiate the malicious download. Conversely, RCI did not

[3]MD5: `c94f917fdc39dfb7245ebdd674b2bdf8`

perform as well. Even by adjusting the filtering and referrer-delay threshold to reduce RCI's false positives without missing any of the interactions, RCI produced a graph with 79 nodes, instead of only 6.

Details: Figure 2.6 shows that `bing.com` (1) was identified as the root. Node (2) is related to the `bing.com` page that shows the search results. This node is colored yellow because its presence in the click graph was inferred through the referrer reported by the next page request on `allhackz[dot]net`. The reason why node (2) is *inferred*, rather than directly reconstructed through ClickMiner's traffic replay, is that `bing.com` populated the search results page dynamically as the user typed the search terms, using asynchronous HTTP requests. As mentioned in Section 2.6.4, it is difficult to replay this type of traffic without knowledge of the semantics of the response content (e.g., a json object). Nonetheless, ClickMiner was able to infer that the user clicked on a search result link to access `allhackz[dot]net` (3). In addition, through the replay of the response for node (3), ClickMiner was able to correctly identify the DOM object that the user clicked on to reach node (4). More precisely, using the *click emulation* approach described in Section 2.6.5, ClickMiner identified the location in the DOM of the download button that would fire an event and call a JavaScript that opened page (4). Finally, while the user's browser was automatically redirected from node (4) to the actual file download hosted on node (6), the page on node (4) also contained two `<a>` tags linking to the URL of node (6). Therefore ClickMiner connected the two nodes and essentially reported that the user likely clicked on one of those two links. While this is not completely accurate, because the download happened through an automatic redirect, the click graph produced is correct. Notice also that if the page on node (4) contained no explicit hyperlink to (6), ClickMiner could still have inferred the link between the two pages through the referrer information, for example.

In summary, we can see that ClickMiner can reconstruct the "click path" to the malware download with only some minor deviation from what the user precisely did to download the file.

2.9.2 Case Study 2

To further evaluate how ClickMiner can aid a forensic analyst, we obtained a set of real-world network traffic traces containing malware downloads *passively collected* from a *live* large academic network. These traces were provided by a network security company specializing in network-based malware detection. Each trace in the set contained a sequence of HTTP exchanges recorded during a small time window preceding (and including) the download of a malicious executable file. Not all HTTP responses in the traces were fully recorded, and in some cases only the response headers were available. This was especially the case for exchanges recorded farther back in time with respect to the download event. While the missing response content represents an obstacle for ClickMiner, it allowed us to evaluate how our systems can cope with this type of missing traffic.

We now briefly describe the content of three of the traces we were able to obtain, which we manually analyzed to enable the comparison with the results automatically produced by ClickMiner. We then summarize the output ClickMiner produced by analyzing these traces. Notice that the following traces are related to three different (anonymized) real-world network users. For brevity, we only describe events related (or "on path") to the malware downloads.

Trace 1: A user visited `www.google.com` to perform a search. The exchanges related to the search are not recorded in the traces, most likely because they were performed via HTTPS. However, as the user clicked on a search result, she landed on the help page of the `thepiratebay[dot]sx` website (`google.com` was reported in the referrer of that request). From there, the user proceeded to visited a page at `bitlordapp[dot]com` where a malicious executable file was downloaded[4].

Trace 2: A user visited the search engine `www.bing.com` and performed a search for the terms "`free bejeweled 3 online`". Next, the user visited a page at `iwin[dot]com` (listed in the search results). Then, the user downloaded a malicious executable file from `dl.iwin[dot]com`[5].

[4]MD5: `e64bef0c045430dfdbc02a824cd19003`
[5]MD5: `25b1d21a555bbd05856555a01b5be2b4`

Trace 3: A user visited `www.yahoo.com` (likely performing a search). The user then visited a page at `securefiledl[dot]com`. Lastly, the user clicked on a link to download an executable file from the host `oi-installer9[dot]com`[6].

Summary of Results: We applied ClickMiner to process the traces described above to automatically reconstruct the click paths that preceded each malware download. Via extensive manual investigation, we determined that that click paths should contain a total of 4 user-browser interactions for Trace 1, 3 for Trace 2, and 3 for Trace 3. Overall, Trace 1 contained 1351 HTTP exchanges, from which ClickMiner derived a click path containing only 7 nodes, including all the 4 expected interactions. Trace 2 contained 634 HTTP exchanges. From this trace, ClickMiner generated a click path preceding the download containing 4 interactions, of which 3 matched the expected (manually confirmed) ones. Trace 3 contained 882 HTTP exchanges. ClickMiner produced a click path that exactly matched the path we had derived manually.

In summary, ClickMiner was able to correctly or very closely derive the chain of interactions by which the user downloaded each malicious executable. In the cases above, the traffic corresponding to each search was not fully recorded; yet, ClickMiner was able to infer the user-browser interactions related to the search pages by constructing an augmented click graph (see Section 2.7). In addition, ClickMiner was able to reduce the input traffic from several hundreds HTTP exchanges, to only few that are highly related to the security incident being analyzed.

2.10 Limitations and Future Work

One of ClickMiner's primary applications, as demonstrated in Section 2.9, is the after-the-fact replay of user-browser interactions before and up to the occurrence of a security incident. However, potential attackers could seek to thwart our analysis by reducing the ability to replay the recorded

[6]MD5: `2d484f0614b1d720dfbccdd788a3ad9c`

incidents. The following are some examples of possible techniques an attacker could use to this end.

An attacker could leverage ClickMiner's limited ability to infer interactions with plugins as a means of inhibiting replay of user behavior. By forcing the majority of user interactions to be handled via a plugin, an attacker could prevent those interactions from being correctly replayed while still performing a successful attack. It is also possible that an attacker could embed a script within an attack page designed to detect the presence of ClickMiner during replay, for example by checking for the presence of our Selenium-based browser instrumentation (either directly or via artifacts). Upon detection, the JavaScript could selectively alter the content of the page (e.g. removing DOM elements, changing the `href` values of all `a` tags, etc.) to prevent certain user-browser interactions from being correctly reconstructed during replay.

Notice, however, that ClickMiner could still be successfully used to reconstruct the user-browser interactions that occurred *before* the inception of the attack, thus enabling an analysis of the *web path* followed by users who eventually fall victims to attack pages on the web. In our future work, we plan to study how ClickMiner's limitations can be mitigated, thus making it harder for the attacker to prevent a detailed analysis of the steps followed by the users after they reach the attack's "entry point".

2.11 Related Work

Traffic Replay: While a number of traffic replay systems have been proposed in the past [1, 28], most tools have focused on replaying traffic at an IP-level or TCP/UDP-level granularity. For example, Hong et al. perform interactive replay of internet traffic [28] by emulating a TCP protocol stack. ClickMiner is different in that it implements in-browser replay of application-level (HTTP) traffic.

In [12] the authors describe WebPatrol, a system for automated collection and replay of malware infection scenarios. Importantly, ClickMiner's purpose is very different and much more general, compared to WebPatrol. In fact, while WebPatrol is limited to replying malware infection scenarios that can be automatically collected by its own "honey clients", ClickMiner aims to reconstructing user-browser interaction from real-world traffic traces. Furthermore, ClickMiner is not limited to reconstructing events related to malware infections, and can instead aid the forensic analysis of other security incidents involving user-browser interactions, such as social engineering attacks, phishing, etc.

Traffic Analysis Tools: Over the last several years numerous network forensic analysis tools have been constructed [43]. These tools allow administrators to monitor, capture, analyze, and in some cases replay network traffic in order to aid in network crime investigations (i.e. malware infections, denial of service, etc.) and help generate appropriate incident responses. To the best of our knowledge, ClickMiner is the first tool that can automatically reconstruct detailed user-browser interactions from web traffic traces, and it could therefore be used as a component of a more comprehensive network forensic analysis framework.

Web Usage Mining: Web usage mining has been studied for example in [62, 33, 23]. Wu et al. [62] proposed a system named SpeedTracer which applies data mining techniques to web server logs in order to extract and group user interaction paths. Etminani et al. [23] propose the application of Kohonen's Self Organizing Maps to preprocessed web logs for extracting common interaction patterns. Also, ReSurf [64] aims to reconstruct web surfing activities from traffic traces via an analysis of referrer headers. Our work takes a different approach, because it aims to reconstruct user browsing activities from recorded network traffic via in-browser replay. Our approach has the advantage of mining complex cross-site activities invisible to techniques that rely on web server logs, and outperforms the RCI approach based on ReSurf [64], as we showed in Section 2.8.

2.12 Conclusion

In this paper, we discussed the importance of aiding the forensic analysis of web traffic traces, for example to help in the investigation of the user-browser interactions that take place right before a security incident. To this end, we proposed a novel system for reconstructing user-browser interactions from network traces that we named ClickMiner. Through a user study, we show that ClickMiner can correctly reconstruct between a $\simeq$ 82% and $\simeq$ 90% of user-browser interactions with low false positives, and that it outperforms a previously proposed referrer-based approach.

2.13 Acknowledgments

We thank Ling Huang and Xiaoning Li from Intel for their support and collaboration, and the anonymous reviewers for their helpful comments. This material is based in part upon work supported by the National Science Foundation under Grants No. CNS-1149051 and ACI-1127195. This material is also partially supported by a gift from the Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation nor Intel.

Chapter 3

WebCapsule: Towards a Lightweight Forensic Engine for Web Browsers

3.1 Abstract

Performing detailed forensic analysis of real-world web security incidents targeting users, such as social engineering and phishing attacks, is a notoriously challenging and time-consuming task. To reconstruct web-based attacks, forensic analysts typically rely on browser cache files and system logs. However, cache files and logs provide only sparse information often lacking adequate detail to reconstruct a precise view of the incident.

To address this problem, we need an *always-on* and *lightweight* (i.e., low overhead) forensic data collection system that can be easily integrated with a variety of popular browsers, and that allows for recording enough detailed information to enable a full reconstruction of web security incidents, including phishing attacks.

To this end, we propose *WebCapsule*, a novel *record and replay forensic engine* for web browsers. WebCapsule functions as an always-on system that aims to record all non-deterministic

Neasbitt, C., Li, B., Perdisci, R., Lu, L., Singh, K. and Li, K. 2015. WebCapsule. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15. (2015), 133-145.

inputs to the core web rendering engine embedded in popular browsers, including all user interactions with the rendered web content, web traffic, and non-deterministic signals and events received from the runtime environment. At the same time, WebCapsule aims to be *lightweight* and introduce low overhead. In addition, given a previously recorded trace, WebCapsule allows a forensic analyst to fully replay and analyze past web browsing sessions in a controlled isolated environment.

We design WebCapsule to also be *portable*, so that it can be integrated with minimal or no changes into a variety of popular web-rendering applications and platforms. To achieve this goal, we build WebCapsule as a self-contained instrumented version of Google's Blink rendering engine and its tightly coupled V8 JavaScript engine.

