WEB-BASED FORENSICS & ATTACK INVESTIGATIONS

Thesis committee:

Dr. Wenke Lee
School of Cybersecurity & Privacy
Georgia Institute of Technology

Dr. Brendan Saltaformaggio
School of Cybersecurity & Privacy
Georgia Institute of Technology

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. Roberto Perdisci
School of Computer Science
University of Georgia

Dr. Paul Pearce
School of Cybersecurity & Privacy
Georgia Institute of Technology

Date approved: May 17, 2023
An uncharted journey of discovery does not merely illuminate the path of knowledge; it also refracts the brilliance of our collective curiosity, revealing a spectrum of truths that resonate within the harmonious symphony of science. In our pursuit of truth, we are not just scholars, but explorers of the infinite landscape of human understanding.

GPT-4
I dedicate this dissertation to my loving wife, Xinlei, and my supportive parents, Stacey and Joe Allen. Your unwavering faith, enduring love, and constant encouragement have been my guiding stars throughout this academic journey. This work is a testament to your belief in me; it wouldn’t have been possible without you.
Firstly, I would like to express my deepest gratitude to my advisor, Dr. Wenke Lee, who has provided me with unfaltering guidance and support throughout my doctoral study. His unending patience, invaluable critiques, and insightful dialogues have not only shaped this dissertation but also significantly influenced my growth as a researcher. My sincere thanks also go to my thesis committee members, Alex Orso, Paul Pearce, Brendan Saltaformaggio, and Roberto Perdisci, for their valuable input and contributions. Their wide-ranging expertise and thorough reviews have significantly strengthened the quality of my work.

I would like to extend special recognition to Roberto Perdisci for the inspiration, wisdom, and insights he has generously shared throughout this process. His deep understanding and passion for research have been a guiding light and a constant source of motivation in my PhD journey.

I am also deeply indebted to my lab mates Ian Yang, Matthew Landen, Yang Ji, and Feng Xiao. The collaborative spirit, collective wisdom, and tireless assistance you provided were instrumental in making this work possible. Thank you for the long discussions, brainstorming sessions, and shared experiences that we have gone through together in our lab, they will always be cherished.

In conclusion, this journey would not have been possible without the collective support, guidance, and encouragement of these extraordinary individuals. I feel privileged to have had the opportunity to work with and learn from all of them. My deepest appreciation is extended to each one of them and their contributions to this dissertation, and more importantly, to my intellectual and personal development.
# TABLE OF CONTENTS

Acknowledgments .............................................................. v

List of Tables ........................................................................ x

List of Figures ......................................................................... xii

Summary ................................................................................ xiii

## Chapter 1: Introduction ..................................................... 1

1.0.1 MNEMOSYNE & Web-Based Attack Provenance .......... 3

1.0.2 WEBRR ..................................................................... 4

1.0.3 Defense Road Map ..................................................... 5

## Chapter 2: Literature Review ............................................ 6

2.1 Data Provenance ........................................................... 6

2.1.1 Data Provenance Auditing Systems ......................... 7

2.1.2 Using Data Provenance for Attack Detection & Attack Reconstruction .................................................. 9

2.2 Forensic Record & Replay ............................................. 10

2.2.1 Web Forensics ......................................................... 12

## Chapter 3: MNEMOSYNE: An Effective and Efficient Postmortem Watering Hole Attack Investigation System .......... 13

vi
3.1 Introduction .......................................................... 13
3.2 Motivating Example & Challenges ............................. 16
  3.2.1 2017 ASEAN Watering Hole Attack ....................... 16
  3.2.2 Challenges ..................................................... 16
3.3 MNEMOSYNE .................................................... 18
  3.3.1 Overview ...................................................... 18
  3.3.2 Threat Model and Assumptions ............................ 20
  3.3.3 Browser-level Causality Graph ............................ 21
  3.3.4 Auditor Daemon .............................................. 23
  3.3.5 Versioning System .......................................... 23
  3.3.6 User-Level Analysis ....................................... 26
3.4 Evaluation .......................................................... 30
  3.4.1 Data Collection .............................................. 30
  3.4.2 Datasets ..................................................... 32
  3.4.3 Attack Scenario Investigation .............................. 35
  3.4.4 Benign Version Analysis .................................. 40
  3.4.5 Runtime Performance ...................................... 41
  3.4.6 Storage Overhead .......................................... 45
  3.4.7 Limitations .................................................. 46
3.5 Related work ....................................................... 47
3.6 Conclusion .......................................................... 48

Chapter 4: WEBRR: A Forensic System for Replaying and Investigating Web-Based Attacks in The Modern Web ................. 49
4.1 Introduction ......................................................... 49
4.2 Background ....................................................... 53
  4.2.1 Render Process Architecture ......................... 53
  4.2.2 Service Workers ............................................ 54
4.3 Motivating Example & Our Approach .................. 54
  4.3.1 Our Approach ................................................ 56
4.4 WEBRR .............................................................. 58
  4.4.1 JavaScript Execution Unit Partitioning ............. 59
  4.4.2 Replaying JEUs .............................................. 60
  4.4.3 Maintaining the DOM State ......................... 64
  4.4.4 Handling Sources of Nondeterminism .......... 65
  4.4.5 Replaying Service Workers ......................... 68
4.5 Evaluation .......................................................... 68
  4.5.1 Experimental Setup ..................................... 69
  4.5.2 Record & Replay Evaluation ......................... 71
  4.5.3 Comparison with Existing Systems ................. 75
  4.5.4 Runtime Overhead ....................................... 77
  4.5.5 Storage Overhead ....................................... 80
4.6 Discussion & Limitations ..................................... 81
4.7 Related Work .................................................... 82
4.8 Conclusion .......................................................... 83

Chapter 5: Conclusion ................................................ 84
LIST OF TABLES

3.1 Browser-based provenance graph objects, relationships, and key attributes. . 21

3.2 The Chromium DevTools Protocol WEBRR relies upon to capture the neces-
   sary events to reconstruct sophisticated browser-based attacks. . . . . . . 22

3.3 The set of TTPs used in Mnemosyne’s weighting system. . . . . . . . . . . 25

3.4 The metadata properties of a user-level version. . . . . . . . . . . . . . . 26

3.5 Description of each attack scenario and the corresponding case in the wild. . 31

3.6 Graph statistics for each attack scenario. . . . . . . . . . . . . . . . . . . . 32

3.7 A detailed performance comparison between manual++ and MNEMOSYNE
   in terms of number of domains and scripts a forensic analyst needs to inves-
   tigate. Scripts† are all scripts related to the corresponding domain. Scripts‡
   are the set of scripts that have behaviors attributed to them (e.g., network re-
   quests, DOM insertions, etc.). MNEMOSYNE can reduce the analysis scope
   significantly, for example, (-99.56%) means reduction based on the raw data. 35

3.8 A report of the user-level version types generated during each attack sce-
    narios. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

3.9 The 15 websites used for the false positive analysis of the version prioriti-
    zation of MNEMOSYNE. . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

3.10 The 10 websites used for the Storage Overhead evaluation (subsection 3.4.6)
    of MNEMOSYNE. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.1 Evaluation of replayed benign websites. For each site, we show the recording operating system (OS), recorded JEU sequence length (Rec-JEU Seq. Len.), replayed JEU sequence length (Rep-JEU Seq. Len.), JEU sequence edit distance (JEU-Seq. E. D.), recorded API sequence length (Rec-API Seq. Len.), replayed API sequence length (Rep-API Seq. Len.), API sequence edit distance (API Seq. E. D.).

4.2 Web-Based Attack Results. For each attack, we show the recording operating system (OS), the recorded JEU sequence length (Rec-JEU Seq. Len.), replayed JEU sequence length (Rep-JEU Seq. Len.), JEU sequence edit distance (JEU-Seq. E. D.), recorded API sequence length (Rec-API Seq. Len.), replayed API sequence length (Rep-API Seq. Len.), API sequence edit distance (API Seq. E. D.), and whether the attack was replayed correctly (Replayed).

4.3 The experimental results for visual determinism.

4.4 Evaluation Results of WebRR and WebCapsule (WC) correctly replaying the web-based attacks. A ✓ in the Visual Component column means the system correctly replayed the visual component of the replay. A ✓ in the Interactions section means the system correctly replay the user interactions with the attack. A ✓ in the Attack Replayed means the visual component and the user-interactions were both correctly replayed. * means the attack was backported and – means the attack could not be recorded.
LIST OF FIGURES

1.1 Web-Based Attack Investigation Framework. . . . . . . . . . . . . . . . . . 1

3.1 Overview of MNEMOSYNE’s architecture. . . . . . . . . . . . . . . . . . . 19

3.2 MNEMOSYNE’s browser-based causality log graph of the social-engineering component used by APT-32 to attack targeted visitors. . . . . . . . . . . . . 22

3.3 Three subgraphs related to visiting hxxps://www.asean.org during the window-of-compromise. (a) represents a visit by a non-targeted user, (b) represents a visit by a targeted user, and (c) represents a visit by a victim of the attack. 26

3.4 Five-point-summary of domain versions generated for each website for the Categories and Alexa datasets, with whiskers being 1.5x IQR. . . . . . . . . . . . 42

3.5 The runtime performance overhead induced on the page load by MNEMOSYNE for the Alexa 1k. (a) presents the runtime overhead increase for the page load. (b) provides the absolute time induced by MNEMOSYNE. Whiskers are set to (0%, 95%). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

4.1 An overview of WEBRR’s Replay Dispatcher running on the render thread. 60

4.2 A high-level overview of the workflow that occurs when the replay dispatcher is executed on the render thread. . . . . . . . . . . . . . . . . . . . . 61

4.3 The runtime performance overhead induced on the page load by WEBRR for the Tranco 1k. (a) The runtime overhead increase for the page load. (b) The absolute time induced by WEBRR. Whiskers are set to (0%, 95%). . . 78

4.4 The runtime resource usage induced on the page load by WEBRR for the Tranco 1k. (a) the distribution of the CPU usages. (b) provides distribution of the memory consumed by WEBRR per process. WEBRR induced negligible CPU overhead and at most 150MB memory (with 15MB overhead maximum) when visiting the Tranco 1k. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
SUMMARY

When a data breach transpires, forensic investigators swing into action to unravel the adversary’s activities within the enterprise network, necessitating the elucidation of attack-induced damages, identification of sensitive resources accessed by the adversary, and formulation of future defense strategies. The rigorous examination often hinges on the organization’s audit logs, which provide insights into each stage of the cyber-kill chain. Addressing this, researchers have devised sophisticated auditing systems that record complete system data provenance. However, a notable drawback is the semantic-gap issue, resulting in limited visibility into web-based attacks, a critical flaw considering the increasing prevalence of such attacks, often used by nation-state adversaries for initial penetration and compromise of enterprise networks. To address this limitation, this thesis presents a web-based attack investigation framework for forensic analysis of web-based attacks, both statically and dynamically, in a postmortem manner. The framework involves a web-based auditor that passively collects audit logs from user browsing sessions at an enterprise level, storing them on a logging server for later analysis. If a data breach occurs, these logs can help determine the root causes and implications of the attack. For static analysis, the logs can be transformed into a causality graph for thorough causality analysis. To demonstrate this, we propose MNEMOSYNE, a system utilizing audit logs to reconstruct, investigate, and assess the impacts of watering hole attacks. For dynamic analysis, the framework produces replayable causality logs, enabling auditors to identify suspicious events and replay the attack site. To achieve this, we developed WEBRR, a novel, OS- and device-independent record-and-replay forensic auditing system for Chromium-based web browsers, allowing an investigator to dynamically analyze the attack through replaying the event postmortem.
CHAPTER 1
INTRODUCTION

Figure 1.1: Web-Based Attack Investigation Framework.

Over the past ten years, there has been a significant rise in sophisticated attacks on businesses and government bodies. Notably, incidents targeting organizations surged by over 15% in 2021 alone, according to a Forbes report [1]. Moreover, data breaches have not only become more frequent but also more costly for those affected. For instance, the widespread SolarWinds breach, which hit more than 18,000 organizations, is projected to require over $90 billion for comprehensive containment and clean-up efforts [2]. Simultaneously, there’s mounting pressure from governmental bodies on organizations to thoroughly comprehend the implications of these breaches. This pressure stems from the privacy risks for individuals whose personal data was accessed during these attacks. As of now, 137 countries have data protection legislation in place, and this number is set to increase [3]. Consequently, the necessity for a thorough forensic analysis post-data breach has never been greater. This involves comprehending the full extent of the attack and identifying which data the attackers accessed. Forensic investigators, usually dispatched to the victim organization, must uncover what transpired while the attacker was active within the network. This involves determining the extent of the damages, identifying which sensitive resources the attacker compromised, and planning for future defenses. In essence, these
investigators often have to comb through the organization’s audit logs to understand how the attacker navigated each stage of the cyber-kill chain [4]. This is crucial for learning from past incidents and for fortifying the organization’s defenses against future attacks.

The rising demand for post-mortem analysis at enterprise organizations has led to the development of sophisticated auditing systems. These advanced systems improve the accuracy and efficiency of the analysis by recording comprehensive data provenance. This allows them to monitor the flow of information throughout an organization. However, these system-level data provenance auditing systems are not without their shortcomings. A significant limitation is their lack of detailed visibility into web-based attacks. This is due to a substantial semantic gap between system-level abstractions (like processes, sockets, and files) and the specific semantics needed to investigate web-based attacks (for example, the rendering of HTML/CSS and execution of JavaScript). This inadequate visibility into web-based attacks is increasingly worrying, especially as these types of attacks have become a favorite tool of nation-state adversaries to breach and initially compromise enterprise networks. A case in point is the recent attack by APT32, a nation-state adversary, which used a “watering hole” attack (an attack that compromises a website, turning it into a stepping stone to the victims’ network) and a malicious OAuth application to compromise several high-profile non-governmental organizations (NGOs), including ASEAN and Vietnamese politicians [5]. Unfortunately, the existing system-based auditing systems would struggle to thoroughly analyze this type of attack due to the aforementioned semantic gap issue.

In this thesis, we introduce a web-based attack investigation framework designed to address the challenges previously mentioned. This framework provides forensic analysts with the tools to investigate web-based attacks both statically and dynamically, in a retrospective manner. The structure of this framework can be seen in Figure 1.1. The process begins with the web-based auditor, which passively gathers audit logs related to a user’s browsing activity from devices used for web browsing within a corporate environment. These logs are then transmitted to a storage server. In the event of a data breach, these
logs can be employed by an investigator to help identify the root causes and implications of web-based attacks. The audit logs gathered by the auditor are adaptable for both static and dynamic analysis. To facilitate static analysis, the logs can be transformed into a causality graph. This transformation enables analysts to carry out a causality analysis, assisting them in comprehending the origins and outcomes of the attack. To illustrate this feature, we present Mnemosyne, a system that uses the audit logs from the web auditor to accurately reconstruct, investigate, and evaluate the fallout from watering hole attacks. For dynamic analysis, we have enhanced the audit log monitor to generate replayable causality logs. This enables the auditor to identify suspicious events and conduct a thorough retrospective replay of the website where the event took place. To replay these logs, we have developed WEBRR, a unique record-and-replay (RR) forensic auditing system that’s compatible with all operating systems and devices for Chromium-based web browsers. This system allows an investigator to replay notable events and conduct a dynamic analysis on the attack by replaying it in a retrospective manner. Further details about Mnemosyne and WEBRR are discussed in the following sections of this thesis.

1.0.1 Mnemosyne & Web-Based Attack Provenance

Compromising a website that is routinely visited by employees of a targeted organization has become a popular technique for nation-state level adversaries to penetrate an enterprise’s network. This technique, dubbed a “watering hole” attack, leverages a compromised website to serve as a stepping stone into the true victims’ network. Recently, watering hole attacks have been employed in multiple state-level cyber crimes to conduct digital-espionage in southeast Asia [5], steal proprietary information from large tech firms such as Google and Apple [6, 7], and leak confidential financial information in Poland and Mexico [8, 9].