We evaluate WebCapsule on numerous real-world phishing attack instances, and demonstrate that such attacks can be recorded and fully replayed. In addition, we show that WebCapsule can record complex browsing sessions on popular websites and different platforms (e.g., Linux and Android) while imposing reasonable overhead, thus making always-on recording practical.

3.2 Introduction

The ability to perform accurate forensic analysis of web-based security incidents is critical, as it allows security researchers to better understand past incidents and develop stronger defenses against future attacks. Unfortunately, analyzing real-world web attacks that directly target users, such as social engineering and phishing attacks, remains an extremely challenging and time-consuming task.

The state-of-the-art methods for reconstructing web-based incidents generally follow two approaches. The first approach relies on analyzing the web browser's history, cache files, and system logs [39, 31]. However, cache files and logs provide only sparse information often lacking adequate detail to reconstruct a precise view of what happened during social engineering and phishing attacks that may have occurred days in the past. The second approach leverages access to full net-

work packet traces, which may provide some indications of how an incident unfolded. However, the complexity of modern web pages results in a large *semantic gap* between the web traffic and the detailed events (e.g., page rendering, mouse movements, key presses, etc.) that occurred within the browser [37]. Such semantic gaps make it very difficult to precisely reconstruct what a victim actually saw and how she was tricked, and to identify what information was consequently leaked.

To address this problem, we need a forensic data collection system that satisfies the following requirements:

- be *always-on*, so that all (unexpected) incidents can be *transparently* recorded, including new attacks that follow previously unknown patterns;
- be *lightweight*, to minimize performance overhead, thus making always-on recording practical;
- be *portable*, to operate in a variety of web-rendering applications and platforms;
- provide critical information to greatly enhance and *facilitate a forensic analyst's investigation of web security incidents*, with particular focus on *attacks that directly target users*, such as social engineering and phishing attacks.

In this paper we propose *WebCapsule*, a novel *record and replay forensic engine* for web browsers. WebCapsule lays the foundations for web-based attack reconstruction and analysis while meeting all of the above stated requirements. Our main goal is to enable an always-on, transparent, and fine-grained recording (and subsequent replay) of potentially harmful web browsing sessions. As depicted in Figure 3.1, WebCapsule aims to record all non-deterministic inputs to the core web rendering engine embedded in the browser, including all user interactions with the rendered web content, web traffic, and non-deterministic signals and events received from the runtime environment.

WebCapsule allows an analyst to later replay previously recorded browsing sessions in a separate controlled environment, where no new external user inputs or network transactions are needed. This enables detailed analysis of security incidents that are (obviously) unexpected, and allows for

reconstructing detailed information about incidents that may follow new, never-before-seen attack patterns. In addition, by replaying all non-deterministic inputs, including all content provided by the server, WebCapsule enables a full forensic investigation of incidents involving *ephemeral web content*, such as short-lived phishing or social engineering attack pages.

While some previous work has studied record and replay to assist the debugging of web applications [11, 7, 48], these studies do not focus on forensic analysis and, more importantly, do not satisfy the associated requirements listed above. For example, TimeLapse [11] is a debugging tool based on Apple's WebKit [59] that allows for recording and replaying web content. However, TimeLapse does not work as an "always on" system. Also, TimeLapse does not allow for transparent recording because it deeply modifies the internals of WebKit, for example to force a synchronous scheduling of threads such as the HTML parser thread [52], thus also impacting performance. In addition, TimeLapse currently only works on MacOS+Safari+WebKit [53], and is not easily portable to other operating systems and browsers. Conversely, WebCapsule can function as an *always-on* system (e.g., it can be configured to start recording at browser startup with no user intervention) to continuously and *transparently* record browsing sessions while introducing low overhead. Furthermore, WebCapsule is highly portable, can be embedded in a variety of web-rendering applications, and can run on a variety of platforms. Furthermore, unlike [7, 48], WebCapsule is not limited to only recording user interactions with web pages, but instead aims to record all non-deterministic events needed to fully replay browsing traces, including all previously rendered web content, without incurring high performance overhead (we further discuss differences with previous work in Section 3.8). This is very important for forensic analysis. In fact, most malicious web pages (e.g., phishing websites) are *short lived*. Therefore, because [7, 48] only record user interactions with pages without recording all other non-deterministic inputs (e.g., network traces, timing information, etc.), they do not enable an after-the-fact replay and investigation of security incidents. WebCapsule solves this problem.

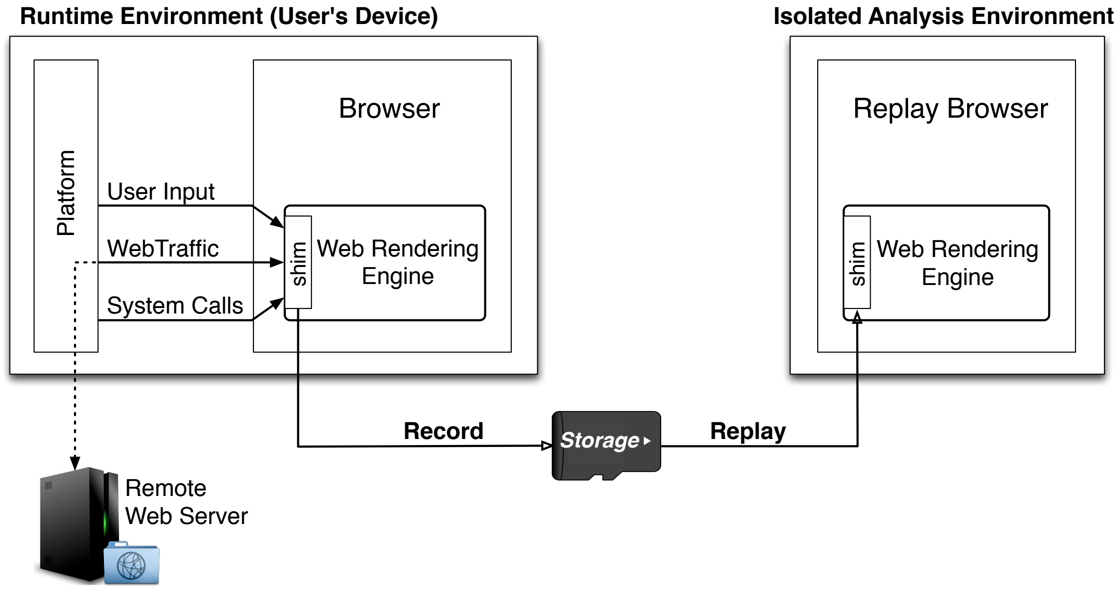

Figure 3.1: High-level overview of WebCapsule's record and replay capabilities. Non-deterministic inputs to the embedded web rendering engine are recorded, and can be fully replayed in an isolated forensic analysis environment where no new external user inputs or network transactions are received.

To make WebCapsule *portable*, so that it can be easily embedded in a wide variety of web-rendering software (e.g., different browsers), we design and implement it as a self-contained instrumentation layer around Google's Blink web rendering engine [9], which is already embedded in a variety of browsers (e.g., Chrome, WebView, Opera, Amazon Silk, etc.) and can run on different platforms (e.g., Linux, Android, Windows, and Mac OS). To implement our WebCapsule forensic engine, we inject lightweight (i.e., low overhead) instrumentation shims into Blink and its tightly coupled V8 JavaScript engine [56] (see Figure 3.2) in a way that allows us to inherit Blink's portability.

WebCapsule's portability has several advantages. Not only can it be readily deployed into existing Blink-based browsers and multiple platforms, but it also allows us to fully replay the

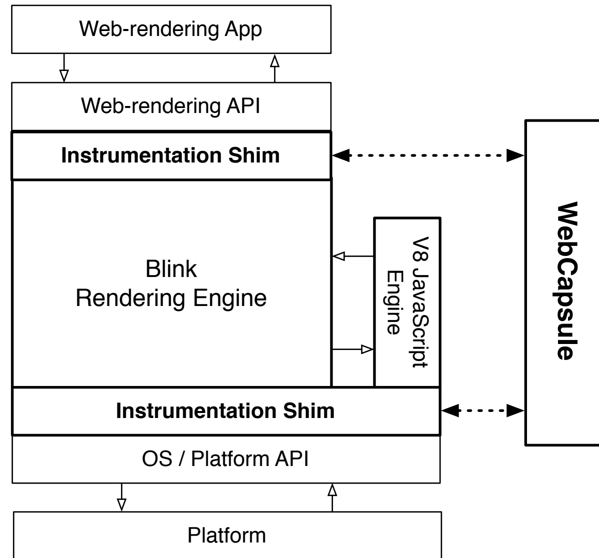

Figure 3.2: Overview of WebCapsule's instrumentation shims.

browsing traces on a device (or virtual machine) whose platform may differ from the platform where the traces were recorded.

At the same time, our design choice of "living" strictly inside Blink imposes a number of constraints that make the instrumentation process challenging, especially for enabling the replay of complex browsing traces. For instance, one of the main challenges we address is how to inject lightweight instrumentation shims without altering the rendering engine's application and platform programming interfaces (APIs), so that we can fully inherit Blink's portability (the challenges we encountered are discussed in more details in Sections 2.5 and 3.6). Moreover, Blink is highly multi-threaded, making the replay of complex browsing traces challenging (e.g., Blink's main thread, HTML parser, and JavaScript WebWorkers could be scheduled differently during replay). Nonetheless, we are able to address these challenges (see Section 3.6) and, in turn, we can record and replay complex browsing sessions, including on popular and highly dynamic websites (e.g., Facebook).

In summary, we make the following contributions:

- We propose WebCapsule, a novel *record and replay forensic engine* for web browsers that enables an *always-on* and *transparent* fine-grained recording (and subsequent replay) of web browsing sessions. To the best of our knowledge, ours is the first work towards creating such an always-on and yet *lightweight* (i.e., low overhead) forensic engine.
- We implement WebCapsule as a self-contained instrumentation layer around Google's Blink and V8 engines without changes to their application and platform APIs, and describe the technical challenges addressed by our solution. Thanks to this design, WebCapsule enables record and replay of web events for any web-browsing application built on top of Blink, making our forensic engine portable.
- We evaluate WebCapsule over numerous real-world phishing attack instances, and demonstrate that such attacks can be accurately recorded and fully replayed. Furthermore, we evaluate WebCapsule on different physical devices and platforms, including Linux and Android, and show that we can record complex browsing sessions on popular highly-dynamic websites while imposing reasonable overhead, thus making always-on recording practical.
- We plan to release our WebCapsule prototype system and a variety of browsing traces that we recorded for evaluation at `http://webcapsule.org`.

3.3 Problem Definition and Goals

In this section we discuss WebCapsule's goals using a representative example use case scenario. We also clarify the scope and non-goals for our work.

Representative Use Case. Assume that an employee, Alice, in a sensitive enterprise or government network, falls victim of a phishing attack. Alice unintentionally leaks credentials (e.g., user name, password, employer id, etc.) that allow the attackers to access confidential organizational information. After a number of days from the phishing attack, as anomalous data access patterns

are discovered, a forensic analyst may be called in to reconstruct a detailed picture of how the incident occurred, starting from the attack inception. Based on an analysis of the credentials used by the attackers to access sensitive information, the analyst suspects a small set of users, including Alice, may have leaked the credentials. At this point, the analyst would need to explore the detailed browsing history of these users, in an attempt to reconstruct how the credentials were actually leaked and learn how the phishing attack unfolded. In particular, in addition to regular browser and network logs, the analyst would like to reconstruct the users' interactions with webpages (e.g., mouse and keyboard inputs) and the browsers' view of rending events (e.g., layout and content changes), to gather detailed and essential information for attack analysis. Ultimately, this fine-grained analysis would allow the organization to understand how Alice was tricked into leaking the credentials, and how organizational security policies and user education can be improved to prevent future attacks.

WebCapsule's Goals. In the above example, WebCapsule's role is to collect the critical information that would enable the analyst to precisely reconstruct how the phishing attack unfolded and how the user was tricked. Specifically, WebCapsule aims to transparently record enough information to enable the forensic analyst to reconstruct a user's historic browsing activities that occurred within a certain time window of interest.

To meet the above goals, while the user is browsing the web WebCapsule *transparently* collects the following information:

- Every time a mouse click or keypress occurs, WebCapsule records the HTML corresponding to the underlaying DOM element that is the target of the input event. In addition, for all mouse clicks and "Enter" keypress events (which may initiate a page navigation or form submission), a snapshot of the current DOM tree is taken and stored. This happens right before the user input event is dispatched to any other browser components (e.g., the JavaScript engine, extensions, etc.) that could alter the DOM. This allows the forensic analyst to ana-

lyze exactly how the page was structured at every significant user interaction with the page's components.

- WebCapsule also aims to record all non-deterministic inputs to the rendering engine, including all input events (e.g., mouse location coordinates, keypress codes, etc.), responses to network requests, and return values from calls to the underlying platform API. All this information is transparently recorded and immediately offloaded to a data collection agent, which can then store it into an archive of historic browsing traces. Furthermore, WebCapsule allows the forensic analyst to later retrieve a browsing trace from this archive, reload it inside the rendering engine, and replay what the user did and saw on the browser.

Non-Goals and Future Work. In this paper, we only focus on laying the foundational work to enable "always on" transparent recording and replay of browsing traces with limited overhead.

Clearly, the data collected by our WebCapsule forensic engine may include very sensitive information, such as passwords, credit card numbers, other personal banking information, etc., whose confidentiality needs to be protected. In this paper we do not focus on protecting the confidentiality of the recorded information. Nonetheless, we believe this problem may be solved via a combination of approaches, as discussed below.

For instance, the forensic engine could maintain a whitelist of websites (e.g., online banking sites, healthcare-related sites, etc.) on which to avoid recording any information, thus following an approach similar to *SSL man-in-the-middle* web proxies commonly deployed for security and compliance reasons in sensitive enterprise networks. In addition, the user (or a system administrator, in corporate environments) may be allowed to customize such a whitelist, thus further improving the protection of potentially sensitive information.

An additional approach is to appropriately choose encryption primitives, which could be implemented by the software agent that concretely collects the recorded data from WebCapsule and stores them on disk. For example, a different key may be generated to encrypt different parts of the

browsing traces (e.g., one key per each new tab opened by the browser). The related decryption keys may be stored in a secure *key escrow*, and a specific key may be released only if a forensic investigation is properly authorized [18].

In this paper, we also do not focus on how to efficiently store the recorded traces to minimize storage use. However, we note that storage costs have been decreasing sharply, and that many enterprise networks already use commercial solutions to store full network packet traces for considerable periods of time [38], for security and compliance reasons. WebCapsule has the ability to *offload the recorded data in real-time* from the browser to a storage application. Thus, it may be possible to adapt current enterprise-level storage solutions to accommodate the recording of WebCapsule's browsing traces. In addition, in our future work we plan to study how the recorded data could be "aged" to reduce the granularity of historic traces, and measure the trade-off between the granularity of the recorded data and replay accuracy.

3.4 Approach and Challenges

Approach Overview. To make WebCapsule portable, we implement it by injecting lightweight instrumentation shims around Google's Blink web rendering engine [9] and the V8 JavaScript engine [56] without altering their application and platform APIs (see Figure 3.2). This allows us to inherit the portability of Blink and V8, effectively making WebCapsule platform agnostic. In addition, because Blink and V8 are at the core of several modern browsers (e.g., Chrome, Opera, Amazon Silk, etc.) and of Android WebView [60], WebCapsule can be readily integrated with minor or no code changes in a wide variety of web-rendering software.

Our design and implementation of WebCapsule aims to minimize the amount of instrumentation code added into Blink. To this end, our record and replay capabilities are implemented in large part by extending Chrome's DevTools [15], which provide access to the internals of Blink (see Section 3.5 for more details). This allows us to inject only thin instrumentation shims at

critical points in Blink's code without modifying any API, code interfaces (i.e., the members of Blink's classes) or data structures already implemented in Blink, thus obtaining a cleaner and more lightweight (e.g., low overhead) implementation of WebCapsule's functionalities.

It is worth noting that WebCapsule could be used to independently record and replay web content rendered in different browser tabs. More specifically, a browser that embeds Blink can spawn a new instance of the rendering engine for each separate tab, as it is done for example by Chromium's default process model[1] [44]. Consequently, each tab will also use its own independent instance of WebCapsule, and could therefore be recorded and replayed independently from other tabs. This is important for forensic analysis purposes, because the analyst may want to replay only a subset of the web content (i.e., only some tabs) visited by the user within a given time window.

Challenges. The design and implementation choices outlined above impose a number of hard constraints that we had to address. For instance, events such as a user's click on the "back button" on the browser toolbar cannot directly be recorded, because the "raw" input event is handled outside of Blink (by the browser's chrome). Instead, we had to reconstruct the series of side effects (in this example, navigation history manipulation) that are communicated to Blink, and record each of these effects at the location in Blink's code where the rendering engine "meets" the embedder browser application.

In addition, Blink is highly multi-threaded, making the correct replay of complex browsing traces challenging (e.g., the main Blink thread, HTML parser, and JavaScript WebWorker threads could be scheduled differently during replay). To compensate for these problems, we implement a number of mechanisms to "re-synchronize" Blink's clock (e.g., by adjusting the result of calls to `currentTime()`) and to precisely pair network requests with the correct response drawn from the recorded traces (we discuss these mechanisms in more detail in Section 3.6).

[1] In some corner cases, Chromium may still use the the same Blink instance to render content within inter-dependent tabs.

It is important to notice that any state information kept outside of Blink, such as the browsers' cache, cookie store, etc., do not represent a significant challenge, as they do not need to be explicitly recorded. In fact, such state information is accessed by Blink via well-defined web-rendering and platform APIs. Because WebCapsule can already record these API calls, it can reconstruct "external" state without a special record and replay component.

3.5 Recording: Design and Implementation

In order to implement WebCapsule's replay functionality, we must first provide the ability to record non-deterministic inputs into Blink/V8. For the sake of brevity, in the following we will refer to a function (or class method) as being *non-deterministic* if it accepts a non-deterministic argument (e.g., a user input event), or if it returns a non-deterministic value (e.g., a network response, the current system time, etc.). Otherwise, we say the function is *deterministic*. Notice, however, that these definitions are not intended to be rigorous, and simply provide a way to conveniently refer to functions that are used to pass non-deterministic events to the rendering engine, or that return non-deterministic values after an explicit call from the engine is made.

The non-deterministic inputs we record can be divided into three categories, as follows:

- Inputs that are injected by the embedder software (e.g., a browser) directly into Blink via the web-rendering API (see Figure 3.2). This category of non-deterministic inputs includes page control messages, such as scroll or resize the web page, as well as any user interaction and gestures via mouse, keyboard, or touchscreen interface.
- Information requested by Blink/V8 via synchronous calls to the underlying system. This information is requested via the platform API (see Figure 3.2) which abstracts the details of the underlying system upon which the embedder software is executing. Information in this category includes the current system time, the user-agent string describing the embedding software, total available memory, etc.

- Lastly, Blink can also request information from the platform API to be returned asynchronously via callback interfaces. Requests for remote resources, including network requests, are primarily handled using this functionality.

Recording Components. WebCapsule's recording functionality is implemented using two primary components, namely a special DevTools `InspectorAgent` (which we named `InspectorForensicsAgent` in our code), and wrappers for the platform API of both Blink and V8. In the following, we discuss in detail how these components can be used to record a user's browsing activities.

3.5.1 Extending DevTools

As mentioned in Section 3.4, one of our design goals is to create our instrumentation shims with as small a footprint as possible on the original codebase of Blink and V8. To this end, we implement a significant portion of WebCapsule's functionalities by extending Blink's built-in instrumentation facility known as DevTools [15]. This allows us to leverage existing quality code and at the same time minimize performance overhead.

DevTools is designed to provide developers with detailed insight into the execution of Blink/V8. The information DevTools provides is divided into categories based on functionality, including information about DOM elements, network traffic, and Javascript execution. The collection and presentation of information from each category is implemented via an `InspectorAgent`. Users can retrieve the desired information collected by DevTools using either a graphical interface (called 'Developer Tools' in Chromium), or via a JSON-based protocol over a WebSocket connection.

It is possible to extend the existing DevTools functionalities by "hooking" events one would like to listen to. As shown in the example in Figure 3.3, this can be done by modifying `InspectorInstrumentation.idl`, which is written using a mix of IDL and C++ code. This allows us to define a special `InspectorAgent`, which we use to add WebCapsule's instrumentation shims around the web-rendering API, as explained below.

```
[WebCapsule]
void handleInputEvent ( Page*, const WebInputEvent& );

[WebCapsule]
void handlePageScroll ( Page*, const WebSize& size,  double delta );

[WebCapsule]
void handleResize ( Page*, const WebSize& size );
```

Figure 3.3: Extending DevTools: instrumentation example (from `InspectorInstrumentation.idl`).

During the recording process, WebCapsule continuously offloads the recorded events to an external data collection agent, thus greatly reducing memory overhead for the rendering process. To allow for the communication between WebCapsule and the external agent, we extend the DevTools JSON-based network protocol[2].