Unfortunately, existing auditing systems [10, 11, 12, 13, 14, 15] are not suitable for reconstructing watering hole attacks due to the semantic-gap issue previously discussed.
In order to address this, we propose Mnemosyne. Mnemosyne is a postmortem watering hole attack investigation engine that can be used at an organization to 1) passively collect web-based audit logs without requiring any instrumentation to the underlying OS or the browser and 2) Mnemosyne’s forensic analysis engine can be used for postmortem analysis of watering hole attacks. First, Mnemosyne relies on extracting audit logs from the browser in terms of browser-based semantics (e.g., JS, HTML, and DOM information). Mnemosyne then leverages these logs to precisely identify exactly when the point-of-compromise of the external website occurred. It then applies a set of versioning techniques on top of these causality logs to precisely pinpoint when the website was compromised and what modifications were made by the adversary. Following this step, Mnemosyne relies on a novel user-level analysis to assess how the malicious modifications affected the targeted enterprise and seeks to identify exactly which employees fell victim to the attack. Throughout our evaluation, we found that Mnemosyne’s forensic analysis engine was able to identify the true victims in all 7 real-world watering hole scenarios, while also reducing the amount of manual analysis required by the forensic analyst by 98.17% on average.

1.0.2 WebRR

In practice, having the capability to record and replay web-based attacks is highly desired during a forensic analysis, because it allows the investigator to replay exactly what occurred during the attack in a postmortem fashion, thus facilitating the reconstruction of the attack’s root causes. In order to address this, we propose a novel system named WebRR, an always-on forensic auditing system that enables deterministic record and replay of modern web applications. WebRR works by embedding its record and replay engine into Blink (Chromium’s rendering engine) and at the interface between Blink and V8 (Chromium’s JavaScript engine). In particular, WebRR records units of JavaScript (JS) code execution along with the state of the DOM at the time when the JS code ran. During replay, WebRR
makes sure that (1) JS execution units are replayed in order, and (2) the state of the DOM before the execution of each JS unit is exactly the same as it was during replay. We show that this approach achieves deterministic replay and demonstrate the replay of a number of real-world attacks, while also showing that prior approaches fail to correctly replay these attacks. Furthermore, we demonstrate that WebRR can also record highly-dynamic modern websites in a deterministic fashion with an average runtime overhead of only 3.44%.

1.0.3 Defense Road Map

In the remainder of this defense, I will provide a literature review of the existing work in this area in chapter 2. Next, I will discussed Mnemosyne in detail in chapter 3. Finally, I will discuss WebRR in chapter 4. This is followed by a brief conclusion in chapter 5.
CHAPTER 2
LITERATURE REVIEW

2.1 Data Provenance

With the recent rise in enterprise data breaches, it is important that forensic investigations are carried out to fully understand how an adversary achieved each stage of the cyber-kill chain [4]. This has lead to researchers developing state-of-the-art auditing systems that are capable of reconstructing attacks in a postmortem fashion [16, 17, 18, 19, 12, 10, 11, 13]. The main goals of these systems are to answer, how was this system compromised and what resources were accessed by the adversaries? In order to answer these questions, researchers have relied on tracking the data provenance [20] of a system. Tracking the data provenance of a system typically relies on capturing how information flows through a system. By capturing this information, it allows forensic investigators to build a data-provenance graph in terms of system-level objects (processes, sockets, and files) and data-flow relationships (e.g., read, write, recv, send) between these objects. Next, during an investigation, backwards queries operations can be made on this graph to determine exactly how the adversary initially compromised the system. Additionally, a forward query on this data provenance graph can be used to identify all resources that were involved in the attack, essentially allowing the forensic investigator to determine which resources were accessed by the adversary. This technique was originally proposed by King et. al. as a technique system admins could use to backtrack intrusions and reconstruct how a system was compromised [16]. Since then, using data provenance for attack detection and reconstruction has been extensively explored to fully understand its capability for intrusion detection and attack reconstruction [21, 27, 16, 18, 19, 12, 35, 36, 37, 10, 11, 15, 22, 14, 23, 24, 13, 25, 26, 28, 29, 30, 31, 32, 33, 34]. Research in this area broadly falls into two categories:
1) developing techniques to record data provenance information on end-point systems at an enterprise [16, 18, 19, 12, 10, 11, 15, 22, 14, 23, 24, 13] and 2) developing systems that leverage the collected data provenance for intrusion monitoring and attack reconstruction [25, 26, 28, 29, 30, 31, 32, 33, 34]. In the remainder of this section, we will describe systems that are responsible for collecting data provenance in subsection 2.1.1 and how this information is used for attack detection and reconstruction in subsection 2.1.2.

2.1.1 Data Provenance Auditing Systems

As previously discussed, data provenance auditing systems seek to track information-flow through a system. At a high-level, this is achieved by monitoring all applications running on the system and emitting audit logs in the form of tuples, \((p, s, r)\), where \(p\) is the process that invoked the system call system call \(s\) on the resource \(r\). These audit logs are then passed to upstream causality analysis systems, which will use them for attack detection and reconstruction. For now, we will focus on the main challenges of collecting these audit logs, and discuss the upstream analysis in subsection 2.1.2.

The first challenge is identifying the correct layer in the OS stack to collect these audit logs in a secure fashion. While prior systems have proposed to collect data provenance at the application layer [38, 39, 40], this is not appropriate from a security manner, because it would need to assume applications running on the machine are part of the trusted-computing base (TCB). In order to address this, Bates et. al proposes Linux Provenance Modules (LPM) which introduces techniques to capture provenance information in the kernel layer of the OS stack [41]. This has two major advantages. First, it allows for the collection of provenance information of untrusted applications. The second major advantage is that by collecting provenance information at the kernel-layer it allows these systems to support a broad set of applications without any instrumentation. For the most part, the majority of systems in data provenance have gone in this direction and now collect data provenance within the kernel itself [42, 10, 11, 31, 12, 43, 44]. In fact, the Linux
The next major challenge in collecting data-provenance audit logs is the issue of dependency explosion [15]. Dependency explosion is a phenomenon that largely impacts long-running applications (e.g., a web server) and greatly diminishes the capability of upstream causality analysis on the process. The underlying issue is that data provenance queries assume that any data that is written to an external resource depends on all data that has been previously read by an application. This is necessary to guarantee no false negatives but it leads to a significant amount of false-positives. There have been many systems that have attempted to address this issue using various methods [15, 22, 14, 10]. First, BEEP [15] takes advantage of the fact that long-running processes tend to use a event-loop model of most long-running applications. Essentially, BEEP seeks to partition the execution into multiple units of execution, where each unit represents a single turn on an event loop. Additionally, it is assumed that all units of execution can be assumed independent of each other. One limitation of BEEP is that it is not that effective when units of execution are causally dependent on each other (e.g., a I/O thread receives a input and pushes a task onto a background worker thread). In order to address this MPI [14] seeks to leverage source-code annotations, which allow for developers to annotate their code with preprocessor macros that allow MCI to accurately create units of execution. Unfortunately, getting developers to annotate their code in practice is a non-trivial task. UIScope [23] takes a different approach and partitions the application’s execution based on GUI elements of an application. RAIN [10] takes a instrumentation free approach to this problem and relies on record-and-replay technologies along with dynamic-information-flow-tracking (DIFT) to address the problem. Essentially, if false-positives need to be pruned from the causality graph, RAIN relies on replaying the application and applying DIFT to the application to prune out any false dependencies. The major advantage of this approach is that it allows it
to defer the expensive performance hit of DIFT (greater than 30x [45]) to during the replay application, which allows it to have a practical runtime performance.

2.1.2 Using Data Provenance for Attack Detection & Attack Reconstruction

The second major area of work in data provenance research is related to using audit logs emitted by systems discussed in subsection 2.1.1 for attack reconstruction and attack detection. Recently, leveraging data provenance for detecting sophisticated attacks on enterprise networks has been extensively studied by security researchers [21, 27, 29, 25, 28]. For example, Holmes supports detecting APT attacks in real-time by monitoring a data provenance stream to identify Technique Tactics & Procedures (TTPs) that are commonly used by adversaries. Next, Holmes leverages the identified TTPs to develop an abstracted version of the data provenance graph called the high-level scenario graph (HSG). Finally, it will try to determine if this HSG is malicious or benign. Additionally, RapSheet extends Holmes by increasing the number of TTPs that can be Holmes’s detection engine. Nodoze [25] relies on causality information to significantly reduce the number of false positives generated by industry alert systems like Splunk [46]. Specifically, when an alert is generated, Nodoze will do causality analysis to identify if this alert is a false positive or is an alert that warrants further investigation.

In addition to attack detection, another problem in data provenance analysis that has been extensively studied is log-reduction [47, 29, 48, 31]. This is because the size of a provenance graph increases in linear proportion to the number of executed system calls on a system. Unfortunately, this magnitude of data makes existing analysis approaches infeasible in real-world scenarios, because the analysis is computationally intractable. To reduce the data footprint of provenance data, several approaches have been proposed [47, 29, 31, 48]. The majority work in this area largely focuses on removing redundancy in the audit logs. An audit log is considered redundant if a prior log subsumes any provenance information the redundant log could provide. For example, LogGC [47] removes redun-
dancy by routinely removing redundancy in the provenance graph in a similar fashion as a garbage collector. Additionally, Kcal [31] customizes the Linux kernel in order to avoid emitting audit logs that only contain redundant information. Finally, Sleuth [29] relies on creating a compact data structures and variable-width encoding to reduce the size of the provenance graph. However, Sleuth’s approach is mainly focused on reducing the size of the in-memory representation of the graph to improve the analysis time.

2.2 Forensic Record & Replay

The majority of systems discussed in subsection 2.1.1 are record-only auditing systems and they cannot support post-mortem dynamic analysis. In practice, having the capability to record and replay attacks is highly desired during a forensic analysis, because it allows the investigator to replay exactly what occurred during the attack in a postmortem fashion, thus facilitating the reconstruction of the attack’s root causes. In this section we discuss works that have relied on record-and-replay to support forensic analysis of attacks.

While there is a storied history of developing record and replay systems, the majority of prior work is largely focused on developing systems to improve the debugging and testing experience [49, 50, 51, 52, 53]. For example, Jalangi [52] is a heavyweight dynamic analysis framework that relies on selective record-and-replay to allow developers to test their application by only replaying selected portions of the original execution. Additionally, MugShot also supports record-and-replay of JS applications by injecting itself into the web application (as the first script to be executed). Next, Mugshot will inject instrumentation hooks to capture all sources of non-determinism to the application. A major advantage of Mugshot’s approach is that it has fairly low overhead of about 7% which makes it deployable in real-world environments [50]. Finally, RR [49] take advantage of ptrace to implement a general purpose debugging tool that supports replaying many application without requiring any modifications to the kernel or application. Unfortunately, existing debugging-based record-and-replay systems are not suitable for recording and replaying
web-based attacks. For example, the popular debugging tool RR [49] achieves a deterministic replay by restricting threads to be only ran on a single CPU. Restricting the threads to a single CPU allows it to avoid having to synchronize the threads during the replay, but it comes at the expense of the performance overhead (greater than 7x in some cases [49]). This overhead is fine in debugging scenarios, but it would be completely impractical to deploy this in an always-recording fashion at an enterprise environment. Additionally, [50, 52] rely on instrumenting the JavaScript (JS) code of the web applications to support record and replay. Unfortunately, instrumenting JS code is unsuitable for a forensic setting, because these systems can easily be detected and in some cases tampered with in an adversarial setting (e.g., the recording may be disabled by the malicious web application).

To overcome the limitations of debugging-based systems, the security community has begun to develop forensic-grade record and replay systems. For example Arnold [42] & RAIN [10] are forensic-based record-and-replay systems that rely on instrumenting the kernel to support record-and-replay of user-space applications. In addition, these systems support allowing a forensic analyst to attach debugging tools to the replayed execution to facilitate fine-grained dynamic analysis. Additionally, RTAG extends the support of prior work to support replaying attacks that occur across multiple host-machines. However, these systems only support an analysis at the system level and are not suitable for investigating web-based attacks due to the semantic-gap between system-level and JS semantics. Therefore, these systems limit the forensic analyst’s ability to investigate web-based attacks, due to the semantic gap between the system- and web-level abstractions (e.g., they do not support attaching a JS debugger during browser replay). In order to address this, WebCapsule [36] supports record and replay of web-based attacks. WebCapsule addresses the semantic-gap issue by embedding its replay-engine into the browser itself and then records all sources of non-determinism that flows into Chromium’s rendering engine, Blink [54].
2.2.1 Web Forensics

A major limitation of the systems described in subsection 2.1.1 is that they are not that effective at investigating web-based attacks that reside within the browser. The underlying issue is that these systems capture auditing information in terms of low-level semantics (e.g., processes, files, and sockets). This is necessary for these systems to support auditing a large number of applications, but it greatly limits their capability of reconstructing web-based attacks. This is because in order to fully understand the intricate details web-based attacks, fine-grained details related to the browser’s execution is necessary. This has inspired an additional line of work, which is web-based forensics [37, 35, 36]. ChromePic [37] supports web-based auditing by taking screenshots in a synchronous manner during a user’s browsing session. The main advantage of this approach is that it allows a forensic investigator to observe the visual component of an attack. However, a major limitation of this work is that it cannot provide the forensic analyst with any fine-grained details related to the javaScript execution of an attack. This means attacks that do not have any ”visual components” (e.g., drive-by downloads and keyloggers) cannot be investigated using ChromePic. JSGraph [35] overcomes this by developing a in-browser based auditing system which captures fine-grained information related to the DOM of a website and its javaScript execution. JSGraph also collects causality information, which allows it to develop a causality graph (analogous to a data provenance graph) that a forensic investigator can use to investigate web-based attacks. One limitation of JSGraph is that it is a *record-only* system and can only support static analysis of attacks. In order to address this, WebCapsule [36] relies on record-and-replay to allow a forensic-based record and replay of web-based attacks. WebCapsule records and replay a user’s browsing session by instrumenting Chromium’s rendering engine to capture all sources of non-determinism. Next, if an attack occurs a forensic investigator use WebCapsule to replay exactly what occurred during the browser at the time of the attack.
3.1 Introduction

Sophisticated, targeted attacks against enterprise networks have been growing more frequent recently. Such attacks often unfold through a sequence of steps sometimes referred to as the cyber kill chain [4]. To deliver the initial attack that compromises the targeted network, attackers leverage different techniques such as sending spear phishing emails to gain a foothold on the victim’s workstation or hoaxing the user to visit a website controlled by the adversary and completing a drive-by-download attack. These tactics have been well studied and enterprises have deployed effective blacklist-based firewall rules (e.g., WAF [55] and Email Defender [56]) and set up periodic security training for its employees. Unfortunately, adversaries have evolved their techniques to infiltrate a targeted enterprise network by compromising whitelisted, third-party entities with which the enterprise normally communicates. For instance, compromising a website that is frequently visited by individuals affiliated with the targeted organization has become a growing trend to achieve the initial intrusion into a targeted organization’s network. Such attacks are referred to as “watering hole” attacks. Recently, watering hole attacks have been employed in multiple state-level cybercrimes to conduct digital-espionage in southeast Asia [5], steal proprietary information from large tech firms such as Google and Apple [6, 7], and leak confidential financial information in Poland and Mexico [8, 9].

Despite watering hole attacks being a key method for achieving the initial compromise into an organization, little research has been conducted to study how to detect, analyze, and investigate these attacks effectively. However, having the capability of completing a thor-
ough postmortem analysis is desired by organizations, since it allows them to understand the attacker’s intentions, prevent additional damage, and provides a mechanism for building future defenses. In this proposal, we propose **MNEMOSYNE**, a system that facilitates comprehensive internal forensic investigation on web-based watering hole attacks.