3.5.2 Recording User Input

Most user inputs (e.g., mouse movements, gestures, and key presses) are sent to Blink via its `WebViewImpl::handleInputEvent()` API. A `WebInputEvent` parameter is passed to this function carrying high-level information describing the input instance, such as its type and location on the page. To record the input, we define an instrumentation shim called `handleInputEvent`, with WebCapsule's `InspectorAgent` declared as the shim handler agent. Consequently, during execution our shim is called for each user input event. When WebCapsule is running in *record mode* the `WebInputEvent` passed to the shim is copied and stored, so that it can be re-injected *as is* during replay (see Section 3.6). On the other hand, when WebCapsule is not set to operate in recording (or replay) mode, then all of its shims perform no operation, letting Blink function as if WebCapsule was not at all present.

[2]defined in Blink within `protocol.json`.

Target Element and DOM Snapshots. One of WebCapsule's goals is to provide the forensic analyst with a detailed recording of the state of the page at critical moments during the user's browsing experience. To this end, every time an input event occurs we also record the URL of the page where the event occurred. In addition, for all key presses and mouse clicks, we record the HTML representation of the element in the DOM tree that is the target of the user input. Furthermore, for events that are the main "cause" of a page transition, such as a mouse click or an "Enter" keypress (which may trigger a form data submission), we also take a full snapshot of the page DOM, including the DOM of all nested frames embedded within the page. We do so in a "blocking" fashion, so that the user input event is not propagated to any other software module, such as V8, that may alter the DOM itself before it's recorded (we intercept these events by injecting thin instrumentation shims within `WebCore::EventHandler`). While taking a snapshot of the DOM imposes some overhead, in Section 3.7 we show that in average the overhead is acceptable and does not significantly affect the user's experience.

3.5.3 Non-Deterministic Platform Calls

During the rendering of a web page, Blink and V8 may issue a number of different system calls to the underlying platform. For example, the rendering and JavaScript engines may initiate calls to `currentTime()` to synchronize rendering events (e.g., animations or other dynamic content). Additionally, the engines may issue system calls to obtain random values from the runtime environment. The values returned to Blink/V8 by such platform calls are non-deterministic, and we therefore need to record them so that they can be later replayed. In the following, we describe how we place instrumentation shims around the Blink and V8 platform APIs to achieve our goals while minimizing performance overhead.

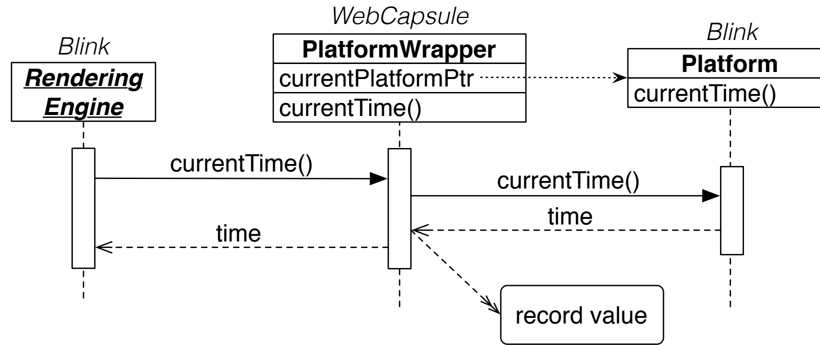

Figure 3.4: Simplified view of WebCapsule's platform wrapper in *record mode*. `PlatformImpl` is the actual implementation of the current underlying system platform.

Instrumenting Blink's Platform API

Blink provides a `Platform` interface that abstracts the provided platform API from its actual underlying implementation. To establish what specific `Platform` it is currently running on, Blink first calls a `Platform::current()` function, which returns a pointer to a static singleton instance of `Platform`. We leverage this "platform discovery" mechanism to our advantage. Specifically, WebCapsule includes a `PlatformWrapper` class, which implements the `Platform` interface and internally stores a reference to the true underlying platform returned by `Platform::current()`, as shown in Figure 3.4.

When recording is initiated, WebCapsule's `InspectorAgent` initializes the platform wrapper in the following manner. First, the `InspectorAgent` retrieves the pointer to the current platform instance. Next, the agent instantiates a new `PlatformWrapper`. Lastly, it replaces the value of the `Platform::current()` pointer with the address of the newly created `PlatformWrapper`. From this point on, every time Blink performs a call to any platform API, it will actually use WebCapsule's `PlatformWrapper` as its platform (see Figure 3.4). This design allows WebCapsule to seamlessly instrument Blink's entire Platform API while confining all instrumentation code

exclusively within the `PlatformWrapper` class. Furthermore, the platform API instrumentation is completely *transparent* from the point of view of its callers.

Our `PlatformWrapper` implements the `Platform` interface as follows. For deterministic functions, the parameters passed to the wrapper's function call are simply forwarded to the same function of the wrapped (true) platform instance. The return value, if there is one, is then directly passed back to the caller. However, for non-deterministic platform functions, their implementation within `PlatformWrapper` is slightly different. When WebCapsule is in *record mode*, we make copies of both the parameters passed to and the return value generated from the call to the wrapped platform instance. The recorded values are then stored in a data structure that allows them to be retrieved and later replayed (we explain in more detail how the `PlatformWrapper` operates during replay in Section 3.6).

Instrumenting V8's Platform API

Blink depends upon V8 for running JavaScript code. Effectively, Blink allows V8 to control the DOM of each page, thus providing the ability for JavaScript code to manipulate pages rendered by Blink. V8's access to the DOM tree is enabled by a set of dynamic *bindings* generated at compile time.

Platform Calls in V8 vs. Blink. To allow for accurate recording and replay, WebCapsule must capture non-determinism introduced by JavaScript that could affect page rendering within Blink. It turns out that because of how V8 is coupled to Blink, some of the Blink instrumentations described earlier allow us to also record certain JavaScript-driven web events. For instance, JavaScript `XMLHttpRequest` network transactions are actually satisfied by Blink and utilize the same network functionality that WebCapsule already instruments. Therefore, we can record `XMLHttpRequest` transactions as any other network request (see Section 3.5.4). Furthermore, many of Blink's platform API functions are passed to V8 as function pointers during initialization. Therefore, some of V8's platform API calls are actually calls to Blink's platform API, which we already record

via WebCapsule's `PlatformWrapper`, as discussed earlier. However, there are also a number of sources of non-determinism that reside solely within V8, and that could indirectly affect Blink's rendering. These sources include V8's own platform API and certain JavaScript functions, such as `Math.random()`.

Wrapping V8's Platform API. V8's platform API is implemented quite differently from Blink's, because it does not utilize a "clean" object-oriented design, and there is no single instance of a `Platform` object within V8 that we can easily "wrap". With the above complications in mind, we took the following approach. We create an (independent) platform within V8, which we refer to as `JSPlatformWrapper` in the remainder of this paper. We then modify the call sites within V8's code related to non-deterministic platform API calls to use our `JSPlatformWrapper` instead. For example, when JavaScript `Date()` objects are instantiated to retrieve the current system time, V8 calls `OS::TimeCurrentMillis()` (via a call to `RuntimeDateCurrentTime`). We slightly modify V8 to call WebCapsule's platform wrapper first, so that we can record the current time value, and transparently pass it back to V8. This design does require that several call sites for non-deterministic platform API calls within V8 be identified and instrumented. However, it has the advantage that we can choose not to instrument a call site if the resulting non-determinism is known not to affect page rendering or Javascript execution.

Leveraging JS-to-C++ Calls. Let us now consider JavaScript's `Math.random()`. The random number generator exposed by `Math.random()` is one of the primary sources of non-determinism internal to V8. Unlike, `Date()`, V8 internally implements `random()` entirely in JavaScript. However, V8 defines several C++ preprocessor macros, which are used to define C++ functions callable from JavaScript code. We implement a new function called `HandleMathRandomVals`, which takes the return value of `random()` as a parameter. We then altered `random()` to call `HandleMathRandomVals` before returning, which in turn passes the values to be recorded to our V8 platform wrapper, (see function calls starting with '`%`' in Figure 3.5).

```
function MathRandom() {
  /* Begin WebCapsule's PlatformInstrumentation Replay Code */
  var retval = %NextMathRandomVals();
  if(retval >= 0) return retval;
  /* End of WebCapsule's PlatformInstrumentation Replay Code */

  /* Original MathRandom() code */
  var r0 = (MathImul(18273, rngstate[0] & 0xFFFF) + (rngstate[0] >>> 16)) | 0;
  rngstate[0] = r0;
  var r1 = (MathImul(36969, rngstate[1] & 0xFFFF) + (rngstate[1] >>> 16)) | 0;
  rngstate[1] = r1;
  var x = ((r0 << 16) + (r1 & 0xFFFF)) | 0;
  retval = (x < 0 ? (x + 0x100000000) : x) * 2.3283064365386962890625e-10;

  /* Begin WebCapsule's PlatformInstrumentation Recording Code */
  %HandleMathRandomVals(retval);
  /* End of WebCapsule's PlatformInstrumentation Recording Code */
  return retval;
}
```

Figure 3.5: WebCapsule's instrumentation of V8's `Math.random()` implementation (from `v8/src/math.js`). Function calls starting with '%' are used to call C++ functions internal to V8 from JavaScript code.

3.5.4 Recording Network Events

Asynchronous Requests. Network requests are primarily served in an asynchronous way, and responses are returned via a callback interface. With this design, the response is constructed piecemeal over the course of several callbacks. The asynchronous network request interface within Blink is defined by two classes, `WebURLLoader` and `WebURLLoaderClient`. The `WebURLLoader` abstracts the underlying network and caching functionality provided by the platform API. The `WebURLLoaderClient` comprises the callback interface used to collect the response.

As shown in Figure 3.6, WebCapsule records network events by leveraging the `PlatformWrapper` described earlier. When Blink, via a `ResourceLoader` instance, requests the platform to create a new URL loader, WebCapsule's `PlatformWrapper` returns a pointer to a `ForensicURLLoader`, which itself is a wrapper to the `WebURLLoader` actually provided by the underlying platform API. In addition, our `ForensicURLLoader` implements the `WebURLLoaderClient` in-

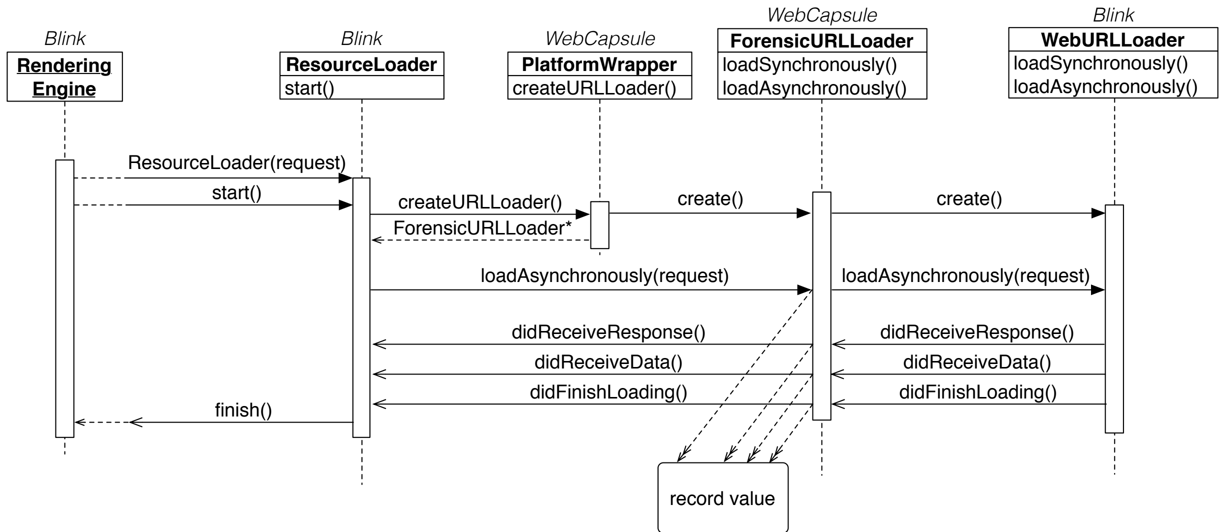

Figure 3.6: Simplified view of how WebCapsule records asynchronous network transactions.

terface, and passes itself as the client to the true `WebURLLoader`. Therefore, as network data arrives, the `ForensicURLLoader` is called first, records the desired information, and then calls back into the `ResourceLoader`, so that Blink can parse and render the response.

Synchronous Requests. In practice, synchronous network requests are activated only in a small number of cases (e.g., requests are made synchronously when `false` is passed as the third parameter to the `open` function of a Javascript `XMLHttpRequest` object). These calls can be recorded fairly easily. Because the results of the request are completely realized by the time `loadSynchronously()` returns, we can record the results with a single wrapped function.

Browser Cache Considerations. WebCapsule's design for recording network transactions has the added benefit of abstracting the actual data source which satisfies resource requests. Obviously, the browser could satisfy Blink's network requests from a physical network, but it could also satisfy a request from the browser-level cache or from the associated resources of a browser extension. However, from the point of view of Blink these different data sources are invisible, in that where the network response is coming from does not really matter. Specifically, by recording the results

of each resource request as explained above, we can replay not only network transactions, but also transparently recreate any browser cache hits without having to explicitly record the cache state.

3.6 Replay: Design and Implementation

WebCapsule's recording capabilities aim to collect enough detailed information to allow a forensic analyst to reconstruct web security incidents such as social engineering and phishing attacks. In addition, we aim to enable the analyst to also perform a detailed replay of previously recorded browsing traces, as explained below.

Entering Replay Mode. To replay a previously recorded trace, we leverage DevTools's JSON protocol. Specifically, we define new DevTools commands that allow for remotely controlling WebCapsule's operating mode. Concretely, to enter replay mode we can send WebCapsule two commands: `LoadRecording <trace-file>`, and `StartReplay`. The first command loads a previously recorded browsing trace from disk, and the second one starts replay by instructing Blink to load the first page URL in the trace and then replay the browsing events, as described in detail below.

Notice that the replay can occur in a separate and completely isolated environment with no network connection or input devices, because during replay mode Blink will be forced to satisfy network requests and receive user input exclusively from the recorded trace.

Replay Strategy Overview. WebCapsule employs two replay strategies, depending on the source of the recorded data. As shown in Figure 3.7, inputs which originated from Blink's web-rendering API are replayed by explicitly re-calling the event handler function from which the input was recorded, effectively forcing the input (e.g., a mouse movement or keypress) to be re-injected into (and processed by) Blink. On the other hand, non-deterministic inputs obtained through the platform API are primarily replayed by simply waiting for Blink and V8 to call the platform as a consequence of the replay of the web-rendering API inputs and the rendering process. For each

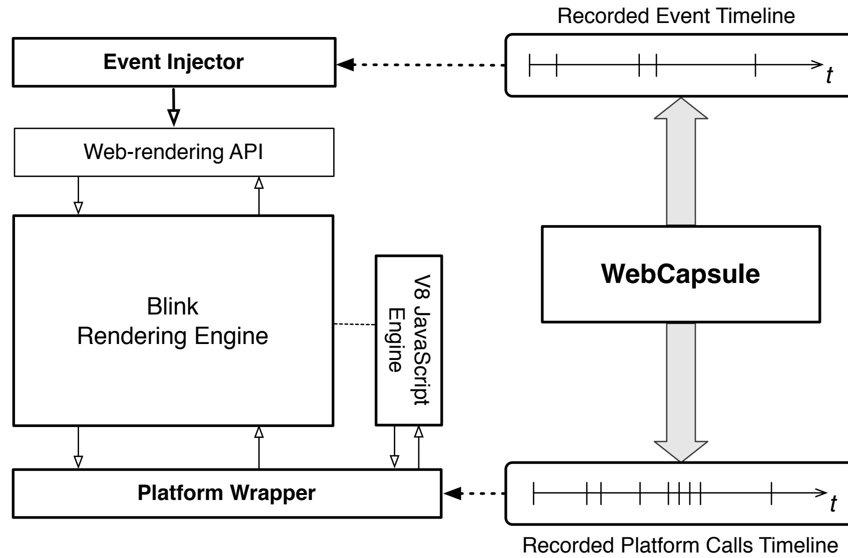

Figure 3.7: Simplified view of WebCapsule's replay strategy.

platform API call issued during replay, we identify the corresponding call observed during recording and directly return the previously recorded return value without having to call the true underlying system platform. Notice also that all inputs to be re-injected are timestamped, and are replayed following a precise event timeline.

For instance, as a mouse click on an HTML anchor element is re-injected into Blink via the web-rendering API, the rendering engine will start issuing the necessary network requests (via the platform API) to navigate to and render the new page. As the network requests are received by WebCapsule's `PlatformWrapper` (see Section 3.5.3), we return the previously recorded network response to Blink. A more detailed explanation of WebCapsule's replay mechanisms is provided below.

3.6.1 Replaying Web-Rendering API Events

Recorded input events (e.g., touch gestures, mouse clicks, keypresses, etc.) carry a timestamp. This allows us to re-inject all inputs into Blink in the correct chronological order, and to preserve the relative time gap between events. Concretely, user input events are replayed by calling the related handler function in Blink (e.g., `WebViewImpl::handleInputEvent`), with the event as an argument, thus asking Blink to process the event and to dispatch it to its internal modules (including possible JavaScript listeners).

3.6.2 Replaying Platform Calls

As user inputs are re-injected into Blink and web content is rendered, the rendering and JavaScript engines will issue calls to the underlying platform, such as network requests, calls to obtain the current system time, etc. As these calls are made, WebCapsule's `PlatformWrapper` and `JSPlatformWrapper` (defined in Section 3.5.3) can return the value that the same call had returned during recording. To identify the correct return value within all recorded calls of a given function, we use the combination of parameters passed to the function during recording as a key. To break "ties" on possible key collisions, the value returned during replay is simply the next unconsumed recorded return value from the related function call (with the same key) chosen in chronological order.

Challenges. As mentioned in Section 3.4, Blink is highly multi-threaded, making the replay of some platform API calls challenging, especially for functions that take no input parameters (and therefore have no obvious key), such as Blink's `Platform::currentTime` or V8's `OS::TimeCurrentMillis`. During replay, depending on the (non-deterministic) scheduling of the threads, Blink and V8 may make some API calls at different "speed", compared to what happened during recording. One way to address this problem would be to record the non-determinism introduced by the thread scheduler. Unfortunately, precisely recording (and replaying) thread schedul-

ing information from within Blink is extremely challenging. Alternatively, one may attempt to manipulate thread scheduling from "outside" of Blink. However, this is not an option for us, because it would violate our main goal of not altering any code outside of Blink (to completely inherit its portability) and of introducing only low overhead so that WebCapsule can be used as an "always on" forensic data collection system.

Proposed Solution. To address these challenges, we use a best effort approach that we found to work well in practice. During replay, whenever `currentTime()` is called (either by Blink or V8), we return the previously recorded time value that is *closest* to the current replay time delta. More specifically, let S_{rec} and S_{rep} be the time when recording and replay started, respectively, and v_{rec} be the list of `currentTime()` return values stored during recording. Suppose that during replay Blink calls `currentTime()` at time t_{rep}. To choose the return value, we first compute the replay time delta $\delta_{rep} = t_{rep} - S_{rep}$. We then find the return value, $t_{rec} \in v_{rec}$, such that $\delta_{rec} = t_{rec} - S_{rec}$ is the closest to δ_{rep} (i.e., we minimize $|\delta_{rec} - \delta_{rep}|$). Because the web-rendering API events (e.g., mouse movements and keypresses) are re-injected respecting the relative time deltas observed during recording (see Section 3.6.1), the approach described above has the effect of loosely "re-synchronizing" the `currentTime()` replay clock to the user input events, thus improving replay accuracy.

3.6.3 Replaying Network Events

To replay network responses, WebCapsule leverages the `ForensicURLLoader` class discussed in Section 3.5.4. During replay, when a network resource is requested WebCapsule's URL loader finds the recorded response for that request (which may include redirection chains or error messages) using the request's URL and some other request parameters (e.g., HTTP request headers) as a key. Once the response is located, a series of WebCapsule's re-execution events are created, representing each of the `WebURLLoaderClient` callbacks to be executed (see Section 3.5.4). Using this technique, we are able to return the desired network response to Blink.

Challenges. In some cases, the URL of network requests driven by JavaScript (e.g., `XMLHttp-Request`) may be created dynamically. As a concrete example, consider a search box element on Amazon.com's front page[3]. Every time the user enters a character, a piece of JavaScript code issues an `XMLHttpRequest` to retrieve a set of search term suggestions. While the structure of the URL is always the same, some of the URL query parameters change. For example, the URL contains the partial search term entered by the user and a timestamp (retrieved via a call to V8's platform API). During replay, as the keypresses are reinfected into the search box, the related `XMLHttpRequest` are re-issued. However, the timestamp appended to the URL may cause a key mismatch, which would not allow us to easily find the correct response to be re-injected into Blink. As explained in Section 3.6.2, WebCapsule is able to "re-synchronize" the `currentTime()` replay clock to the user input events, which alleviates this problem. However, in some cases our `PlatformWrapper` may return a time value that is a few milliseconds off, compared to what was observed during recording for the same `XMLHttpRequest`, thus still causing a mismatch.