Detecting watering hole attacks and reconstructing their provenance is challenging. First, there has been a significant amount of work dedicated to attack reconstruction, with most solutions relying on whole-system provenance tracking and attack reconstruction [10, 22, 15, 14, 42, 13, 12, 16, 23, 57, 24, 58, 34, 41, 29, 28, 21, 27, 30, 25, 33]. These systems typically collect audit logs that track the information-flow at the system-level and tend to rely on low-level semantics (processes, socket IO, and system calls), which is necessary in order for them to support as many applications as possible. Unfortunately, existing systems’ reliance on low-level semantics limit their capability of reconstructing sophisticated watering hole attacks. This is because capturing information at the system level is limited in terms of its capability of understanding fine-grained details related to Javascript (JS) execution within the browser. To overcome this semantic gap, **MNEMOSYNE** collects audit logs that capture information in terms of browser-level semantics (e.g. page, script, domain, etc.). While prior browser auditing systems exist [35, 37, 36], they require extensive modifications to the browser itself, making deployment in a real-world scenario difficult. In contrast, **MNEMOSYNE** relies on a browser-modification-free, lightweight approach that takes advantage of existing debugging interfaces already provided by off-the-shelf Chromium-based browsers (e.g., Chrome, Opera, Microsoft Edge, Brave, etc.).

The second challenge in developing a system for investigating watering hole attacks is that watering hole attacks are highly targeted and during the early stages of a forensic investigation, it is unclear which visitors are considered the true targets. To address this challenge, we argue that simply detecting a website is compromised is not enough. Instead, for organizations that routinely visit this compromised website, they need to complete a independent and accurate investigation to determine if visiting this site while it
was compromised had any adverse effects on their own enterprise networks. However, completing this investigation in an independent manner is not straightforward, since the server-side logs related to how and when the compromised website was modified are external and inaccessible to the targeted organization. To overcome this, MNEMOSYNE relies on a lightweight auditing approach that passively collects audit logs during a user’s browsing sessions. Finally, while MNEMOSYNE completes the investigation in a postmortem fashion, it is still necessary to complete the investigation in a time-sensitive manner. This is because during the investigation, the decreased system uptime can easily cost millions of dollars [59]. Additionally, an efficient investigation that allows the investigators to quickly identify the scope of the attack can reduce the overall damage created by the attack. To make the investigation as efficient as possible, MNEMOSYNE applies a set of differential analysis techniques on the audit logs collected to quickly identify which employees at the organization should be considered victims of the attacks and which employees were unaffected by the attack.

In summary, we make the following main contributions:

• **A watering hole attack investigation system.** We propose MNEMOSYNE, a system that is able to accurately reconstruct the provenance and impact of sophisticated watering hole attacks.

• **Browser-modification-free design.** MNEMOSYNE does not require browser code modifications and can therefore be more easily deployed within users’ browsers for collecting detailed web audit logs.

• **Accurate and efficient analysis.** MNEMOSYNE applies a set of versioning and prioritization methods to efficiently reconstruct and analyse enterprise-level watering hole attacks. Using seven scenarios based on real-world security incidents involving watering hole attacks, MNEMOSYNE is able to identify the individuals who were victims of the attack in all of these scenarios with efficient runtime.
3.2 Motivating Example & Challenges

In this section, we describe an attack scenario modeled off of a real-world watering hole campaign that showcases the challenges that MNEMOSYNE addresses.

3.2.1 2017 ASEAN Watering Hole Attack

This case study relates to a real-world, politically-motivated campaign that leveraged a watering hole attack to achieve digital surveillance and espionage on employees and high-powered individuals affiliated with the Association of Southeast Asian Nations (ASEAN), an organization that helps to foster peace between member countries. The attack was carried out in 2017 and was recently attributed to APT32, a nation-state threat actor that is known to carry out cyberattacks against political enemies of the Vietnamese Government [5]. We chose this motivating example because it clearly demonstrates the challenges a forensic analysis will face during postmortem analysis of complex watering hole attacks.

The attack was divided into two stages. The first stage performed reconnaissance by collecting sensitive information related to the user’s browser and underlying system to accurately identify if this visitor matched the profile of the targeted victims. After a visitor’s profile was developed, it was used to identify the targeted visitors. Finally, the targeted visitors were exploited using social-engineering that pursued victims to grant the attackers access to their Gmail accounts via a malicious OAuth App.

3.2.2 Challenges

Next, we discuss the challenges that a forensic analyst faces when completing an internal investigation on a sophisticated watering hole attack and discuss the limitations of existing postmortem analysis systems. The challenges described in this section are the challenges faced by the targeted organization, not the organization hosting the compromised website.
External Point-of-Compromise

During a traditional investigation of a sophisticated attack, the initial point-of-compromise occurs at an endpoint system in the enterprise’s network. The intrusion is usually achieved through spearphishing emails or traditional exploitation techniques. One advantage this provides is that the audit logs related to this attack will be accessible to the victim organization. Unfortunately, this is not the case for watering hole attacks, since the point-of-compromise is external, beginning at a third-party website that is unlikely to be affiliated with the true victim organization. This creates the challenge that the audit logs related to the compromised website and the attacker’s modifications to the site are only accessible by the website’s maintainers, not by the forensic investigator. Due to this limitation, we found that forensic investigators often have to fallback on internet archive sites, such as archive.org or passiveTotal [60] to identify how the compromised domain was modified [61, 62, 63, 64]. However, because watering hole attacks are highly targeted, the machines used for snapshotting the web page will not match the intended victim profile. Finally, since the forensic analyst does not know how the web server was modified, the analyst will begin the investigation with minimal information about the initial compromise.

Also, due to the lack of server-side logs, it is challenging to identify the dwell time, which is the time window in which the attacker controlled the compromised website. Identifying the window-of-compromise is necessary to ensure a comprehensive investigation, since any visit to the compromised website within this time window may have led to a successful attack. Without a clear window-of-compromise, the analyst may have to review irrelevant website traffic logs generated prior to the incident, prolonging the investigation.

Semantic Gap

Another limitation is the semantic gap that exists when completing a postmortem analysis on web-based attacks using only system-level logs. Recently, whole-system provenance auditing has been shown to be effective at investigating sophisticated attacks [10, 11, 12,
These systems typically collect audit logs that track the information-flow at the system-level and rely on low-level semantics to causally connect all artifacts and resources involved in the attack. However, using low-level semantics limits reconstruction of watering hole attacks because the semantic gap between system-level and browser-based semantics prevents a thorough understanding of JS execution. Prior work that has attempted to address this limitation requires extensive modifications to the browser itself [35, 37, 36], which makes deployment in most enterprises difficult.

**Highly Targeted**

Watering hole attacks are highly targeted and the granularity of the adversary’s targets varies with different attacks. For example, watering hole attacks may only target specific victims at an organization, specific departments of an organization, or a set of organizations. Unfortunately, during the early stages of an investigation, the motive of the attack is unknown and identifying who is targeted by the attack and which individuals fell victim to the attack is challenging. However, identifying the victims of the attack is arguably the most important part of the forensic investigation, since it allows the forensic analyst to determine which user sessions they should prioritize. Also, when the forensic analyst spends time investigating logs related to untargeted users, the investigation is prolonged.

### 3.3 **Mnemosyne**

#### 3.3.1 Overview

Mnemosyne is a forensic analysis engine that completes a postmortem analysis from within the targeted organization with minimal external information. The only information that Mnemosyne requires is the domain name of the compromised website. This design choice was made based on the fact that during the early stages of the investigation, information related to malicious domains used by the adversaries or the modifications made to the compromised domain will be limited and potentially inaccessible to the forensic inves-
tigator. Also, the audit logs related to how the website was compromised are external, and in some real-world cases, communication between all entities involved was limited \[65]\).

An overview of our system is provided in Figure 3.1, which illustrates its three essential components. The first component is the browser auditor daemon, which is deployed on each endpoint system at an organization to monitor web-browsing activities. The auditor daemon passively collects audit logs throughout the user’s browsing sessions without the need to alter the browser. Next, the audit logs from each endpoint are collected and stored on a backend server responsible for maintaining security auditing information.

The next module of \textsc{Mnemosyne} is the versioning system that tracks and analyzes the external website’s behavior. The first phase in the analysis is the domain versioning system, which works with the browser-level audit logs to determine when the website was compromised and what modifications the adversaries made to the website. Specifically, the versioning system reconstructs how the compromised website changed over time. Notably, our goal is not to create a single version each time a minor change is observed in the underlying audit logs. Instead, our versioning system helps the forensic analyst quickly identify the window-of-compromise, or the version that includes the adversary’s modifications to the compromised website that introduced some attack-controlled content.
Next, MNEMOSYNE provides a version-prioritization approach that prioritizes versions based on their likelihood to be the version that truly represents the window-of-compromise. Developing a prioritization scheme is essential because prior studies have shown that the dwell time can be excruciatingly long, in some cases lasting over 53 months [66]. Meanwhile, benign updates, which will lead to MNEMOSYNE generating new versions, will also occur, which leads to a challenge in identifying which domain-version actually represents the window of compromise and which versions are related to the natural evolution of the website. The last stage of MNEMOSYNE’s analysis is its user-level analysis module, which takes a suspicious domain version and identifies how this domain version behaved differently based on the user that was visiting the site.

3.3.2 Threat Model and Assumptions

We envision MNEMOSYNE being deployed in enterprise organizations that have a high risk of being targeted by a sophisticated, nation-state-level attacker. MNEMOSYNE is capable of logging details about users’ browsing activities. This means a trade-off between security and privacy must be found. In the envisioned deployment scenarios, it is a reasonable assumption that the executives would be willing to accept a potential reduction to employee privacy to achieve a higher level of security. Furthermore, the audit logs captured by MNEMOSYNE can be encrypted and securely stored on a file server. A different encryption key can be used for each website and for different time windows. These keys can then be stored in a key escrow, as proposed in previous work [36]. This allows the release of only those keys that are truly needed to enable a forensic investigation.

We also assume that the browser audit logs are stored securely, e.g. using append only log files [67], and thus cannot be tampered with even if the browser is later compromised. Also, we assume that at the time of the watering hole attack, the browser itself is not compromised and the audit logs can be trusted as correct (note that assuming the integrity of the trusted-computing based (TCB) is common in the auditing community [21, 27, 16,
Table 3.1: Browser-based provenance graph objects, relationships, and key attributes.

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Attributes</th>
<th>Relationship</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame</td>
<td>securityOrigin,sessionId, URL</td>
<td>Attached</td>
<td>Frame → Frame</td>
</tr>
<tr>
<td>Iframe</td>
<td>securityOrigin,sessionId, URL</td>
<td>CompiledBy</td>
<td>Script → Frame</td>
</tr>
<tr>
<td>Remote Host</td>
<td>2nd-level domain, domain</td>
<td>Created</td>
<td>Script → Frame</td>
</tr>
<tr>
<td>File</td>
<td>path, remoteOrigin</td>
<td>Download</td>
<td>Frame → File</td>
</tr>
<tr>
<td>Resource</td>
<td>URI, type</td>
<td>Navigated</td>
<td>Frame → Frame</td>
</tr>
<tr>
<td>Script</td>
<td>hash, sourceOrigin, URL, sessionId</td>
<td>Opened</td>
<td>Frame → Frame</td>
</tr>
<tr>
<td>Session</td>
<td>user-agent, timestamp</td>
<td>Request</td>
<td>Parser → Resource</td>
</tr>
<tr>
<td>HTML Parser</td>
<td>-</td>
<td>Response</td>
<td>Resource → Script</td>
</tr>
<tr>
<td>User</td>
<td>-</td>
<td>Located</td>
<td>Resource → Host</td>
</tr>
</tbody>
</table>

(a) Graph Objects: each object has a unique ID. The bolded attribute represents the object’s identifier.

(b) Relationship between objects.

18, 19, 12, 35, 36, 37, 10, 11, 15, 22, 14, 23, 24, 13, 25, 26, 28, 29, 30, 31, 32, 33, 34]).

If the browser is compromised, MNEMOSYNE can still record correct audit logs related to the attack up until the point when the browser is exploited, thus allowing a forensic analyst to reconstruct the attack setup phase. In our extensive evaluation, we demonstrate that by only recording the “setup” phase of a drive-by download attack campaign MNEMOSYNE is still capable of identifying the victims of the attack.

3.3.3 Browser-level Causality Graph

As a first step to enable attack reconstruction, we construct a causality graph based on the browser-level audit logs, which will be used during the postmortem analysis and investigation. The browser objects are defined as nodes in the audit graph, as in Table 3.1a, and the causal relationships between the objects are defined as edges in the graph, as listed in Table 3.1b. The graph presents the chain of browser events that occurred and the causal relations they induced. To demonstrate MNEMOSYNE’s capability to reconstruct a web-based attack, we demonstrate how MNEMOSYNE’s logs are able to reconstruct the social-
Figure 3.2: MNEMOSYNE’s browser-based causality log graph of the social-engineering component used by APT-32 to attack targeted visitors.

Table 3.2: The Chromium DevTools Protocol WEBRR relies upon to capture the necessary events to reconstruct sophisticated browser-based attacks.

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Events Received</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network</td>
<td>responseReceived, requestWillBeSent</td>
</tr>
</tbody>
</table>
| Page       | frameAttached, frameNavigated, downloadWillBegin, windowOpen, javascript-Dialo
gOpening |
| Debugger   | scriptParsed                                         |
| Target     | targetCreated, attachedToTarget, targetInfoChanged   |

engineering component of the motivating example in Figure 3.2. The social-engineering attack has three major stages. The first stage fingerprints the user to identify if they match the targeted profile. If so, the second stage blurs the original site’s content and injects a malicious overlay into the DOM that redirects the user to a malicious, adversary-controlled site. The final stage of the attack occurs when users navigate to the malicious website that contains a malicious OAuth application. If the user is tricked into granting permission to the malicious OAuth application, it grants the adversaries full access to the user’s Gmail accounts.
3.3.4 Auditor Daemon

To ensure our system can be widely deployed in enterprise settings, we take a different approach than the previous systems that alter the browser (e.g., [35, 36, 37]), and rely on the existing debugging interfaces provided by Chrome that do not require extensive modifications to the browser. Specifically, we rely on Chromium’s DevTools interface to extract information about the user’s browsing session. Chromium’s DevTools Protocol allows tools to instrument, inspect, and profile Chromium, Chrome, and other Blink-based browsers. MNEMOSYNE’s auditor daemon collects the necessary information related to the browser’s execution needed to reconstruct the audit logs described in subsection 3.3.3. A list of the Chrome namespaces used to captured this information is in Table 3.2.

3.3.5 Versioning System

The domain-based versioning system takes in the domain name of the website that is suspected to have been compromised to launch a watering hole attack. The domain versioning system has two major components. First, the version reconstruction component reconstructs client-side versions of the compromised domain (subsubsection 3.3.5). Second, the version prioritization component prioritizes the versions in terms of their likelihood of representing the compromised version (subsubsection 3.3.5). These two components are detailed below.

Version Reconstruction

The first step of the version reconstruction component is to refine the audit graph to only include pages related to the compromised domain. Specifically, we create the set $P_{\text{domain}} = \{ p : p.\text{securityOrigin} = \text{domain} \}$ where domain is the compromised domain. Next, for all pages, $p_i \in P_{\text{domain}}$, MNEMOSYNE performs a reachability analysis. The reachability analysis searches the browser causality graph, beginning at $p_i$, to collect all of the involved objects and network events that occurred when loading the page. This query identifies the
domain set, \( D \), which is the set of domains that were communicated with while pages in \( P_{domain} \) were loaded into the browser and identifies the earliest and latest timestamps in which network events were made to a domain \( d_i \in D \). By extracting the timestamps from the relevant network events, MNEMOSYNE can reconstruct a chronology profile, which lists the domains in \( D \) in descending order by timestamps. Next, the version reconstruction component converts the generated chronology report into versions of the website. The versioning system reconstructs versions based on the domains that were communicated with while the page was loaded into the browser. To construct versions of the website, our system breaks up the chronology report into time windows and then aggregates domains together based on the time the domain first interacted with the website. Specifically, when a set of domains fall within the same time window, they are aggregated into the same domain-version, \( D_v \).

**Definition 1** (Domain Version). *Given a time window, \([t_s, t_e]\), and a webpage, \( p_i \), a Domain Version* := \( \{d \in D : p_i \text{ communicated with } d \text{ for the first time when loading } p_i \text{ in } [t_s, t_e]\} \)

Manually inspecting the domain sets to determine the boundaries of new versions is time-consuming and leads to analysis fatigue [25]. For this reason, we automated this process. We first rely on a profiling phase that identifies the profile domain version, \( D_{profile} \), which represents the set of benign domains that are responsible for commonly serving content to visitors of the compromised website. We define the time window required to learn \( D_{profile} \) as the profiling phase, which has a duration of \( \omega \) days. After \( D_{profile} \) has been learned, MNEMOSYNE begins creating new domain versions on the date that a new domain was observed. Additionally, when multiple domains appear in the same day, MNEMOSYNE will aggregate these domains into the same domain-version.

**Version Prioritization**

To make MNEMOSYNE more efficient in locating the window of compromise, we prioritize the domain versions in the order of their likelihood to be the version that truly represents the
Table 3.3: The set of TTPs used in Mnemosyne’s weighting system.

<table>
<thead>
<tr>
<th>TTP ID</th>
<th>Severity</th>
<th>Score</th>
<th>Description &amp; Pattern</th>
</tr>
</thead>
</table>
| T1204.001 | Medium | 6     | **User Execution**: User navigates to an unknown domain. 
(p:Page) → Navigated → (t:Page)  
where p.securityOrigin = domain  
and t.securityOrigin \( \notin D_{profile} \) |
(p:Page) → File-Download → (t:File)  
where p.securityOrigin = domain  
and t.remoteOrigin \( \in D_v \) |
| T1189 | Medium | 6     | **Initial Access**: Unknown iframe Injection  
(s:Script) → Inserted → (i:Iframe) → Attached → (p:Page)  
where (s.sourceOrigin \( \in D_v \) or i.securityOrigin \( \in D_v \))  
and p.securityOrigin = domain |

window-of-compromise. This prioritization analyzes each domain version independently to identify suspicious behavior causally dependent on this domain version. We quantify the suspiciousness of these behaviors using a weighting system. Based on the behaviors found, an overall *suspiciousness* score is defined for the domain version. The domain versions are then placed in a priority queue based on their suspiciousness score. This prioritization focuses the analysis on the most suspicious versions, increasing the investigator’s efficiency.

**Weighting System**

MNEOSYNE’s weighting system is TTP-based, analogous to existing state-of-the-art, whole-system auditing approaches (e.g., Holmes [21] and Rapsheet [26]) in the sense that it relies on matching browser-based audit logs to existing attack patterns in the MITRE ATT&CK Framework [68]. The set of TTPs MNEOSYNE relies on to detect suspicious domain versions and the patterns required to match these TTPs to the browser-level audit logs is defined in Table 3.3. For each domain version \( D_v \), MNEOSYNE calculates a suspiciousness score. First, MNEOSYNE conducts a reachability analysis, starting from the set of domains in \( D_v \), to identify the set of pages impacted by this domain version,
which we call $P_{affected}$. For each page in $P_{affected}$, we search for audit events that match a TTP pattern. The score of the domain version is the sum of the severity scores of the matched TTPs defined in the Score column of Table Table 3.3.

### 3.3.6 User-Level Analysis

The final stage in Mnemosyne’s analysis is the user-level analysis. The purpose of the user-level analysis is to identify how a domain-version behaved differently based on the user that was visiting the compromised website, with the ultimate goal of minimizing the effort required by the forensic analyst to determine which users were unaffected, targeted, or victims of the attack. For each domain version, $D_v$, pulled from the priority queue, and the set of pages, $P_{affected}$, associated with it, the user-level analysis clusters pages in $P_{affected}$ that had similar behaviors while they were loaded into the browser. This clustering minimizes the number of pages the FA needs to analyze by aggregating pages with similar behaviors.
behaviors. The FA can then analyze clusters as one and assess and make decisions about entire clusters of pages instead of only a single page, reducing the amount of effort and time required to complete the investigation.

The clustering approach has two stages. The first stage extracts a features set from each page in $P_{affected}$. In the second stage, a differential analysis is completed over the feature sets extracted. The differential analysis phase generates clusters of pages. We call these clusters user-level versions; the term cluster and user-level version are used interchangeably.

**Definition 2** (User-Level Version). A User-Level Version := \{ $p_i \in P_{affected}$, metadata($p_i$) : all $p_i$ share the same feature set\}.

For each user-level version, a set of metadata properties, defined in Table 3.4, is tracked. We will provide additional information about each property during the remainder of this section. The final step of the differential analysis is to insert the user level versions into a versioning tree, where nodes are user level versions and edges represent dependencies between the versions. The benefit of this version tree is that it orders the versions and allows the forensic analyst to quickly determine what modifications were made to create the new version and the ancestry of each version.

**Feature Extraction Phase**

For each page $p_i$ in $P_{affected}$, we extract a set of features. Specifically, MNEMOSYNE collects all paths from the domains in $D_v$ to the page $p_i$ by querying the audit graph. The result of this query is the set of all paths, $PATHS_{d \rightarrow p_i}$, where $d \in D_v$. For example, in Figure 3.3.a we see the results of this query on the domain version, $D_v = \{ jupyter.elfinwood.top, jsdeliver.net \}$. It returns four paths (cyan, purple, green, and blue). The set of paths returned from the causality query are combined to create a subgraph, $S_{p_i}$. This process is repeated for all pages in $P_{affected}$. Figure 3.3 shows the subgraphs for three example pages. (a) represents a page where the user was unaffected by the attack,
Algorithm 1: Differential Analysis Algorithm

Input: UserVersions The set of initial user-level versions.
Result: VersionGraph: The resulting Versioning Graph.

begin
VersionGraph ← list();
while |UserVersions| > 0 do
    // Initialize current based on pageCount.
    current = max(UserVersions.pageCount)
    // Insert current version into VersionGraph.
    VersionGraph.insert(current);
    foreach u v in UserVersions do
        // Determine if current is a parent of u v.
        isParentVersion = u v.deltaSet ∩ current.deltaSet
        if |isParentVersion| > 0 then
            // Add current as a parent to version u v.
            u v.parents.append(current.versionId);
            // Complete diff operation on version u v.
            u v.deltaSet = u v.deltaSet - current.deltaSet;
            foreach m v in UserVersions do
                if m v == u v ∨ m v == current then continue;
                else if u v.deltaSet == m v.deltaSet then
                    // Complete Merge on u v and m v.
                    u v.pageSet.append(m v.pageSet);
                    u v.userSet.append(m v.userSet);
                    UserVersions.remove(m v);
                end
            end
        end
    end
    // Remove current version from UserVersions.
    UserVersions.remove(current);
end
return VersionGraph
end
(b) represents a page where the user was targeted, which is highlighted in orange, and (c) represents a page where the user was a victim of the attack, which is highlighted in pink.

The next step converts the subgraph, $S_p$, to a feature set, $\hat{S}_p$. For each node in $S_p$, we extract its identifier. The identifier for each node is the bolded attribute in Table 3.1a. For the edges, we create a three tuple containing the relationship type, and the identifier of the start and end nodes. The feature set consists of the nodes’ identifiers and the relationship tuples for the page $p_i$. This process is completed for all the related subgraphs. After creating $\hat{S}_{pi}$ for all pages, the next task is to assign each page to its initial user-level version. A page is assigned to its initial user-level version based on its feature sets. Specifically, pages are assigned to a user-level version, $U_v$, if $\hat{S}_{pi} == U_v.\Delta$. If no match is found, a new user-level version will be created, this page will be assigned to it, and $U_v.\Delta$ will be set to $\hat{S}_{pi}$.

**Differential Analysis**

The final phase in Mnomoseyn’s analysis is differential analysis. Given the initial user-level version set UserVersions, the differential analysis creates a version graph, where each node represents a unique user-level version and the edges represent ancestral relationships between the versions in the graph. For our analysis, an ancestral relationship implies the resources in the parent’s $\Delta$-Set were also observed by pages in the child’s pageSet. The advantage of presenting the versions embedded into a version graph is that it allows the FA to assess the modifications and differences made between the child and parent versions, and quickly determine which users observed which behaviors.

The analysis is initiated by selecting the root version from the set UserVersions, shown on line 5 of Algorithm algorithm 1. The selection of the root version is based on size of the user level version’s pageSet. The version with the largest page set is selected to be the current version. The current version will then be inserted into the VersionGraph. Next, the algorithm iterates over the remaining user-level versions. For each user-level version, $u_v \in$ UserVersions, the algorithm will determine if current
is a parent version of $u_v$. $current$ is considered a parent when the intersection of $u_v.\Delta$-$Set$ and $current.\Delta$-$Set$ is non-empty, as shown on line 10. When a parent version is found, the differential analysis updates the user version’s $\Delta$-$Set$. Specifically, a $\text{diff}$ operation is performed on the user version’s delta set, where $u_v.\Delta := u_v.\Delta - current.\Delta$. This operation prevents duplicating the same behaviors, which would increase the FA’s workload and prolong the analysis. After updating the $\Delta$-set of $u_v$, we compare $u_v$’s $\Delta$-set to the remaining user versions’ $\Delta$-set. If they are equivalent, we run a $\text{merge}$ operation and merge the two user-level versions. This step maximizes the cluster sizes, minimizing the feature sets that the FA has to manually analyze. We repeat this process, assigning the version with the largest, remaining page set to $current$, until every user-level version has been inserted into $\text{VersionGraph}$. The output of this differential analysis is the $\text{VersionGraph}$ for the user-level versions, which the forensic analyst can use to quickly assess the different behaviors exhibited by the domain version they are evaluating.

### 3.4 Evaluation

Our evaluation addresses the following research questions:

- How effective is MNEMOSYNE at reducing the analysis scope of the forensic investigation?
- How does the benign evolution of websites affect MNEMOSYNE analysis?
- What is the runtime performance overhead of MNEMOSYNE’s auditing daemon and analysis?
- What are the data storage requirements for MNEMOSYNE?

#### 3.4.1 Data Collection

The highly-targeted nature of watering hole attacks makes them extremely difficult to detect in the wild, which results in difficulty of collecting data on them. To overcome this, we de-
Table 3.5: Description of each attack scenario and the corresponding case in the wild.

<table>
<thead>
<tr>
<th>Attack Scenario</th>
<th>Website &amp; Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious OAuth Access</td>
<td><a href="http://www.cfr.org">www.cfr.org</a> — The adversary injected a malicious script into the homepage. If users were targeted, the original content of the page was blurred, and a malicious overlay was injected into the DOM. If the client interacted with the overlay, it redirected them to an attacker-controlled website hosting a malicious OAuth app that requested sensitive email permissions.</td>
<td>[5]</td>
</tr>
<tr>
<td>Clickjacking</td>
<td><a href="http://www.acumen.org">www.acumen.org</a> — The attack embedded a malicious iframe onto the page, which redirected users to a malicious website, hosting a malicious OAuth app. The app requested sensitive Gmail permissions.</td>
<td>[69]</td>
</tr>
<tr>
<td>Malicious Software Update</td>
<td><a href="http://www.energy.gov">www.energy.gov</a> — The adversary injected a malicious flash update onto the webpage. Victims of the attack were tricked into downloading a trojanized version of Adobe Flash.</td>
<td>[70]</td>
</tr>
<tr>
<td>Credential Harvesting</td>
<td><a href="http://www.cipe.org">www.cipe.org</a> — The adversary manipulated the original webpage’s DOM to mimic a Google login page. Victims of the attack would be tricked into leaking sensitive credentials.</td>
<td>[62]</td>
</tr>
<tr>
<td>Keylogging</td>
<td><a href="http://www.xero.com">www.xero.com</a> — The adversary injected a keylogger into the webpage, which logged all keystrokes by targeted clients.</td>
<td>[71]</td>
</tr>
<tr>
<td>Tabnabbing</td>
<td><a href="http://www.thebanker.com">www.thebanker.com</a> — The adversary used a tabnabbing attack to distract the user. Next, the attackers injected an iframe, which mimicked the institution’s login page. The attack victims leaked their sensitive email credentials.</td>
<td>[72]</td>
</tr>
<tr>
<td>Driveby</td>
<td><a href="http://www.zingnews.vn">www.zingnews.vn</a> — The adversary injected a malicious script into the homepage. If the users matched the client profile, the malicious script injected a 1x1 iframe into the DOM, which navigated to a malicious website that exploited CVE-2020-6405 to complete a drive-by download attack.</td>
<td>[73]</td>
</tr>
</tbody>
</table>

Developed a scalable testbed that is capable of simulating sophisticated watering hole attacks on a large organization. This testbed has the capability to make arbitrary modifications to an otherwise benign website and simulate visits to compromised websites. To simulate a compromised website, we relied on Chromium’s DevTool’s `Fetch` namespace, which can intercept and modify network requests made by the browser. Our testbed supports directly modifying HTML pages and scripts on-the-fly to support various modification techniques. This allows the testbed to simulate malicious modifications being made to the website.

To simulate visits, we developed a driver based on puppeteer [74]. During a visit, the driver navigates to up to 15 webpages on the site. However, only visiting webpages limits execution coverage because modern webpages are highly-dynamic and event-driven. To address this, our driver simulates JS-based events while visiting the page. Also, the driver automatically emulates different browser/OS combinations, including mobile operating systems (e.g., screen size and other system properties are adjusted according to the
Table 3.6: Graph statistics for each attack scenario.

<table>
<thead>
<tr>
<th>Attack Scenario</th>
<th>Nodes/Edges</th>
<th>Visit Sessions</th>
<th>Pages Visited</th>
<th>Distinct URLs</th>
<th>Script Instances</th>
<th>Network Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious OAuth Access</td>
<td>9.20M / 16.1M</td>
<td>5.37K</td>
<td>57.8K</td>
<td>4.96K</td>
<td>8.80M</td>
<td>3.10M</td>
</tr>
<tr>
<td>Clickjacking</td>
<td>4.40M / 5.90M</td>
<td>950</td>
<td>7.66K</td>
<td>267</td>
<td>3.99M</td>
<td>658K</td>
</tr>
<tr>
<td>Malicious Software Update</td>
<td>628K / 1.30M</td>
<td>1.00K</td>
<td>8.69K</td>
<td>640</td>
<td>547K</td>
<td>374K</td>
</tr>
<tr>
<td>Credential Harvesting</td>
<td>770K / 1.60M</td>
<td>927</td>
<td>8.40K</td>
<td>364</td>
<td>710K</td>
<td>449K</td>
</tr>
<tr>
<td>Keylogging</td>
<td>20.9M / 33.8M</td>
<td>1.91K</td>
<td>100K</td>
<td>5.33K</td>
<td>18.8M</td>
<td>6.20M</td>
</tr>
<tr>
<td>Tabnabbing</td>
<td>6.70M / 15.0M</td>
<td>2.18K</td>
<td>83.2K</td>
<td>741</td>
<td>6.20M</td>
<td>4.30M</td>
</tr>
<tr>
<td>Driveby</td>
<td>952K / 3.10M</td>
<td>580</td>
<td>6.72K</td>
<td>1.29K</td>
<td>742K</td>
<td>926K</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>6.20M / 11.0M</strong></td>
<td><strong>1.84K</strong></td>
<td><strong>39.0K</strong></td>
<td><strong>1.94K</strong></td>
<td><strong>5.7M</strong></td>
<td><strong>2.31M</strong></td>
</tr>
</tbody>
</table>

emulated system). Finally, we designed our testbed to be scalable by making each crawler a container-based application, which allowed us to execute multiple crawlers in parallel. Each container contains a headless Chromium browser and the driver that simulates a visit to the compromised domain.

3.4.2 Datasets

Leveraging the watering hole testbed discussed in subsection 3.4.1, we collected datasets to develop 7 attack scenarios discussed in subsubsection 3.4.2 and two benign datasets discussed in subsubsection 3.4.2.

**Attack Scenarios**

To extensively evaluate MNEMOSYNE, we developed 7 attack scenarios inspired by watering hole attacks that have been reported in the wild. A detailed description of the attack scenarios is provided in Table 3.5, including the website that we simulated being compromised and a reference to the real-world attack that inspired this scenario. For each attack scenario, we collected data for at least two weeks, and each scenario had three phases. During the first phase, the website was benign, and no malicious modifications were made. The purpose of this phase was to collect the auditing information necessary to model the benign behavior of the website. The second phase simulated a reconnaissance phase, where the
website was compromised. At this point, the simulated attacks had not targeted any users. The final stage was the targeting phase, where the attack actively sent the malicious payload to victims. We provide statistics related to the size of the attack graph for each scenario and important crawling statistics in Table 3.6. On average, we simulated 1,844 visits during the attack scenarios. Also, we found that, on average, 1,941 distinct URLs related to the compromised domain were visited. Finally, the average graph size for each attack scenario was 6.2M and 11.0M nodes and edges respectively.