Proposed Solution. To address the above problem, we use a best effort approach that works very well in practice. During recording, every time a network request is issued, we determine if it was initiated (directly or indirectly) by JavaScript. If that's the case, we reconstruct the JavaScript call stack to identify exactly what JavaScript function caused the network request to be issued, and store this information in the browsing trace. Then, for those replay events in which there is a URL mismatch and we cannot easily identify the related network response data, we analyze the JavaScript call stack of all the *not-yet-consumed* responses in the recorded browsing trace, and return the "closest" response. Specifically, let $R_{rec} = (q_i, r_i)_{i=1...n}$ be such a set of unconsumed network requests, q_i, and related responses, r_i. Also, let $q_i^{(rep)}$, be the network request issued during replay that we are trying to match with a response. We then find the request $q_{i^*} \in R_{rec}$ whose JavaScript call stack matches the call stack associate to $q_i^{(rep)}$, and whose timestamp is the closest to $q_i^{(rep)}$'s timestamp (computed as a delta from the replay start time and record start time,

[3]Latest page analysis performed on February 22, 2015.

respectively). Then, we return the related response r_{i^*}. In the rare cases in which the network response search still fails, we return an empty response with HTTP code `204 No Content`.

3.6.4 Divergence Detection and Self-Healing

The replay approaches described in Section 3.6.2 and 3.6.3 work well in practice. Nonetheless, there may be (rare) cases in which we still fail to correctly replay an event (e.g., due to complex thread scheduling issues), causing the replay to differ slightly from the recorded browsing trace. As a concrete example, assume that during recording the user clicked on an element on a given web page, P_1, and the browser navigates to page P_2. Suppose that during replay a problem occurs, and the same re-injected click does not cause the expected transition from P_1 to P_2. To recover from these problems, WebCapsule implements a replay *self-healing* approach. As mentioned in Section 3.5, during recording each recorded user input event includes the URL of the page where the input occurred. Therefore, if during replay the browser does not navigate to page P_2, this will cause a mismatch between the URL associated to the next input event to be replayed (i.e., P_2's URL), and the URL of the current page rendered on the browser (which erroneously remained on P_1). WebCapsule is able to detect such a mismatch, and responds to these cases by forcing the browser to load P_2, before continuing with normal replay of the remaining events on the trace.

If self-healing occurs, WebCapsule outputs detailed information about the self-healing process to the replay logs, to notify the forensic analyst that a replay problem has been encountered and to explain how WebCapsule recovered from it. In general, WebCapsule implements a number of mechanisms to *detect and explain* any differences between the recorded traces and the replay events, so that the forensic analyst can accurately reconstruct what happened while the user was browsing.

3.7 Evaluation

3.7.1 Experimental Setup

We performed experiments on two different devices: a desktop Dell Optiplex 980 with a Core i7 870 CPU and 8GB of RAM running Ubuntu Linux; and a Asus Nexus 7 tablet with 2GB of RAM, 32GB of storage, and running Android 5.0.1. We also used an x86-based Android Virtual Device (AVD), to demonstrate that WebCapsule can be used to record traces on a physical device (the ARM-based Nexus 7) and to later replay the traces in a different platform.

For all experiments, we used the Chromium[4] codebase. We deployed Chromium with our WebCapsule instrumentations on the desktop computer, and a ChromeShell APK [16] with WebCapsule enabled on the Nexus 7 (we were also able to perform some preliminary experiments with WebView + WebCapsule on Android, which are not reported here; we plan to expand and report the WebView experiments in our future work).

All experiments were performed using the default browser process model [44], whereby each browser tab is handled in a different process (except in some corner cases). In addition, each process uses its own separate instance of Blink and V8. In this process model, WebCapsule could record multiple tabs independently. Therefore, all our results refer to experiments performed on one single tab.

Overall, our main code modifications to Blink and V8 (including the DevTools modifications and platform API wrappers discussed in Sections 3.5 and 3.6) consist of approximately 14,000 lines of code (primarily C++ code, plus a number of Python scripts for log analysis). We plan to release our WebCapsule prototype system and a variety of browsing traces collected for evaluation at `http://webcapsule.org`.

[4]Git commit: `45eed524365a1cbc612aba31ab36aafd7788d825`.

3.7.2 Functionality Tests

First, we performed a set of *functionality tests*, which aim to verify that WebCapsule's record and replay capabilities do not negatively impact Blink's functionalities (e.g., support for JavaScript and DOM manipulation functionalities). To this end, we leverage two popular web browser benchmarks that aim to test functional correctness. These benchmarks include Web Standards Project's Acid3 [5, 61] and the Dromaeo Test Suite developed by Mozilla [19].

Using both Chromium and the ChromeShell APK with WebCapsule on (first in record mode, and then in replay mode), we separately ran the Acid3 and Dromaeo tests (for Dromaeo, we ran the "DOM Core Tests" and the "V8 JavaScript Tests"). WebCapsule was able to record and replay the Dromaeo tests with no errors. Similarly, the Acid3 tests completed correctly during recording, though one test raised an exception during replay. Specifically, Acid3 ran 100 different JavaScript and DOM manipulation tests, and during replay WebCapsule missed to pass only one test on both Linux and Android, namely `Test 80`[5], due to a network request that we did not match correctly.

The above results show that in record mode WebCapsule is completely *transparent*, because it does not alter the core functionalities of Blink/V8. In addition, these tests show that WebCapsule can replay complex web page events with high accuracy (perfectly for Dromaeo, and 99% accuracy for Acid3), as shown in Table 3.1.

After investigating the browsing events related to `Test 80`, we found that the URL requested for that test had a timestamp embedded in the query string; during replay, the timestamp embedded in the requested URL was always a few milliseconds off, compared to the recording phase[6], thus causing a mismatch. We later[7] solved this URL mismatch problem with a minor adjustment to the implementation of the algorithm discussed in Section 3.6.3 for "re-synchronizing" `currentTime()`'s return value. This allowed us to correctly replay Acid3 tests with 100% accuracy, and further improve the overall replay accuracy of WebCapsule.

[5]Error message: "Test 80 failed: timeout – could be a networking issue"

[6]e.g., recorded URL: `empty.html?1431430350104`; replay URL: `empty.html?1431430350157`

[7]After initial submission

Table 3.1: Functionality Tests

	Acid3 Errors		**Dromaeo Errors**	
	Record	**Replay**	**Record**	**Replay**
Linux	0/100	1/100	*no errors*	*no errors*
Android	0/100	1/100	*no errors*	*no errors*

3.7.3 Evaluation on Phishing Attacks

WebCapsule's main goal is to enable an always-on and transparent collection of browsing data that can help a forensic analyst to precisely reconstruct web-based attacks, especially for attacks that directly target users, such as phishing attacks.

To test if WebCapsule can successfully record and subsequently replay real-world phishing attacks, we proceeded as follows, using Chromium on our desktop machine. We selected a large and diverse set of recently reported phishing web pages from PhishTank[8]. The pages we tested represented "fresh" (recently reported) attack URLs. Overall, the attack traces were recorded by six different users who visited and interacted with a total of 112 different active phishing URLs. Each user visited around 15 to 20 URLs and simulated the leakage of (fake) information. Essentially, all phishing pages aim to trick the user into providing some type of personal user information, such as the user name and password required to access popular services (e.g., Google Drive, Yahoo Mail, etc.). In addition, we tested several phishing attacks mimicking online banking sites (e.g., Bank of America, Barclays, Paypal, etc.). These attacks are particularly aggressive, in that they attempt to trick the user into providing a large number of highly sensitive data, including social security numbers, date of birth, driver's license numbers, mother maiden name, answer to multiple security questions, etc.

[8] http://www.phishtank.com/phish_archive.php

To determine how well WebCapsule can record and replay phishing attacks, we measured the following quantities. For each attack trace, we wanted to quantify how well each trace could be replayed. To this end, every time a mouse click or keypress event occurred during replay, we compared the target DOM element of the replay event to the target DOM element of the same event observed during recording. For example, assume that during replay a mouse click m was injected on an anchor element, say e' =`<a href="go.html">go</a>`. As discussed in Section 3.5, during recording not only do we store the internal details of m, but also its original target element e. Therefore, during replay we can perform a comparison between e' and e, and increment the number of target errors if the elements differ. Similarly, we recorded the URL of each page (specifically, the main frame URL) through which the user navigated while under phishing attack. Then, we compared the URLs observed during replay with the ones observed during recording, and counted differences in the page URL sequences.

The results are summarized in Table 3.2. As can be seen, the vast majority of traces (almost 90%) replayed perfectly. Specifically, WebCapsule was able to replay 106 out of 112 phishing traces with no page transition errors. The 6 page errors were primarily caused by corner case scenarios that are not currently handled by our proof-of-concept code and that could be fixed with additional engineering effort. For example, in some cases our network replay approach (see Section 3.6.3) failed to match a dynamically generated URL, and at the same time the JavaScript call stack matching algorithm described in Section 3.6.3 was not able to correctly recover the appropriate network response. We plan to add support for these corner cases in our next releases of WebCapsule.

Also, 100 traces had no target element errors for mouse click events, and 103 traces had no keypress target element errors. For the remaining traces with click and keypress target element errors, most of them were related to the 6 page transition errors. For example, if a page transition error occurs, a click event may be replayed on the wrong page, and therefore also on the wrong target element. In other cases, an error may occur even if the event is injected in the correct page. One

Table 3.2: Replay correctness for phishing attack pages. Measurements performed over 112 phishing traces collected by 6 users.

	Avg. # Events per Trace	**100% Correct Traces**	**Traces with Some Errors**
Page Transitions	4	106/112	6/112
Clicks	14	100/112	12/112
Keypresses	270	103/112	9/112

of the main causes for this is as follows. Some attack pages made heavy use of JavaScript, to the point that the entire page content was generated in a completely dynamic way. While our prototype implementation of WebCapsule can replay the vast majority of these cases, we encountered some scenarios in which the replay of JavaScript code used for building the page was not completely accurate. Therefore, the page content did not render the same exact way as during recording, and the click and keypresses missed the related targets on those pages. In our future work, we plan to also add support for the above cases as well.

3.7.4 Record & Replay of Popular Websites

To further evaluate WebCapsule's record and replay functionalities and measure the overhead introduced by our instrumentation of Blink, we performed tests on several representative popular websites, on both Linux and Android devices (see Section 3.7.1 for details on device configurations).

Performance Analysis. For each website shown in Table 3.3, we performed a few minutes of "fast pace" browsing, during which we issued numerous input events, such as mouse clicks, touchscreen gestures, and keypresses, and navigated through several pages. During this test, we recorded the browsing activities and measured the overhead introduced by our WebCapsule instrumentation code over Blink. To accomplish this goal, we leveraged the profiling framework already imple-

Table 3.3: Performance test results. Overhead computed during recording of browsing activities on popular websites.