**Benign Datasets**

The benign datasets contain simulated visits to a large set of benign websites. We collected two datasets, which we will call “Categories” and “Alexa”. The details of each dataset is described below.

**Categories**

The categories dataset was developed by crawling 900 websites from February 6th, 2020 to August 18, 2020. Each website crawled had an associated category tag, where the category tag represents the website type (e.g., News, Sports, etc.). To categorize the websites, we leveraged DMOZ; the most comprehensive, human-edited directory of the Web [75]. The 900 websites were randomly selected. After removing websites that returned an error, we had a dataset of 830 valid websites that included 278,177 unique pages related to the websites.

**Alexa**

The Alexa dataset was developed by crawling the Alexa 1k from July 13th, 2020 to August 18, 2020. In total, we collected 120,245 unique pages related to these websites.
Data and Evaluation Limitations

There are some potential limitations to pay attention to when relying on simulated attack scenarios to complete an evaluation. First, when visiting each site in the attack scenario, the navigation through different pages on this website was randomized. In practice, website visitors will typically follow specific and routine visiting patterns to complete specific tasks. However, MNEMOSYNE’s analysis does not rely on the visiting pattern of the users, so this is not expected to represent a significant issue in practice. Next, our testbed does not support automatically logging into a webpage. Since portions of a website may require authentication to view, some portions of a website may be unavailable to our testbed. While this does limit the visibility of the website in our experiments, it will not be a significant issue in practice. This is because, in a real-world deployment, MNEMOSYNE would have visibility to these portions of the website once the user logged into the site, since MNEMOSYNE will record audit logs as the user interacts with webpages through the browser. Finally, one limitation of our testbed is content tailored to a specific user for benign use-cases. While our testbed can simulate different users, this simulation mainly alternates the User-Agent string when visiting the website. Unfortunately, for websites that distribute content based on profiling the user or requiring the user to log in, our current implementation of emulating different users will most likely lead to the websites not serving "user-specific" content in a meaningful way. However, we believe benign use-cases of user-specific content will not have a large affect on MNEMOSYNE’s analysis because, while websites routinely serve user-specific content, this content will be served off the same set of domains. Since MNEMOSYNE would identify these domains during its profiling phase, these user-specific modifications would be filtered out of the analysis scope.
Table 3.7: A detailed performance comparison between manual++ and MNEMOSYNE in terms of number of domains and scripts a forensic analyst needs to investigate. Scripts† are all scripts related to the corresponding domain. Scripts‡ are the set of scripts that have behaviors attributed to them (e.g., network requests, DOM insertions, etc.). MNEMOSYNE can reduce the analysis scope significantly, for example, (-99.56%) means reduction based on the raw data.

<table>
<thead>
<tr>
<th>Attack Scenario</th>
<th>Raw # of Domains</th>
<th>Raw # of Scripts</th>
<th>Raw # of Domains</th>
<th>Raw # of Scripts†</th>
<th>Raw # of Domains</th>
<th>Raw # of Scripts‡</th>
<th>Raw # of Versions</th>
<th>manual++ # of Domains</th>
<th>manual++ # of Scripts</th>
<th>manual++ # of Scripts†</th>
<th>manual++ # of Scripts‡</th>
<th>MNEMOSYNE # of Domains</th>
<th>MNEMOSYNE # of Scripts</th>
<th>MNEMOSYNE # of Scripts†</th>
<th>MNEMOSYNE # of Scripts‡</th>
<th>MNEMOSYNE # of Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious OAuth Access</td>
<td>59</td>
<td>6,243</td>
<td>45 (-23.73%)</td>
<td>5,814 (-6.87%)</td>
<td>477 (-92.36%)</td>
<td>4 (-93.22%)</td>
<td>6 (-99.90%)</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clickjacking</td>
<td>18</td>
<td>523</td>
<td>12 (-33.33%)</td>
<td>507 (-3.06%)</td>
<td>116 (-77.82%)</td>
<td>2 (-88.89%)</td>
<td>5 (-99.04%)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Malicious Software Update</td>
<td>23</td>
<td>1,102</td>
<td>20 (-13.04%)</td>
<td>1,086 (-1.45%)</td>
<td>318 (-71.14%)</td>
<td>1 (-95.65%)</td>
<td>63 (-94.28%)</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Credential Harvesting</td>
<td>20</td>
<td>534</td>
<td>15 (-25.00%)</td>
<td>521 (-2.43%)</td>
<td>112 (-79.03%)</td>
<td>1 (+95.00%)</td>
<td>85 (-84.08%)</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keylogging</td>
<td>11</td>
<td>1,761</td>
<td>11 (0.00%)</td>
<td>1,761 (0.00%)</td>
<td>37 (97.90%)</td>
<td>2 (+81.82%)</td>
<td>6 (-99.66%)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tabnabbing</td>
<td>64</td>
<td>27,477</td>
<td>48 (-25.00%)</td>
<td>27,259 (-0.79%)</td>
<td>26,262 (-4.42%)</td>
<td>2 (+96.88%)</td>
<td>13 (-99.95%)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Driveby</td>
<td>571</td>
<td>3,251</td>
<td>567 (-0.70%)</td>
<td>3,055 (-5.42%)</td>
<td>350 (-89.16%)</td>
<td>2 (+99.65%)</td>
<td>2 (-99.94%)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>109</td>
<td>5,838</td>
<td>103 (-6.27%)</td>
<td>5,715 (-2.21%)</td>
<td>4,554 (-22.01%)</td>
<td>2 (-98.17%)</td>
<td>26 (-99.56%)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.4.3 Attack Scenario Investigation

**Forensic Analysis Scope Reduction**

To measure the efficiency gains that Mnemosyne provides, we completed an empirical evaluation to quantify how much of the analysis space is reduced when using MNEMOSYNE to complete the investigation.

**Defining the Analysis Space**

We define the analysis space as the set of domains and scripts related to each attack scenario in Table 3.5. We want to point out that the number of scripts reported for each attack scenario is the number of unique script URLs, not to be confused with the number of script instances. The choice to focus on domains and scripts is based on a preliminary study we conducted with 5 security-trained professionals. The purpose of this study was to assess how different professionals approach forensic investigation tasks. To this end, we assigned an investigation task to each participant, provided them access to the browser logs, and asked them to determine the window-of-compromise and the attack’s victims. Each participant was provided with access to a graph database that contained the attack scenario logs and a graphical interface\(^1\) for interacting and making queries to the database.

\(^1\)https://neo4j.com/developer/neo4j-browser
to enable the investigation. After each participant completed the task, we conducted an exit interview to discuss what strategies the participants adopted to perform the investigation. We found that most participants used a two-phased approach. First, they filtered out well-known domains. Then, for the remaining domains, they analyzed the scripts served by those domains. Because of this approach, we consider the number of domains and scripts involved in the attack scenario to play a larger role in the analysis time compared to other types of resources (e.g., images, CSS files, etc.).

**Developing a Baseline**

Following the practical strategies observed during this study, we developed a baseline system to compare against MNEMOSYNE, which we call manual++. manual++ attempts to generalize the approaches used by the different participants to perform a forensic investigation based on browser logs. Specifically, manual++ first collects all domains that communicated with the compromised website. Next, it filters out any of these domains that are listed on the Alexa 10k, since they are highly likely to be benign. Next, it further reduces the number of domains by filtering out domains that only served static content (e.g., images, fonts, CSS files, etc.).

**Measuring Analysis Reduction**

To measure the analysis reduction MNEMOSYNE provides, we compare it to manual++. We focus on the number of domains and scripts that would require manual inspection when using MNEMOSYNE, compared to manual++. An extensive reporting of the results is provided in Table 3.7. The number of domains and unique script URLs found in each attack scenario is reported in the raw column. We see that MNEMOSYNE was able to filter out, on average, 98.17% of the domains while manual++ was only capable of filtering out 6.27% of the domains. It's even less for the case of Driveby because the website employs ads that generate random domain names. Next, we inspected the number of scripts filtered
out of the analysis space by MNEMOSYNE and manual++. Our experiments show that
MNEMOSYNE was able to filter out 99.56% of the scripts from the analysis space, which
shows a significant reduction in the number of scripts required for manual analysis by the
investigator. To measure the number of scripts filtered by manual++, we provide two
results in columns Scripts† and Scripts‡. Scripts† provides the number of scripts remaining
after applying manual++’s filtering. We see that on average, only 2.21% of scripts
were filtered by manual++. Also, we provide a second set of results in column Scripts‡.
The results presented in column Scripts‡ were calculated by adding an additional filtering
stage, which was more aggressive and filtered out scripts that did not have behaviors
causally attributed to them (e.g., they made no network requests or DOM insertions). The
results show that applying this additional filtering stage can reduce the number of scripts
by 22.01% on average, and in some cases, such as the Keylogging scenario, this additional
filtering stage performs well. However, we also see that in the Tabnabbing attack scenario,
it performs extremely poorly, and was only able to filter out 4.42% of the scripts in the
analysis space, while MNEMOSYNE was able to filter out 99.95% of scripts. We also found
that the naive approach used by manual++ to filter out domains based on the Alexa 10K
led to a false-negative in the Malicious Software Update attack scenario, because the adver-
saries served the malware from a Git repository on hxxps://www.github.com. On the other
hand, MNEMOSYNE correctly identifies this attack component. These results show that
MNEMOSYNE can significantly reduce the scope of the analysis space that requires manual
analysis for the forensic investigation.

**Attack Scenario Domain Versions**

Next, we investigated the number of domain versions generated for each attack scenario.
For each scenario, we set $\omega = 1$ (i.e., we used one day for the profiling phase). We found
that a low number of versions were generated for each attack scenario, as reported in the
last column of Table 3.7. One outlier was the Malicious OAuth Access Scenario,
with 4 versions after removing the core domain-version. We further investigated and found that there were 3 new benign versions generated. These versions were generated shortly after the profiling phase, which means this phase was too short for this website. This is reasonable since this scenario compromised `hxxp://www.cfr.org`, which was a larger website and had 4,957 distinct URLs visited during the scenario. Also, the version prioritization prioritized all 3 of these benign versions lower than the malicious version.

**Version Prioritization**

In all attack scenarios, our system accurately prioritized the compromised version over the benign versions, with the one exception of the Keylogging Attack Scenario. This shows MNEMOSYNE’s version prioritization approach was effective, and we investigated each attack scenario to determine exactly why the prioritization was effective. The Malicious OAuth Access attack lured the user into navigating to `jupyter.elfinwood.top`\(^2\), which was captured as a cross-origin navigation in the causality graph. As this cross-origin navigation was attributed to a script served by `jupyter.elfinwood.top`, it flagged TTP `T1204.001`, which increment its suspiciousness score. Finally, for the Clickjacking and Tabnabbing scenarios, MNEMOSYNE identified TTPs `T1189 & T1204.001` being causally dependent on the attack scenarios’ respective malicious domain version. For the Malicious Software Update attack, it successfully detected TTP `T1204.002`. Finally, for the Driveby scenario, we detected TTP `T1189` when the iframe was injected into the DOM. MNEMOSYNE was not effective in prioritizing the Keylogging attack scenario since the attack did not insert an iframe nor trick users to do anything, but rather sent network requests in the background. We believe this is acceptable, as version prioritization does not aim to determine the compromised version directly but aims to prioritizes domain-versions relying on identifying typical suspicious events that are related to social-engineering attacks.

\(^2\) `jupyter.elfinwood.top` was the malicious domain used throughout the attack scenarios.
Table 3.8: A report of the user-level version types generated during each attack scenarios.

<table>
<thead>
<tr>
<th>Attack Scenarios</th>
<th>User-level Versions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malicious OAuth Access</td>
<td>✓</td>
</tr>
<tr>
<td>Clickjacking</td>
<td>✓</td>
</tr>
<tr>
<td>Malicious Software Update</td>
<td>✓</td>
</tr>
<tr>
<td>Credential Harvesting</td>
<td>✓</td>
</tr>
<tr>
<td>Keylogging</td>
<td>✓</td>
</tr>
<tr>
<td>Tabnabbing</td>
<td>✓</td>
</tr>
<tr>
<td>Driveby</td>
<td>✓</td>
</tr>
</tbody>
</table>

**User-Level Versions**

To measure the effectiveness of the User Level Analysis, we determine how effective it is at developing “uniform” clusters. We define a cluster as uniform if the pages mapped to it only represent unaffected users, targeted users, or victims. We define six different types of user-level versions; unaffected users (Unaff), targeted (Tar), and victim (Vic) versions only include a single user-type. Next, the groups unaffected-targeted (Unaff-Tar), targeted-victim (Tar-Vic), and unaffected-victim (Unaff-Vic) are mixed user-level versions. Mixed groups are generated by the user-level analysis when there is not enough context in the underlying audit logs to accurately distinguish between the two user types. The results are shown in Table 3.8, where each column represents a different category of user-level versions. We see that in 6/7 attack scenarios, MNEMOSYNE generates ideal “uniform” clusters. For the Malicious Software Update attack scenario, MNEMOSYNE created two user-level version types, a victim user-level version and a mixed unaffected-targeted version. To understand this, we analyzed the attack scenario in more depth and found that the attack relies on inserting an **overlay** tag into the page to lure the user into installing the malicious file. Since MNEMOSYNE relies on an instrumentation-free approach for auditing the browser, it has less visibility in terms of DOM modifications compared to prior work [35] and unfortunately could not attribute the insertion of the **overlay** to a spe-
Table 3.9: The 15 websites used for the false positive analysis of the version prioritization of MNEMOSYNE.

<table>
<thead>
<tr>
<th>White-list of Websites Used in False Positive Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>cloudflare.com, cloudfront.net, doubleclick.net, facebook.net, google.com, googleadservices.com, googleapis.com, googlesyndication.com, googletagmanager.com, googletagservices.com, hotjar.com, microsoft.com, outbrain.com, twimg.com, twitter.com</td>
</tr>
</tbody>
</table>

Specific script. However, since file download events can be detected, MNEMOSYNE is able to narrow down the analysis space to identify all the users that were victims of the attack.

3.4.4 Benign Version Analysis

Next, we completed a study to evaluate the number of versions reconstructed over an extended time period. To achieve this, we evaluated MNEMOSYNE’s domain-version reconstruction using the benign datasets with $\omega = 1$. The average number of versions reconstructed per category is shown as the solid triangle in Figure 3.4. The light yellow box shows the five-point-summary for the Alexa and Categories dataset. We found that news websites have the highest average with 4.33 versions generated during the crawling period, while gaming sites only had 1.52 versions. The average number of versions generated in one month is 2.15 for the Alexa Top 1k. An extreme outlier, hxxp://auctiongr.com, in the shopping category generated 22 versions. This website was a shopping site back in February 2020 and only generated one version till March. However, the domain was registered to a porn and malvertising site in June 2020, and started adding malicious domains frequently. There are 7 websites considered outliers, for the Alexa category, which have more than 6 versions generated in one month.

This experimental evaluation shows that MNEMOSYNE’s domain-versioning system can be effective for long periods of time.
False-Positive Analysis

Benign updates made after the profiling phase will generate a new domain version. We completed an experimental evaluation to assess how often benign updates would be flagged as suspicious. To complete this experiment, we used the benign versions discussed in sub-section 3.4.4. We inspected 3,663 benign versions generated across 1,830 websites. We found that 14.12% of the benign versions were flagged as suspicious. We further investigated the flagged benign versions and found that 2.38% of the versions were flagged due to cross-origin navigation to an unknown domain, while 11.74% of the flagged domains were related to anomalous iframes being injected into the DOM. However, in almost all cases the iframes injected were ad-related. We found that by checking the iframe’s src property, against a white list of 15 domains (listed in Table 3.9), it allowed us to filter out 9.15% of the iframes related to ads. Finally, after applying the white list filtering approach we find that only 4.97% of the benign versions were flagged as suspicious. Since the analyst will only need to investigate benign versions related to the compromised version, this shows that benign updates will not significantly increase the time of the investigation.

3.4.5 Runtime Performance

To evaluate the runtime performance of the auditing daemon, we measured the page load time for the top 1,000 most popular websites according to Alexa.com, using out-of-the-box Chromium version 80.0.3987.163. The page load metric is important because previous studies have shown that a slow page load time can lead to frustrated users and drive websites’ revenue lower [76]. For each site, we conducted 10 trials, both with and without our auditor attached. Prior to measuring the page load time, we loaded the homepage of each site into the browser so that it would heat up the browser’s cache. The purpose of this was to minimize the influence that potential network latency variations would have on the experiment. It is important to notice that MNEMOSYNE logs web requests regardless if the object was fetched from the cache or the actual server, so the time spent within MNEMOSYNE’s
Figure 3.4: Five-point-summary of domain versions generated for each website for the Categories and Alexa datasets, with whiskers being 1.5x IQR.
Figure 3.5: The runtime performance overhead induced on the page load by MNEMOSYNE for the Alexa 1k. (a) presents the runtime overhead increase for the page load. (b) provides the absolute time induced by MNEMOSYNE. Whiskers are set to (0%, 95%).
logging functions will be the same whether an object is retrieved from the cache or directly from the network.

The experimental results are presented as a five-point-summary in Figure 4.3, with (a) presenting the overhead percentages compared to loading a page while MNEMOSYNE is off, and (b) presenting the absolute time induced by MNEMOSYNE on the page load. We found that MNEMOSYNE’s auditor daemon had a low performance overhead of only 2.93% on average and a 95th-percentile overhead of 9.80%, which is similar to the overhead introduced by previous work [35]. Additionally, (b) shows that on average MNEMOSYNE increases the load time by only 0.04s. However, we found two outliers, hxxps://www.tripadvisor.com and hxxps://www.atlassian.com, which had page load overheads that were slightly over 25%. We spent a significant amount of time assessing why these two cases were outliers. This included toggling the DevTools namespace to identify exactly which namespace(s) were causing the performance overhead. We found that the Network and Debugging DevTools namespaces appear to be contributing the most to the overhead induced. Unfortunately, a more fine-grained approach to identify exactly which other DevTools hooks were contributing to the overhead and by how much would require instrumenting Chromium itself. Since this outlier overhead was observed only on 2 out of 1,000 websites, we leave this detailed analysis to future work.

It is important to notice that MNEMOSYNE leverages only a small set of DevTools hooks within a small set of namespaces, namely Network, Page, Debugger, and Target. Therefore, runtime performance could be further optimized by developing a customized Forensics DevTools namespace, which would only activate the hooks that are necessary for the logging, while avoiding the overhead introduced by calls to other unused hooks that occur when other DevTools namespaces are present. In summary, our performance evaluation shows that MNEMOSYNE has a reasonable overhead, especially for a prototype, and could be deployed in real-world scenarios without significantly affecting the user’s browser experience.
Table 3.10: The 10 websites used for the Storage Overhead evaluation (subsection 3.4.6) of MNEMOSYNE.

<table>
<thead>
<tr>
<th>List of Websites Used in Storage Evaluation</th>
</tr>
</thead>
</table>

Next, we measured the performance of MNEMOSYNE’s automated log analysis process (see subsection 3.3.5 and subsection 3.3.6) on a standard laptop with Intel I7-8700B CPU running at 3.2 GHz and 32GB of physical memory. On average, the log analysis process takes less than 5 minutes for a graph of 6.2M nodes and 11.0M edges. Additionally, a breakdown of the runtime performance for every attack scenario is provided in ??.. This shows the runtime performance for analyzing each attack scenario is efficient.

3.4.6 Storage Overhead

To measure the disk space overhead, we ran MNEMOSYNE’s auditor for a 50-minute browsing session and visited 10 heavily dynamic and popular websites. The websites used are listed in Table 3.10. The compressed version of MNEMOSYNE’s audit logs for the entire browsing session was only 3.1 MB. This means that, on average, the disk space requirement for MNEMOSYNE is only .06 MB per minute for highly active browsing sessions. If we assume MNEMOSYNE is deployed in a typical enterprise environment, it would only require 28.8MB of storage for a single device in an 8-hour work day. If we assume a 262 workdays per year, less than 7.4GB of disk space is required to store MNEMOSYNE’s audit log per year. For an enterprise network of 1,000 devices, only 7.4 TB of disk space is required to store the entire dataset for a single work year. This experimental evaluation shows that MNEMOSYNE’s lightweight approach to collecting audit logs has significant improvements compared to JSGraph [35] and reduces the required storage by 82.4%.
3.4.7 Limitations

There are a few limitations that can occur with MNEMOSYNE. First, the current version of DevTools only supports attributing DOM modifications to scripts when the DOM node being inserted is an iframe or a script node. However, despite limited capability in attributing DOM modifications, MNEMOSYNE was able to perform exceptionally well during our experimental evaluations. This limitation was introduced because we chose not to introduce any code changes to the browser. Although, prior approaches have shown that fine-grained DOM modification attribution is feasible, it requires extensive modifications to the Blink-V8 bindings layer of the browser [35]. Since MNEMOSYNE was able to detect the attacks in each scenario without requiring these extensive modifications, we believe this was the correct design choice, as it clearly provides significant advantages for deployment in real-world enterprise environments. Finally, we believe that if an enterprise network prefers a more fine-grained auditing approach (e.g., using JSGraph [35]) the generated audit logs could still be leveraged by MNEMOSYNE’s analysis modules with limited engineering effort. Second, MNEMOSYNE relies on a domain-versioning technique to identify the window-of-compromise. One potential limitation that could occur with domain-versioning is that the adversary could orchestrate the entire attack campaign off the compromised website. To achieve this, the adversary would need to store all malicious scripts and payloads on the compromised site’s origin. If the adversary chose to use this approach, MNEMOSYNE’s domain-versioning would not be able to identify the modifications made by the adversary. However, we argue this is extremely unlikely. First, after reviewing a corpus of over 300 well-documented sophisticated attacks carried out by various APT groups, we found that all the watering hole attacks modified the page such that it communicates with a new domain, specifically, their C&C server [77]. A main reason for this is that it provides the attacker the flexibility to update and modify the code without having to make significant modifications to the compromised website. By minimizing the modifications made, it decreases the likelihood of their attack being detected on the compromised server via the
hosting organization’s firewall or data loss prevention software (DLP).

Finally, as previously discussed in subsection 3.3.2, MNEMOSYNE has limited visibility when investigating attacks that rely on a drive-by download. Specifically, MNEMOSYNE can only identify the ”setup” phase before the browser is exploited. However, despite only recording the setup phase of a drive-by download attack campaign MNEMOSYNE was still capable of identifying the victims of the driveby attack scenario in our evaluation (subsubsection 3.4.3), which demonstrates that MNEMOSYNE has the capability of improving the efficiency of the analysis, even when the adversary relies on a drive-by download.

3.5 Related work

Causality Analysis Systems

Developing systems that rely on capturing attack provenance to investigate sophisticated attacks has become a growing area of research [10, 11, 22, 15, 14, 42, 13, 12, 16, 23, 57, 24, 58, 34, 41, 36, 37, 35]. One shortcoming of whole-system provenance systems is the dependency explosion problem, which occurs when long-running processes communicate with many external entities. To address dependency explosion, several works have proposed partitioning the execution of a long-running process into units-of-execution [15, 14, 23, 24]. For example, BEEP [15] proposes to partition long-running processes into execution units based on the internal event loop found in applications. UIScope [23] takes a different approach and partitions the application’s execution based on GUI elements of an application. However, one limitation of all existing whole-system provenance systems is the semantic gap between system level semantics and browser-level semantics. To bridge this gap, JSGraph [35] develops a customized browser that tracks fine-grained information related to the provenance graph in terms of browser-level semantics. Unlike MNEMOSYNE, JSGraph requires extensive modifications to the browser itself, which makes real-world deployment difficult.
There has been a significant amount of work that uses attack provenance to improve the efficiency of identifying attacks in a postmortem fashion [21, 27, 29]. For example, Holmes [21] relies on the attack kill chain [4] to identify attacks. Priotracker [27] aims to improve the efficiency of postmortem analysis by automatically prioritizing abnormal causal dependencies for enterprise security. Additionally, Nodoze [25] relies on causality information to significantly reduce the number of false positives generated by industry alert systems like Splunk [46]. In addition to causality-based attack detection systems, there has also been a significant amount of work related to detecting malicious activity on the web [78, 80, 81, 82, 83, 84, 85, 86, 87, 79]. For example, Zozzle [86] detects JS-based malware by identifying syntax elements that are highly predictive of malware. Additionally, several systems have been developed that aim to detect compromised websites [80, 78, 81]. Most similar to MNEMOSYNE is Delta [78], which aims to identify changes associated with malicious and benign behaviors in a website. However, unlike MNEMOSYNE, Delta’s goal is to identify compromised webpages. MNEMOSYNE extends this work by identify the impacts a compromised website had on an organization. that routinely visited this website.

3.6 Conclusion

In this paper, we present MNEMOSYNE, a novel postmortem analysis engine for analyzing sophisticated watering hole attacks. We completed an extensive evaluation on several real-world watering hole attack scenarios and our results show that MNEMOSYNE is capable of efficiently identifying the victims of a watering hole attack at an enterprise environment.
CHAPTER 4
WEBRR: A FORENSIC SYSTEM FOR REPLAYING AND INVESTIGATING WEB-BASED ATTACKS IN THE MODERN WEB

4.1 Introduction

With the recent rise in enterprise data breaches, it is important that forensic investigations be carried out to fully understand how an adversary achieved each stage of the cyber-kill chain [4]. To address this growing need, researchers have developed system-level state-of-the-art auditing systems [16, 18, 19, 12, 10, 11, 15, 22, 14, 23, 24, 13, 31, 34, 29, 28, 21, 27, 30, 25, 33], including whole-system record-and-replay approaches [88, 10, 11, 42, 89]. Unfortunately, a major limitation of all these systems is that they provide extremely limited visibility into web-based attacks, because a large semantic gap exists between system-level abstractions (e.g., processes, sockets, files) and the necessary semantics required to investigate web-based attacks (e.g., HTML/CSS rendering and JavaScript execution).

As an initial attempt to address this gap, security-grade browsers with some auditing capabilities have been proposed [90, 91] and security researchers have begun studying web-specific auditing systems [37, 35, 92]. However, these systems (e.g., [35, 37]) are record-only systems and do not offer replay functionalities. In practice, having the capability to record and replay web-based attacks is highly desired during a forensic analysis, because it allows the investigator to interactively investigate (e.g., via dynamic analysis) the attack in a postmortem fashion. Additionally, record-only logs have limited capability of expressing the visual component of attacks, which is especially concerning in the context of the web, since many web-based attacks have a visual component (e.g., social engineering attacks).

While prior work has attempted to provide GUI information [35, 92, 23], these systems can only provide limited data such as textual information and GUI metadata. However,
for modern browsers, this is not enough information to express what was rendered on the
page, because styling or layout information cannot be provided in a meaningful way. This
limitation is alleviated by record and replay technology, since it allows the analyst to render
exactly what the user saw during the attack.

While a few web-oriented record and replay systems have been proposed, the majority
of prior work focused on supporting debugging and testing [49, 50, 51, 52, 53]. Unfor-
tunately, these debugging systems were not designed with security auditing in mind, and
consequently, they do not have the necessary properties to be deployed in a forensic setting.
For example, forensic-grade record and replay systems need recording to be always on, but
the popular debugging tool, Mozilla’s rr [49], can lead to a performance overhead of > 7x
in some cases [49]. Additionally, Jalangi [52] and Mugshot [50] rely on instrumenting the
JavaScript (JS) code of web applications to support record and replay. Unfortunately, in-
strumenting JS code is unsuitable in a forensic setting, because these systems can easily be
detected and in some cases evaded (e.g., the recording may be disabled by the malicious
web application).

To summarize, we argue that to support forensic-grade record and replay, the system
needs to satisfy the following fundamental properties:

- **Replay Determinism** – Deterministic replay is critical for reliable forensic analysis. If
  the replay diverges from the recorded browsing trace, this may prevent the attack from
  replaying correctly or at all. Therefore, the analyst may not be able to reconstruct the
  root causes of the attack, or even worse they may determine that an attack did not occur
  at all because it failed to replay.

- **Always-On Recording** – Because attacks are unpredictable and ephemeral, audit log
  recording must be always on.

- **Tamper Proof** – The auditing system must not be easily tampered with or disabled by
  an adversary.
• **Portable** – Web-based, forensic record and replay systems need to be OS- and device-independent.

The current state-of-the-art for in-browser record-and-replay systems is WebCapsule [36], which attempts to meet the needs outlined above by instrumenting Chromium’s rendering-engine to support record-and-replay of web-based attacks. Unfortunately, WebCapsule’s design does not allow for achieving deterministic replay, due to design limitations that prevent it from preserving the correct execution order of callbacks, events, and script execution during the replay. Due to its inability to preserve the correct order, it introduces *executional divergence* during the replay, which eventually causes the replay to become non-deterministic or crash during an investigation. Unfortunately, due to this limitation (which is also acknowledged in the original paper, [36], Section 5.4), WebCapsule is unable to replay a number of real-world attacks, as we demonstrate in our evaluation (subsection 4.5.3).

To fill the fundamental gaps left by previous work, we propose a novel system named WebRR, an *always-on* forensic auditing system that enables *deterministic* record and replay of modern web applications. WebRR re-imagines how to replay a website by leveraging the concept of *JavaScript Execution Unit Partitioning* to partition a web app’s JS into a sequence of JavaScript execution units (JEUs), where a JEU represents a script, callback, or event handler(s). Next, during the replay, WebRR replays the application by replaying each individual JEU while also preserving the recorded execution order. This allows WebRR to be resilient to executional divergence that would otherwise cause the replay to fail. However, there were still several challenges that had to be addressed in order to properly leverage this novel technique, including having to synchronize the DOM prior to each JEU execution and ensuring that all sources of non-determinism are recorded. Nevertheless, WebRR is capable of handling all of these issues. Finally, unlike previous work, we designed WebRR to simultaneously record both a web application and its Service Worker (SW). This is important because Service Worker abuse enables an increasingly large vari-
ety of sophisticated web-based attacks ([93, 94, 95, 96]). To the best of our knowledge, we are the first web-based record-and-replay system to demonstrate this capability.

In our evaluation, we show that WEBRR achieves deterministic replay and demonstrate the replay of a number of real-world attacks (subsection 4.5.2), while also showing that prior approaches fail to correctly replay these attacks (subsection 4.5.3). Additionally, we demonstrate that WEBRR can replay Service Workers execution, allowing us to properly replay modern web applications.

In summary, we make the following major contributions:

• **Forensic-grade record and replay of web-based attacks.** We implement WEBRR, a forensic-grade, record and replay system that allows forensic analysts to replay fine-grained, web-based attacks in a post-mortem fashion. We intend to open source WEBRR and the datasets in the evaluation.

• **Deterministic replay.** We design WEBRR to enable deterministic replay of modern web applications (including those that use SW) while avoiding the enormous complexity of replaying the entire browser and at the same time filling the semantic gap between system- and web-level abstractions.

• **Comprehensive evaluation.** Through our extensive evaluation, we demonstrate that WEBRR is capable of replaying many different types of sophisticated, web-based attacks that prior works fail to correctly replay. We also shows that WEBRR is capable of replaying popular benign websites found on the Tranco 1k list [97]. Furthermore, we demonstrate that WEBRR’s recording runtime performance and storage overhead are sufficiently low to make its deployment in read-world enterprise environments practical. For instance, our experiments show that WEBRR’s average runtime overhead is under 3.44% when visiting popular websites. Finally, we demonstrate WEBRR’s portability by evaluating it on a diverse set of devices and operating systems, including Linux, Android, and Windows.
4.2 Background

As WEBRR is built on the Chromium browser, we begin with a brief overview of Chromium’s architecture. Chromium-based browsers use a multi-process architecture. The main process is the browser process, which is analogous to an OS-level kernel. Web content is then rendered in sandboxed render processes that do not have direct access to the network or underlying filesystem. As WEBRR is mainly implemented within the render process, we provide more detail about the components of this process.