Platform	Website	WebAPI overhead %	Platform overhead %	Network overhead %
Linux (Optiplex)	Google	16.08	0.12	3.76
	Facebook	5.30	1.58	0.54
	Youtube	5.04	0.67	2.57
	Amazon	16.78	0.72	0.32
	Yahoo	8.99	0.20	0.89
	Wikipedia	15.51	0.22	0.33
	Ebay	7.63	0.34	0.31
	Reddit	13.16	0.14	1.39
Android (Nexus 7)	Google	6.89	0.78	1.49
	Craigslist	7.68	0.58	0.19
	Youtube	7.77	0.66	1.39
	Flickr	8.23	0.75	0.75
	IMDB	7.48	0.76	0.15
	Yelp	6.33	0.59	1.59
	Ebay	7.23	0.82	0.77
	Reddit	4.40	0.57	1.85

mented in Blink. Specifically, we added calls to `TRACE_EVENT` macros [55] within each single Blink function we instrumented, including around all web-rendering API (or Web API, for short), platform, and network-related "hooks" that we use to record non-deterministic inputs. This allowed us to precisely compute the CPU overhead introduced by our recording infrastructure. The results are reported in Table 3.3. We break down the overhead introduced by the code used to record Web API, platform, and network events, respectively.

Our results show that WebCapsule introduces reasonable overhead both on Linux and Android, making its use as an always-on system practical. For the Web API events overhead, which is always lower than 17% on Linux and 9% on Android, we need to consider that this corresponds to only a few milliseconds of overhead added to the processing of a user input event. During the recording of our browsing traces this delay was visually unnoticeable to the user. The platform

overhead, which measures the added time spent to record the input and return values of calls to Blink's platform API calls, is always low on both platforms, never exceeding 2%. Finally, the time spent by WebCapsule to record network requests and responses has only a small impact on network latency, with an overhead always below 4%. The lower WebAPI performance overhead for the Android traces is likely due to the lower complexity of the mobile version of the websites (e.g., taking a snapshot of the DOM at every click is less expensive).

We also computed the amount of data that would need to be stored to archive WebCapsule's browsing traces. On average, using Chromium our browsing on popular websites produced 37.3kB/s of offloaded browsing events data, with network-related data being responsible for the vast majority (almost entirety) of the offloaded information. As we mentioned in Section 3.3 (see non-goals and future work), in this paper we do not focus on how to minimize storage use. However, it is worth noting that many enterprise networks already use commercial solutions to store full network packet traces for considerable periods of time (e.g., for compliance or security reasons). Therefore, it would be possible to adapt such solutions to store WebCapsule's traces.

Replaying Browsing Traces. Besides measuring the performance overhead introduced by WebCapsule in recording mode, we also tested how well the recorded traces could be replayed. As shown in Table 3.4 the vast majority of traces replayed correctly, with no visually noticeable difference in the rendering of the pages between recording and replay. On Linux, only Youtube caused a replay problem. Specifically, while replaying the Youtube browsing trace we encountered an assertion failure[9] on a part of Blink's code dedicated to "painting" the rendered page, causing the page, and our instrumentation agent, to freeze. We plan to further investigate and correct this issue in future versions of WebCapsule. We were also able to successfully replay searching/browsing on Google.com. However, replay was fully successful (on both Linux and Android) only with Google Instant predictions turned off [27]. Google Instant's JavaScript code seems to use a non-deterministic input that is not fully supported by our prototype. This causes one of the parameters

[9] `!m_needsToUpdateAncestorDependentProperties` – in `RenderLayer.h`

of the URLs for network requests issued by Google Instant during replay to be slightly different from recording. While our JavaScript call stack matching algorithm (Section 3.6.3) helps us identify the correct (previously recorded) network response to re-inject into Blink, it appears that Instant's code performs some sort of "response content verification," which prevents the Instant search results to be correctly rendered. Because the relevant JavaScript code is heavily minimized, reverse-engineering Google Instant is fairly complex. Therefore, we plan to add support for Google Instant in future releases of WebCapsule.

For our Android experiments performed on the Nexus 7, we performed multiple tests on each of the sites listed in Table 3.4. For each site, we were able to record and fully replay the browsing activities. Only two sites caused some issues, and only for specific browsing scenarios. Specifically, on Youtube and Yelp our recording engine did not support the site's search functionality. Namely, after typing a search term and hitting Enter or the search button, the search results would load but in some cases the browser page would freeze. Performing other browsing activities on Youtube (with no search) did not cause any noticeable issues, and both record and replay worked with no problems. Recording and replaying Yelp also worked better when the site's search function was not used, though we experienced some other less critical issues (e.g., a missed page transition) during replay, due to a fixable engineering issue in our prototype code.

To better understand the origin of the replay issues related to the search functionality on Yelp and Youtube, we performed an in-depth investigation of WebCapsule's execution traces. We found that the above mentioned problems were caused by a combination of a small bug within our WebCapsule code (now fixed) plus two known bugs within the version of Chromium we developed against. Our own bug was related to the implementation of the JavaScript call-stack reconstruction code (see Section 3.6.3), which would sometimes return an unhandled `null` value, causing WebCapsule to crash. Fixing this bug solved the problem with the search functionality on Yelp. However, Youtube's search functionality was still not working properly. After further debugging, we were able to confirm that this was entirely due to two separate bugs in Chromium itself (Issue-

Table 3.4: Record and replay tests on popular websites. Test are marked as follows: ✔ = successful test; ★ successful test with some divergence; ✖ test with problems; ✳ test with problems that we later fixed. Multiple symbols indicate different results depending on the type of browsing activity on the site.

Platform	Site	Record	Replay	Comment
Linux	Google	✔	✔★	Google Instant predictions off
	Facebook	✔	✔	No noticeable difference
	Youtube	✔	✖	Replay page rendering problem
	Amazon	✔	✔	No noticeable difference
	Yahoo	✔	✔	No noticeable difference
	Wikipedia	✔	✔	No noticeable difference
	Ebay	✔	✔	No noticeable difference
	Reddit	✔	✔	No noticeable difference
Android	Google	✔	✔★	Google Instant predictions off
	Craigslist	✔	✔	No noticeable difference
	Youtube	✔✳	✔✳	Search function issues
	Flickr	✔	✔	No noticeable difference
	IMDB	✔	✔	No noticeable difference
	Yelp	✔✳	★✳	Search function issues
	Ebay	✔	✔	No noticeable difference
	Reddit	✔	✔	No noticeable difference

365858 and Issue-460328) that caused ChromeShell to crash whenever a key-stroke was entered into Youtube's search field. After implementing a workaround for both Chromium's bugs, WebCapsule was able to correctly replay Youtube's search functionalities as well.

All other Android record and replay tests worked notably well. In addition, we were able to successfully replay the traces we recorded on the ARM-based Nexus 7 into a separate x86-based Android virtual device, thus showing that WebCapsule's traces can be replayed in a different platform and in a separate isolated environment.

Demos. To further demonstrate the record and replay abilities of WebCapsule, we have recorded five example "demo videos" that show a representative demonstration of how WebCapsule can be used to record and replay browsing activities. We produced two videos related to phishing attack

traces, and three for highly popular websites (Flickr, Amazon, and Wikipedia). These videos have been posted before the submission deadline, as demonstrated by the post dates on Youtube. We also took care to remove any identifiable information from the videos themselves, so not to break anonymity during the paper submission and review process.

- *Flickr (mobile)*: `http://youtu.be/K1CwIwcTgbE`
- *Amazon*: `http://youtu.be/inhkt88RqN8`
- *Wikipedia*: `http://youtu.be/AelqP91QfLg`
- *Phishing 1*: `http://youtu.be/hOcH3OQj9HU`
- *Phishing 2*: `http://youtu.be/mMiZ17Qlh0M`

3.8 Related Work

Forensic analysis of web-based incidents generally relies on analyzing the web browser's history, cache files, and system logs [39, 31], or on web traffic traces [1, 28, 64, 37]. However, such approaches do not provide the ability to fully reconstruct and replay web security incidents, especially for incidents that directly involve the user (e.g., phishing and social engineering attacks), as also discussed in Section 3.2.

Record and replay is a commonly desired feature for debugging and troubleshooting complex software and systems, and a number of previous efforts have been explored to support record and replay at different levels. For example, approaches based on virtual machines [58, 21, 14, 40] have been proposed to record system level events (interrupts, thread scheduling, etc.) and replay application and system execution at the level of single instructions. While they are designed for generic application record and replay, VM-based approaches cannot be easily deployed with low performance overhead to resource constrained devices, such as smartphones and other mobile devices. In contrast, our WebCapsule system focuses on *always on* record and replay of web browsing traces, and provides a lightweight and practical solution that can be deployed with no changes in a vari-

ety of different web-rendering applications and platforms, including mobile devices (e.g., Android devices).

WebCapsule also differs significantly from prior work that focuses exclusively on replaying specific web components, such as JavaScript code [35], user gestures and other sensor inputs on mobile devices [26], or user interactions with web applications [7]. Unlike these previous studies, WebCapsule aims to record and replay the execution of a web browsing trace in its entirety, including network transactions, and non-deterministic calls to the underlying system platform. This allows us to replay in a completely isolated environment, without requiring new user or network inputs. This is especially important when there is a need to replay potential web-security incidents for which the server-side content is ephemeral and many not be otherwise available at the time of replay.

Another work related to ours is TimeLapse [11], a developer tool that focuses on the record and replay of human visible web events, and on the interoperability with existing web application debugging tools. As discussed in more details in Section 3.2, TimeLapse does not work as an always on recording system and is not easily portable to different browser and operating systems (it is currently limited to Safari+MacOS). In addition, Timelapse is not transparent, because it deeply modifies WebKit (e.g., to force a synchronous scheduling of threads). In contrast, WebCapsule can perform low-overhead always on recording, and is also transparent and portable.

3.9 Conclusion

In this paper, we presented *WebCapsule*, a novel *record and replay forensic engine* for web browsers. WebCapsule's main goal is to work as an *always-on* and *lightweight* (i.e., low overhead) forensic data collection system that enables a full reconstruction of web security incidents, including phishing and social engineering attacks. We designed WebCapsule to be a *portable* system by instrumenting Google's Blink rendering engine without altering its application and platform APIs. Our

experimental results on both Linux and Android systems show that WebCapsule can accurately record and replay complex web applications, including popular websites and real-world phishing attacks, with reasonable performance overhead, thus making always-on recording practical.

Acknowledgments

We thank Minesh Javiya (Stony Brook University) and Jienan Liu (University of Georgia) for their help with collecting the browsing traces used to evaluate WebCapsule. We would also like to thank the Chromium development team for their documentation of Blink and of the entire Chromium project, and the anonymous reviewers for their constructive and very helpful comments.