4.2.1 Render Process Architecture

Most of WEBRR is implemented via render process instrumentations, which allow us to extend Chrome’s DevTools [98, 99] with a custom forensic Inspector Agent [100]. To facilitate the discussion of WEBRR’s design, we provide a high-level overview of the Render Process architecture.

**Rendering Engine Major Components.** Each render process includes four major components: (i) the rendering engine, named Blink; (ii) the JS engine, named V8; (iii) the bindings layer that connects V8 to Blink, and (iv) the platform layer. The rendering engine is a highly multithreaded component that renders a website. This includes HTML parsing, maintaining the DOM, and implementing the web APIs that are exposed to V8. Additionally, the rendering engine schedules all macro-tasks consisting of JS code to be executed by V8. Each render process also includes an instance of the V8 engine, which executes all JS components (scripts, callbacks, and event handlers) of a web application. The bindings layer glues together Blink and V8 by exposing functionality such as the Blink APIs needed to enable programmatic DOM changes from JS code. The bindings layer includes over 4,000 APIs and 2,000 call attributes that are exposed to V8. These extensive APIs are automatically generated when Chromium is compiled, via a bindings generator that transcompiles WebIDL files into C++ code [101]. The platform layer is an additional mid-
dle layer that communicates with the browser process. Both Blink and V8 communicate with the browser process through this layer.

**Threading & Cross-Context Communication.** The render process includes many threads (e.g. render thread, HTML parsing thread, compositing thread, etc.) To simplify the discussion, we focus on threads that execute JS code: the RenderThread, WebWorker Thread(s), and the ServiceWorker Thread (see subsection 4.2.2). Scripts that are included or inserted into the page’s document run on the RenderThread. Additionally, JS code running on the RenderThread can create *WorkerThreads*. However, only JS code running in the RenderThread can access the DOM. JS code running in worker threads have access to a number of APIs, including *postMessage* to enable communication with the RenderThread, but cannot directly manipulate the DOM. Due to this strong isolation from the DOM, JS code can be viewed as single-threaded (with respect of how it affects page rendering and events).

### 4.2.2 Service Workers

Service Workers (SWs) are JS workers that execute in the background, independently from the web page that registered them. SWs are designed to improve the offline web app experience during times of lost connectivity. Due to their special role, SWs can register for events that are not available in the window context. This includes the *fetch* event, which allows the SW to intercept and modify all network requests related to the origin that registered the SW. Additionally, SWs can register for *Push* events, which allows them to receive push messages from a backend server and issue browser notifications. Recent work has shown that SWs can be leveraged to launch a variety of sophisticated web-based attacks [96, 93, 94, 95].

### 4.3 Motivating Example & Our Approach

In this section, we provide an example to motivate WEBRR’s design decisions and to point out the limitations of prior work. This motivating example, shown in Listing 4.1, is
const URL = "https://malicious-server.com"

async function getpayload() {
    const res = await fetch(URL, {
        method: "POST",
        body: {"type": "getPayload"}})
    document.body.innerHTML = await res.text()
}

async function heartbeat(idleDeadline) {
    const res = await fetch(URL, {
        method: "POST",
        body: { "type": "heartbeat",
                "now": Date.now()}})
    window.requestIdleCallback(heartbeat)
}

window.requestIdleCallback(heartbeat)
setTimeout(getpayload(), 5000)

Listing 4.1: deliver.js

inspired by how modern browser exploitation frameworks like BeEF [102] monitor and deliver payloads to their victims. BeEF (Browser Exploitation Framework) is an open source framework commonly used by nation-state adversaries to launch web-based attacks on their victims [103, 104, 105]. First, the malicious script, deliver.js, establishes a heartbeat between the victim and the server using the heartbeat function. The heartbeat is used to monitor the user’s browsing session and to send fingerprinted information to the server (we omit the fingerprinting for clarity). Next, the script uses the fetch API to get the malicious payload from the server. Finally, once the response for the payload is received, it will change the HTML of the page to the contents in the response.

In this example, WebRR’s goal is to replay the entire interaction between the malicious app and the victim. Unfortunately, we found that prior work [36] fails to replay the payload delivered by deliver.js due to uncaptured non-determinism which arises from the heartbeat mechanism. Specifically, the number of heartbeats that occur prior to receiving the payload is nondeterministic. This is because idle callbacks are only executed in the browser’s idle periods (which vary across different executions). When attempting to
replay this on prior work, we found that the payload response is not returned, but instead an empty response related to the heartbeat is returned. Upon further investigation, we found the underlying issue to be that the JS execution diverges during the replay, because fewer heartbeats occurred in the replay, compared to the original execution. Consequently, when the `getPayload` callback is executed, WebCapsule returns the wrong network response. This is due to *executional-divergence*, which occurs due to the non-deterministic behaviors that arise in JS scheduling. While WebCapsule does attempt to alleviate this issue by using a key-value data structure, where the keys are URLs and the values are FIFO-based queue of responses, this is not effective in scenarios where the URLs are the same (e.g., server-side routing is being used). More concerning, we found that executional-divergence fundamentally breaks key design assumptions made by WebCapsule. Specifically, WebCapsule assumes the order of JS execution will be the same during the replay and recording. Because of this assumption, when executional-divergence occurs during the replay, WebCapsule can no longer reliably return the values observed during the recording for sources of non-determinism (e.g., `Math.random`, `Date` values, network requests, etc.) because it cannot determine if the callback or script requesting the non-deterministic value is the same one as during the recording. During a forensic investigation, *executional-divergence* can lead to the attack failing to be replayed and lead the forensic investigator coming to the conclusion that an attack did not occur at all.

4.3.1 Our Approach

The reason prior works fail to properly replay web applications when executional divergence occurs is because they view the web application as a single monolithic unit of execution, which is fundamentally different then the event-loop execution model used by JS, where individual units of JavaScript (scripts, event handlers, and callbacks) are scheduled to be executed one-by-one. Additionally, in modern browsers, external factors such as user inputs, network latency time, and CPU load influence the order in which these units are exe-
cuted. Consequently, if the order of execution changes during the replay, due to executional divergence, prior work is unaware of these changes, and it fails to replay the attack.

In order to address this, we reimagine how to replay the web application by partitioning the web application’s JavaScript execution, into a sequence of individual *JavaScript Execution Units (JEUs)*. This allows us to independently replay these units and enforce the same order-of-execution that was observed during recording, which alleviates the issue of *executional-divergence*. For example, in *WEBRR*’s approach, the *deliver.js* script would include a JEU for the *deliver.js* script, another JEU for the *getPayload* callback, and a JEU for each time the *heartbeat* was executed. Next, during the replay *WEBRR* will schedule each JEU to be executed at the same point in the execution sequence. This makes the sequence in which the JS execution become deterministic. Finally, as JEU’s are replayed, *WEBRR* ensures that any sources of non-determinism that were queried upon by the JEU return the same values observed in the recording.

**Threat Model.** We envision *WEBRR* deployed in an enterprise environment to enable forensic investigations to be conducted on web-based attacks. As users interact with websites, our system generates logs capable of replaying each visit. We assume that these recorded logs are tamper-proof, which can be achieved by encrypting the logs and storing them on a secure file server. As these logs store personal information about the user, the user’s privacy must be considered. To minimize the personal information released during an investigation, the logs from different sites and time periods can be encrypted with different keys. These keys can then be securely stored in a key escrow, as proposed in previous work [36]. Then, only the logs necessary for the investigation can be decrypted and shared with analysts. Additionally, we assume that the browser application is not compromised at the time of the attack and, as such, the logs can be trusted (note: assuming a trusted computing base is common in the auditing community [21, 27, 16, 18, 19, 12, 35, 36, 37, 10, 11, 15, 22, 14, 23, 24, 13, 25, 26, 28, 29, 30, 31, 32, 33, 34]). If the browser is compromised, *WEBRR* will still record everything from the attack setup, until the actual
compromise occurs. Therefore, an analyst can replay the attack up until the point of compromise. We present such an example in our evaluation. This allows WEBRR to achieve a highly deterministic replay.

4.4 WEBRR

WEBRR is designed to be a forensic-grade record and replay system. With this goal in mind, we developed WEBRR such that it can be an always-on, portable, tamper proof, and deterministic system. In a forensic setting, WEBRR’s recording needs to be always-on, since it will not be known when the attack occurs ahead of time. To support this always-on property, WEBRR’s design takes special care to not introduce any instrumentation that significantly hinders the browser’s performance. Next, to maximize portability, WEBRR was implemented within Blink and on top of Chromium’s existing DevTools framework [98]. By strategically placing our instrumentation in Blink, it allows WEBRR to be OS- and device- independent and support a large set of different operating systems and devices. Additionally, while WEBRR is designed as a prototype in Chrome, it can be easily ported to other browsers that use Blink (e.g., Brave, Edge, Opera, etc.). Next, we designed WEBRR to be tamper proof such that the customizations are not accessible from JS applications (e.g., we do not attach any new properties to the window object). Finally, WEBRR seeks to provide a deterministic replay that allows analysts to faithfully replay attacks, enabling a accuracy and fine-grained forensic investigation.

High-Level Record & Replay Strategy. During the recording WEBRR will partition the JS execution into individual units using JS execution unit partitioning (discussed in subsection 4.4.1). Next, during the replay we sequentially replay each JEU in a synchronous manner while also preserving the recorded execution order (discussed in subsection 4.4.2). Additionally, during the replay, WEBRR must ensure that right before a JEU is replayed, the the DOM state is consistent with what was observed during recording at this point in the execution sequence. Therefore, during the recording, WEBRR will record the DOM state
and synchronize the DOM when necessary in the replay (discussed in subsection 4.4.3). Finally, as JEUs are executing during the replay, they will query upon sources of non-determinism such as network requests and non-deterministic web APIs (discussed in subsection 4.4.4).

4.4.1 JavaScript Execution Unit Partitioning

During the recording, our goal is to partition the JS execution into a sequence of JEUs. A JS execution unit (JEUs) represents a single unit of JS execution, such as when a script is inserted into the DOM, an event is fired, or a callback is executed. In order to partition the execution, we add hooks into Blink, which allow us to record when a JEU begins and ends execution using our JS-Execution Unit Recorder (JEU Recorder) module. Below, we explain the locations that we instrumented in Blink to partition the JS execution into a sequence of units.

**Script Units.** In Blink, when a script tag is inserted into the DOM, the corresponding page’s ScriptController is called, which calls V8 to execute the associated script. Specifically, the ExecuteScriptAndReturnValue passes the script to V8 to be executed. To record script execution, we insert hooks at the point where Blink passes control to V8. For each script executed, WEBRR records the frame id, script id, sequence number, and hash of the script’s source code. We do not have to record the script’s source code because this will be recorded by the Blink-Platform recording shims when the network request for the script is made. This is also the case for inline scripts since they are stored in the HTML document.

**Callback Units.** Callbacks represent the execution of callbacks registered by the application’s JS code. The most common web APIs used to register callbacks are setTimeout and setInterval. When setTimeout and setInterval are called, Blink creates a DOMTimer object. This object encapsulates the callback and Blink will schedule the DOMTimer::Execute on the render thread to execute the callback after the specified
time period. To record these callback units, we add hooks inside `DOMTimer::Execute` to record when the callback’s execution begins and finishes. Additionally, `setTimeout` and `setInterval` return a timeout id, which we record to determine which callback is registered during the replay. Finally, `WEBRR` also supports recording callbacks registered with `requestIdleCallback` and `requestAnimationFrame`.

**Event Units.** To record events, we insert hooks into the `EventTarget::FireEventListener` and record the necessary information to reconstruct the event during the replay. Finally, to minimize the data footprint, we only record events that actually have event listeners registered for them.

### 4.4.2 Replaying JEU

During the replay, `WEBRR` schedules the execution of each JEU to force the web application’s JS code to execute in the same order as what was recorded. To achieve this, we designed a replay scheduler, which ensures that all JEUUs are replayed in the same sequence as observed during the recording. The scheduler has two major components, the *Replay Queue* and *Replay Dispatcher* which are discussed below.

When a replay is initialized, a recording is loaded into `WEBRR`. The JEUUs recorded by the JEU Recorder are stored in the *replay queue*. The queue is a FIFO queue and
Figure 4.2: A high-level overview of the workflow that occurs when the replay dispatcher is executed on the render thread.

operations are inserted based on when they occurred during the recording. At a high-level, a replay operation is a data structure that contains metadata related to a JEU (more details in subsubsection 4.4.2). During the replay, operations are pulled off the queue and scheduled to be executed on the render thread by the *Replay Dispatcher*. This process is completed until the queue is empty. This approach is displayed in Figure 4.1. First, when the replay begins, **WEBRR** schedules the replay dispatcher by posting it as a task on the render thread’s task queue. Next, the render thread’s scheduler will execute the replay dispatcher task. When this occurs, the dispatcher will pop the next replay operation off the queue, process it, and execute it on the render thread. How the operation is “processed” varies based on the type of replay operation that is popped off the queue (discussed further in subsubsection 4.4.2). Finally, after the operation has been completed, **WEBRR** repeats this process by placing the dispatcher task back in the render thread’s task queue until the replay operation queue is empty. At first glance, the process of repeatedly rescheduling the dispatcher to be executed on the render thread may appear unnecessary. However, this is necessary in order to ensure the other components of the rendering pipeline (e.g., layout and paint) are given time to be executed, in addition to the replay operations.
Replay Operations

Internally, each operation is a map data structure that contains the necessary information for the replay engine to process the operation. In Figure 4.2, we provided a high-level description of how a replay operation is processed. First, the replay dispatcher pulls a replay operation off the queue. Next, the dispatcher dispatches this operation to the correct frame. This is necessary when the page we are replaying has multiple frames (iframes). Internally, we do not dispatch this operation directly to the frame. Instead, we dispatch it to WebRR’s internal ReplayFrame class, which is a thin wrapper around Blink’s LocalFrame. Blink uses LocalFrame objects to encapsulate and isolate the rendering of individual frames. During the replay, WebRR also isolates the replay of different frames, and the ReplayFrame is responsible for driving the replay of the LocalFrame it encapsulates. The remainder of this section discusses the JSReplayer and the Event Replayer.

JSReplayer

The JSReplayer is responsible for replaying all script and callback JEUs in the correct order. To achieve this, the JSReplayer takes over the responsibility of scheduling script and callback units to be executed on the render thread. Specifically, when JS registers callbacks using APIs such as setTimeout and setInterval, Blink does some bookkeeping and then places the registered callbacks into the render thread’s task queue to be executed later. However, during a replay, WebRR needs to control when callbacks execute to ensure the replayed sequence of JEUs matches the recorded sequence. To accomplish this, the JSReplayer maintains a registration data structure that it uses to determine when each script or callback should begin executing. During replay, when Blink is about to place a callback on the render thread’s task queue, WebRR intervenes and instead saves the callback in the registration data structure according to a unique id. At this point, WebRR has complete control over when the callback starts and waits until the correct time to run it. While this
discussion mainly focuses on callbacks, the approach for handling script tags is similar and is discussed in detail below.

**Script Registration.** In Blink, each LocalFrame has a reference to a `ScriptRunner` object and a `ResourceFetcher` object. The `ResourceFetcher` class downloads resources from the network for the frame, and the `ScriptRunner` serves as a middleman between the `ResourceFetcher` and the render thread’s scheduler. The `ScriptRunner` manages the state of scripts as they transition from a *pending state* to a *ready state*. When a script is in a “pending” state, the `ScriptRunner` is waiting on this script’s source code to be fetched from the network by the `ResourceFetcher`. Once the `ResourceFetcher` has completed downloading the script, the script will transition to a *ready state*, which in turn will signal the `ScriptRunner` to schedule this script to be executed on the render thread. During the replay, we add registration hooks in the `ScriptRunner::NotifyScriptReady` method to force all scripts to register themselves with their frame’s `JSReplayer`.

**Callback Registration.** Web apps can register callbacks to be executed at a later point in time using APIs such as `setTimeout` and `setInterval`. Within Blink, when a callback is registered using these APIs, it creates a `DOMTimer` object, which manages executing this callback. During the replay, we add hooks to the timer’s constructor, such that when a `DOMTimer` is created, it will be registered with the `JSReplayer`. WEBRR also supports replaying callbacks registered with `requestIdleCallback` and `requestAnimationFrame`.