This material is based in part upon work supported by the National Science Foundation, under grants No. CNS-1149051 and CNS-1514142. Any opinions,findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Chapter 4

Summary

Analyzing real-world web attacks that directly target users is an extremely challenging and time-consuming task. Unfortunately, as the prevalence of web-based cyber attacks has increased so has the difficulty of providing accurate and timely forensic analysis of said attacks. The state-of-the-art methods for reconstructing web-based security incidents often rely on an analyst manually collecting and correlating disparate pieces of digital evidence in order to obtain an often imprecise view of what really happened.

Record and replay systems offer an avenue to create tools which mitigate the challenges that hinder the analysis of web-based security incidents. Record and Replay systems are those which capture and deterministically reproduce the execution of some system. In this dissertation we present two such systems designed to aid in analyzing web security incidents, ClickMiner and WebCapsule.

ClickMiner aims to automatically reconstruct user-browser interactions from archived web traffic. Unlike previously proposed approaches based solely upon the analysis of HTTP headers, ClickMiner reconstructs user-browser interactions by actively replaying archived (potentially partial) HTTP traffic within an instrumented browser. Our evaluation demonstrates that ClickMiner can reconstruct between $\simeq$ 82% and $\simeq$ 90% of user-browser interactions with false positives be-

tween 0.74% and 1.16%. We also demonstrate via a case study how ClickMiner can be used to reconstruct the full chain of user-browser interactions made during a real social engineering-based malware download attack.

WebCapsule is a record and replay forensic engine for web browsers. WebCapsule is implemented as a self-contained instrumentation layer within Google's Blink web rendering engine which records all non-deterministic inputs into the engine. WebCapsule enables always-on, transparent, and fine-grained recording (and subsequent replay) of web browsing sessions. As such it allows an analyst to fully investigate incidents involving ephemeral web content, such as short-lived phishing or social engineering attack pages. We evaluate WebCapsule over numerous real-world phishing attack instances, and demonstrate that such attacks can be accurately recorded and fully replayed. We also evaluate WebCapsule on different physical devices and platforms, including Linux and Android, and show that we can record complex browsing sessions on popular highly-dynamic websites while imposing reasonable overhead, thus making always-on recording practical.

Bibliography

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

[2] Xvfb - virtual framebuffer X server for X version 11. `http://www.x.org/archive/X11R7.7/doc/man/man1/Xvfb.1.xhtml`.

[3] The official easylist website, 2013. `https://easylist.adblockplus.org/en/`.

[4] Selenium webdriver, 2013. `http://docs.seleniumhq.org/projects/webdriver/`.

[5] Acid3. `http://acid3.acidtests.org`.

[6] Altekar, G., and Stoica, I. Odr: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 193–206.

[7] Andrica, S., and Candea, G. Warr: A tool for high-fidelity web application record and replay. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks* (Washington, DC, USA, 2011), DSN '11, IEEE Computer Society, pp. 403–410.

[8] Bergan, T., Anderson, O., Devietti, J., Ceze, L., and Grossman, D. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 53–64.

[9] Blink web rendering engine. http://www.chromium.org/blink.

[10] Büchner, A. G., and Mulvenna, M. D. Discovering internet marketing intelligence through online analytical web usage mining. *SIGMOD Rec. 27*, 4 (1998), 54–61.

[11] Burg, B., Bailey, R., Ko, A. J., and Ernst, M. D. Interactive record/replay for web application debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2013), UIST '13, ACM, pp. 473–484.

[12] Chen, K. Z., Gu, G., Zhuge, J., Nazario, J., and Han, X. Webpatrol: automated collection and replay of web-based malware scenarios. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 186–195.

[13] Choi, J.-D., and Srinivasan, H. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools* (1998), ACM, pp. 48–59.

[14] Choi, J.-D., and Srinivasan, H. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools* (New York, NY, USA, 1998), SPDT '98, ACM, pp. 48–59.

[15] Chrome devtools. https://developer.chrome.com/devtools/docs/integrating.

[16] Chromeshell. https://code.google.com/p/chromium/wiki/AndroidBuildInstructions.

[17] Cortesi, A. mitmproxy: a man-in-the-middle proxy, 2013. http://mitmproxy.org/.

[18] Denning, D. E., and Branstad, D. K. A taxonomy for key escrow encryption systems. *Commun. ACM 39*, 3 (Mar. 1996), 34–40.

[19] Dromaeo javascript test suite. `http://dromaeo.com`.

[20] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 211–224.

[21] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation* (New York, NY, USA, 2002), OSDI '02, ACM, pp. 211–224.

[22] Dunlap, G. W., Lucchetti, D. G., Fetterman, M. A., and Chen, P. M. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 121–130.

[23] Etminani, K., Delui, A., Yanehsari, N., and Rouhani, M. Web usage mining: Discovery of the users' navigational patterns using som. In *Networked Digital Technologies, 2009. NDT '09. First International Conference on* (2009), pp. 224–249.

[24] Georges, A., Christiaens, M., Ronsse, M., and De Bosschere, K. Jarec: a portable record/replay environment for multi-threaded java applications. *Software: Practice and Experience 34*, 6 (2004), 523–547.

[25] Gomez, L., Neamtiu, I., Azim, T., and Millstein, T. Reran: Timing-and touch-sensitive record and replay for android. In *Software Engineering (ICSE), 2013 35th International Conference on* (2013), IEEE, pp. 72–81.

[26] Gomez, L., Neamtiu, I., T.Azim, and T.Millstein. Reran: Timeing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 ICSE* (2013).

[27] Google instant predictions. `https://support.google.com/websearch/answer/186645?hl=en`.

[28] Hong, S.-S., and Wu, S. On interactive internet traffic replay. In *Recent Advances in Intrusion Detection*, A. Valdes and D. Zamboni, Eds., vol. 3858 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 247–264.

[29] Huang, J., Liu, P., and Zhang, C. Leap: lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering* (2010), ACM, pp. 207–216.

[30] Jha, A. K., Jeong, S., and Lee, W. J. Value-deterministic search-based replay for android multithreaded applications. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems* (New York, NY, USA, 2013), RACS '13, ACM, pp. 381–386.

[31] Jones, K. J. Forensic analysis of internet explorer activity files. `http://www.mcafee.com/us/resources/white-papers/foundstone/wp-pasco.pdf`.

[32] King, S. T., Dunlap, G. W., and Chen, P. M. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 1–1.

[33] Labroche, N., Lesot, M.-J., and Yaffi, L. A new web usage mining and visualization tool. In *Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on* (2007), vol. 1, pp. 321–328.

[34] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[35] Mickens, J., Elson, J., and Howell, J. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX Conference on Networked Sys-*

tems Design and Implementation (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 11–11.

[36] Narayanasamy, S., Pereira, C., Patil, H., Cohn, R., and Calder, B. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2006), SIGMETRICS '06/Performance '06, ACM, pp. 216–227.

[37] Neasbitt, C., R.Perdisci, Li, K., and Nelms, T. Clickminer: Towards forensic reconstruction of uesr-browser interactions from network traces. In *Proceedings of the 2014 ACM Computer and Communication Security Conference (CCS)* (2014).

[38] Rsa netwitness. `https://www.emc.com/collateral/data-sheet/rsa-netwitness-nextgen.pdf`.

[39] Oh, J., Lee, S., and Lee, S. Advanced evidence collection and analysis of web browser activity. *Digit. Investig. 8* (Aug. 2011), S62–S70.

[40] Panda. `https://github.com/moyix/panda/blob/master/docs/record_replay.md`.

[41] Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K. H., and Lu, S. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 177–192.

[42] Pierrakos, D., Paliouras, G., Papatheodorou, C., and Spyropoulos, C. D. Web usage mining as a tool for personalization: A survey. *User Modeling and User-Adapted Interaction 13*, 4 (2003), 311–372.

[43] Pilli, E. S., Joshi, R. C., and Niyogi, R. Network forensic frameworks: Survey and research challenges. *Digital Investigation 7*, 1 (2010), 14–27.

[44] Reis, C., and Gribble, S. D. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, ACM, pp. 219–232.

[45] Ronsse, M., and De Bosschere, K. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst. 17*, 2 (1999), 133–152.

[46] Russinovich, M., and Cogswell, B. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1996), PLDI '96, ACM, pp. 258–266.

[47] Saito, Y. Jockey: A user-space library for record-replay debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging* (New York, NY, USA, 2005), AADEBUG'05, ACM, pp. 69–76.

[48] Selenium webdriver. `http://docs.seleniumhq.org/projects/webdriver/`.

[49] Sridharan, M., Dolby, J., Chandra, S., Schäfer, M., and Tip, F. Correlation tracking for points-to analysis of javascript. In *Proceedings of the 26th European conference on Object-Oriented Programming* (Berlin, Heidelberg, 2012), ECOOP'12, Springer-Verlag, pp. 435–458.

[50] Srinivasan, S. M., Kandula, S., Andrews, C. R., and Zhou, Y. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track* (2004), Boston, MA, USA, pp. 29–44.

[51] Srivastava, J., Cooley, R., Deshpande, M., and Tan, P.-N. Web usage mining: discovery and applications of usage patterns from web data. *SIGKDD Explor. Newsl. 1*, 2 (2000), 12–23.

[52] Timelapse htmlparser. https://github.com/burg/timelapse/blob/timelapse/Source/WebCore/html/parser/HTMLDocumentParser.cpp; see "// The timing of yields is nondeterministic, so just don't yield during capture/replay".

[53] Timelapse wiki. https://github.com/burg/timelapse/wiki/Frequently-asked-questions.

[54] Townsend, K. R&d: The art of social engineering. *Infosecurity 7*, 4 (2010), 32–35.

[55] Adding traces to chromium. http://www.chromium.org/developers/how-tos/trace-event-profiling-tool/tracing-event-instrumentation.

[56] V8 javascript engine. https://developers.google.com/v8/.

[57] Veeraraghavan, K., Lee, D., Wester, B., Ouyang, J., Chen, P. M., Flinn, J., and Narayanasamy, S. Doubleplay: Parallelizing sequential logging and replay. *ACM Transactions on Computer Systems (TOCS) 30*, 1 (2012), 3.

[58] VMWare Inc. Replay debugging on linux, October 2009. http://www.vmware.com/pdf/ws7_replay_linux_technote.pdf.

[59] The webkit open source project. https://www.webkit.org.

[60] Webview. http://developer.android.com/guide/webapps/webview.html.

[61] Wikipedia - acid3. http://en.wikipedia.org/wiki/Acid3.

[62] Wu, K.-L., Yu, P., and Ballman, A. Speedtracer: A web usage mining and analysis tool. *IBM Systems Journal 37*, 1 (1998), 89–105.

[63] Wundsam, A., Levin, D., Seetharaman, S., Feldmann, A., et al. Ofrewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference* (2011).

[64] Xie, G., Iliofotou, M., Karagiannis, T., Faloutsos, M., and Jin, Y. Resurf: Reconstructing web-surfing activity from network traffic. In *IFIP Networking Conference, 2013* (2013).