**Script & Callback Execution.** The final aspect of the `JSReplayer` is how and when it executes script and callback units. When the `JSReplayer` receives a replay operation, the operation contains a unique identifier that is used to determine which JEU should be executed. For scripts, this is the hash of the source code and for callbacks, it is the id provided by the return value. Next, the `JSReplayer` will look up the correct JEU in its registration map by using the unique-identifier. Finally, the `JSReplayer` will synchronously execute the JEU.
**EventReplayer**

The EventReplayer is responsible for replaying event-related replay operations, and it achieves this by reconstructing the original event that occurred during the recording and then firing this event at the correct EventTarget, which will invoke any event listeners for this event. Event reconstruction is done by creating an Event object using the information provided in the replay operation. Specifically, each replay operation contains the necessary information to create a blink::Event class or one of its subclasses (e.g., MouseEvent, KeyboardEvent, etc).

### 4.4.3 Maintaining the DOM State

During the replay, WEBRR must ensure that right before a JEU is replayed, the state of the DOM exactly matches what was observed during the recording right before the related JEU was executed. This ensures that during replay, the JEU will be provided the same inputs that it observed during the recording phase. A main challenge is that browsers support script attributes such as async and defer, which allow the page’s HTML to be parsed in parallel to the fetching of resources; this is a possible source of non-determinism during replay. Therefore, to achieve deterministic replay, the state of the DOM must be tracked and recorded. To achieve this, WEBRR uses a DOM Recorder, which records insertions into the DOM tree made by the parser. For each DOM node inserted, we record the necessary information to reconstruct the same DOM tree during the replay in the correct sequence.

During the replay, to ensure that the state of the DOM is exactly the same as during recording before a JEU is executed, WEBRR disables Blink’s HTML parser and uses a custom DOMReplayer to control when DOM nodes are inserted into the page’s document (note that during recording, WEBRR does not change the parser or page rendering; changes are applied only during replay). The main challenge for the DOMReplayer is that it needs to ensure that prior to the execution of a JEU, the DOM state is the same as it was during the recording. In order to achieve this, we create additional replay operations to represent
DOM construction operations. These DOM construction operations are then placed in the same replay queue as the replay operations related to JEUs, such that if a DOM construction operation occurred prior to the execution of the JEU is it guaranteed to occur prior to its execution during the replay.

4.4.4 Handling Sources of Nondeterminism

The final piece to WEBRR’s replay system is the shims used to capture and replay sources of non-determinism. As JEUs are executing during the replay, they will query upon sources of non-determinism such as network requests and non-deterministic web APIs. To support a deterministic replay, WEBRR must record these values. This is achieved by using a set of shims (Blink-Platform shims, Blink-V8 shims, and V8-Platform shim) which are discussed in the remainder of this section. Additionally, during the replay all shims are placed in replay mode and any function calls that pass through them will return the value observed in the recording.

**Blink-Platform Shims**

The *Blink-Platform* shims record responses to all network requests made by an application. This includes network requests that fetch resources (e.g., images, scripts, etc.) and Ajax requests made by JS via `XMLHttpRequest` and `fetch` APIs. We implemented this shim at the Blink-Platform layer because all network request made by the application flow through this layer. This allows us to create a single recording hook that can record all network responses. Specifically, the `ResourceFetcher` class is used to handle all network requests for a render process. The `ResourceFetcher::HandleLoaderFinish` method is called when a response is received. We implement WEBRR’s recording shim at this point to record the network response. During the replay, WEBRR needs to uniquely identify each network request so that it can return the correct response. For this, WEBRR maintains a map data structure where the key identifies the unique request and the value
is the recorded response. The key contains the request’s URL, resource type, and HTTP
method. We found this approach to work well in practice with no collisions. For the re-
response, WEBRR records the response’s HTTP status, headers, and payload.

**Blink-V8 Bindings Layer Shims**

The Blink-V8 bindings layer shims capture non-deterministic values that flow into V8 via
the web APIs exposed by Blink. While most web APIs are deterministic, there are several
APIs that return non-deterministic values. For example, the APIs exposed by the *Perfor-
mance* module are dependent on the current execution speed, which differs across execu-
tion. Additionally, we consider APIs related to storage and cookies to be non-deterministic.

The main challenge with capturing non-deterministic values returned by web APIs is
the amount of APIs that needs to be hooked. As discussed in section 4.2, the bindings
layer exposes the web APIs that are implemented in Blink to JS applications. These APIs
are exposed through C++ code, generated using Chromium’s Bindings compiler [101]. To
avoid manually hooking every API, we customize the bindings compiler to add our hooks
to the necessary APIs during compilation. These customizations rely on modifying the
templates used by the compiler to generate the C++ code for web attributes [106] and web
APIs [107]. This approach can also facilitate porting WEBRR’s instrumentation to future
browser releases.

For each attribute or method that we need to hook, WEBRR introduces a small snippet
of code that redirects the control to our recording engine when a hooked web API is called.
Next, WEBRR records the method or attribute that was called and the value that will be
returned and then returns control back to the bindings layer and the execution flow resumes
as usual.
**V8-Platform Shims**

The V8-Platform shims record non-deterministic data that flows into V8 via the V8-platform boundary including values returned by the `Date` object’s methods and values returned by `Math.random`. To record these values, we made minor modifications to the V8 engine at the V8-Platform layer. First, calls to the `Date` module go through the platform layer to get the current time via `CurrentClockTimeMillis`, which returns the current time’s timestamp. We add a small wrapper around this method that records return values. For each recorded value, we record the execution context’s id to determine context this value occurred. Instrumenting `Math.random` works in a similar way, where WEBRR inserts a small wrapper around calls to `Math.random` to record return values.

**Workers & Frame Communication**

As previously discussed in section 4.2, JS can run script operations on background threads using the Web Worker’s Interface [108]. It is important to point out that web workers have their own context (i.e., they do not share memory with JS running on the render thread). Instead, communication between workers and render contexts occurs via message passing through the `postMessage` API. We decided to not record worker threads and instead record the communication between the threads, which allows us to isolate the replay of threads. Currently, WEBRR supports recording inbound messages to the render thread. However, it could easily be extended to support replaying workers in isolation. Since WEBRR’s main goal is to support replaying web-based attacks and attacks rarely occur in the worker context, we leave this to future work.

We also want to point out that frames (i.e. iframes and mainframe) also run in their own contexts and communicate via the `postMessage` interface. WEBRR’s approach to handling this communication is to treat it as a source of non-determinism and record all messages passed between frames.
4.4.5 Replaying Service Workers

To support replaying SWs, we developed a SW-mode, which allows a forensic analyst to replay an application’s SW in isolation (i.e., it does not need to replay the ”webpage” portion of the web app). In SW-mode, WEBRR updates the replay scheduler so that the replay dispatcher task is posted to the SW’s task queue, instead of the render thread’s task queue. This allows the dispatcher to be executed on the SW thread and execute replay operations in the SW’s context. Additionally, we added support for events that are only available in the SW context (e.g., push notifications, sync, and lifecycle events [109]). One challenge we had to address was how to record both the web application’s main context and the SW context at the same time, since an attack may occur in either contexts. In order to achieve this, we strategically designed WEBRR’s recording engine to be stateless, so that its recording hooks could be called on different threads. This allows the APIs and JEU Recorder to be able to record the SW and render threat simultaneously.

4.5 Evaluation

Our evaluation addresses the following research questions:

- How well does WEBRR replay sophisticated real-world web-based attacks during a forensic investigation?
- Can WEBRR replay highly dynamic web applications?
- How well does WEBRR perform compared to existing state-of-the-art systems?
- What is the runtime performance overhead and data storage requirements of WEBRR’s recording engine?
4.5.1 Experimental Setup

We complete WEBRR’s evaluation on a diverse set of devices and operating system to demonstrate WEBRR’s portability. For Linux, we used a standard desktop with a AMD Ryzen 9 5950X 16-Core 2.2GHz CPU and 128 GB of physical memory, running Ubuntu 20.04. For Windows, we used a standard desktop with an AMD Ryzen 9 3900X 12-Core 3.9GHz CPU and 128 GB of physical memory running Windows 10. For Android, we used an emulated Pixel 6 XL device running Android version 12.0 (S) - API 31. We implemented WEBRR by instrumenting Chromium version 83.0.4103.97

Evaluation Metrics

To evaluate WEBRR’s capability to deterministically replay web-based attacks and benign websites, we consider four major factors: (i) the website is correctly recorded, (ii) the JEU sequence in the replay matches the JEU sequence of the recording, (iii) sources of nondeterminism return the values observed during recording, and (iv) the replay is visually deterministic. We discuss how we measure each factor below.

Successful Recording. We consider the recording to be successful if the website was recorded, the browser did not crash, and there were no significant issues with the website.

Comparing JEU sequences. For each experiment, we log the sequence of JEUs both during recording and replay and compare how closely the two sequences align. To measure JEU sequence alignment, we use the Levenshtein Distance [110] (i.e., edit distance) Algorithm, which is a common approach used for measuring the difference between two sequences. In this evaluation, smaller distance values are better and a score of zero means the sequences are perfectly aligned.

Verifying Return Values of Nondeterministic Sources. Another important aspect of the replay that needs to be evaluated is (i) if our replay shims are correctly returning the values observed during the recording and (ii) that we support all APIs that may be returning non-
deterministic values. To measure this, we customize WEBRR so that it records every web API that is invoked in the Blink-V8 bindings. Additionally, for APIs that return primitive types, we record these values. This allows us to record the web API sequence that occurred during the recording and the replay. Each entry in the sequence is a tuple that includes the method name and the value the API call returned. Next, similar to how we compared the JEU execution sequences, we compare the API call sequences, which allows us to verify that each invoked API returns the same value as what was observed during the recording.

**Visually Deterministic.** We consider a replay to be visually deterministic if the visual layout of the replayed execution is consistent with what was observed during the recording. To evaluate visual determinism, we take screenshots during the recording and replay prior to the execution of a JEU unit related to user inputs (notice that, for the sake of this measurements, we instrumented WEBRR to synchronously take screenshots precisely at the time of user inputs by temporarily pausing the render thread prior to taking the screenshot). Essentially, this allowed us to create pairs of screenshots \((\text{rec}, \text{repl})\) that capture the page rendering at the exact same point in the execution sequence for the record and replay traces. Therefore, we expect the screenshots to be visually equivalent. To measure the similarity between these screenshots, we train a deep learning-based similarity function, similar to state-of-the-art methods used in recent phishing detection work [111]. Specifically, we use SimCLRv2 [112], a contrastive learning method for measuring image similarity. This method trains a neural network to output encoding vectors for input images such that similar images map to similar image embeddings. Starting from the publicly available pre-trained SimCLRv2 model, we fine-tuned it to measure the similarity among web pages, using a dataset of 7,203 web page screenshots collected by us from the Tranco 1K [97] list of websites. During the evaluation, we embed the corresponding recorded and replayed screenshot pairs \((\text{rec}, \text{repl})\) for a given website, and report the mean cosine similarity between the two embedding vectors over all pairs for the same website. If the cosine similarity between the two vectors is one, it means the model considers these screenshots
to be an exact match.

4.5.2 Record & Replay Evaluation

In this section, we evaluate how well WEBRR is able to replay web-based attacks and popular-but-benign websites.

**Attacks Used.** We evaluated WEBRR on seven real-world web-based attacks, shown in the *Attack* column in Table 4.2. We evaluate the same set of attacks on all three test platforms: Linux, Windows, and Android. We chose these attacks because they i) represent a diverse set of different web-based attacks, ii) are commonly observed at enterprise organizations [113], and iii) similar attacks have been used to evaluate prior work [11, 29, 30, 37]. We also included a drive-by-download attack because it exploits a bug in the version of Chromium on which WEBRR is built and demonstrates that WEBRR is also able to replay attacks that exploit the browser itself (see section 4.6 for a discussion of related limitations). The first attack (attack 1) is a phishing-based attack that was collected from OpenPhish [114]. The second attack (attack 2) is a credential harvesting attack inspired by a real-world attack carried out by the OceanLotus APT group [5].

For the other attacks, we set up several different types of attacks locally and evaluated how well WEBRR could replay them. This included a keylogger that relied on XSS (attack 3), a Clickjacking attack (attack 4), and a driveBy attack that relied on exploiting a type-confusion vulnerability in V8 to (attack 5). Furthermore, we record and replay two recent web attacks that make use of Service Workers (SW) and thus can target modern progressive web apps (attacks 6 and 7). For each attack, we started WEBRR in *recording* mode and navigated to the malicious website that hosted the attack. Next, we interacted with the website so that it would trigger the attack. For all of the attacks, we verified that WEBRR properly recorded the attack without errors.

**Popular Websites Used.** We used the benign websites in the *Website* column in Table 4.1. The diverse set of websites we selected were all found on the Tranco 1k list [97].
Table 4.1: Evaluation of replayed benign websites. For each site, we show the recording operating system (OS), recorded JEU sequence length (Rec-JEU Seq. Len.), replayed JEU sequence length (Rep-JEU Seq. Len.), JEU sequence edit distance (JEU-Seq. E. D.), recorded API sequence length (Rec-API Seq. Len.), replayed API sequence length (Rep-API Seq. Len.), API sequence edit distance (API Seq. E. D.).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lin.</td>
<td>stackoverflow</td>
<td>134</td>
<td>134</td>
<td>0</td>
<td>12,868</td>
<td>12,865</td>
<td>13</td>
</tr>
<tr>
<td>Lin.</td>
<td>wikipedia</td>
<td>181</td>
<td>181</td>
<td>0</td>
<td>52,700</td>
<td>52,700</td>
<td>4</td>
</tr>
<tr>
<td>Lin.</td>
<td>whitehouse</td>
<td>60</td>
<td>60</td>
<td>0</td>
<td>6,148</td>
<td>6,148</td>
<td>4</td>
</tr>
<tr>
<td>Lin.</td>
<td>mozilla</td>
<td>141</td>
<td>141</td>
<td>0</td>
<td>8,078</td>
<td>8,078</td>
<td>0</td>
</tr>
<tr>
<td>Lin.</td>
<td>craigslist</td>
<td>345</td>
<td>345</td>
<td>0</td>
<td>26,534</td>
<td>26,536</td>
<td>8</td>
</tr>
<tr>
<td>Win.</td>
<td>stackoverflow</td>
<td>146</td>
<td>146</td>
<td>0</td>
<td>14,477</td>
<td>14,477</td>
<td>11</td>
</tr>
<tr>
<td>Win.</td>
<td>wikipedia</td>
<td>173</td>
<td>173</td>
<td>0</td>
<td>6,790</td>
<td>6,794</td>
<td>14</td>
</tr>
<tr>
<td>Win.</td>
<td>whitehouse</td>
<td>39</td>
<td>39</td>
<td>0</td>
<td>5,878</td>
<td>5,878</td>
<td>5</td>
</tr>
<tr>
<td>Win.</td>
<td>mozilla</td>
<td>91</td>
<td>91</td>
<td>0</td>
<td>4,557</td>
<td>4,565</td>
<td>12</td>
</tr>
<tr>
<td>Win.</td>
<td>craigslist</td>
<td>52</td>
<td>52</td>
<td>0</td>
<td>4,546</td>
<td>4,546</td>
<td>10</td>
</tr>
<tr>
<td>Andr.</td>
<td>stackoverflow</td>
<td>161</td>
<td>161</td>
<td>0</td>
<td>15,391</td>
<td>15,397</td>
<td>18</td>
</tr>
<tr>
<td>Andr.</td>
<td>wikipedia</td>
<td>69</td>
<td>69</td>
<td>0</td>
<td>26,790</td>
<td>26,790</td>
<td>12</td>
</tr>
<tr>
<td>Andr.</td>
<td>whitehouse</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>5,909</td>
<td>5,911</td>
<td>13</td>
</tr>
<tr>
<td>Andr.</td>
<td>mozilla</td>
<td>67</td>
<td>67</td>
<td>0</td>
<td>1,769</td>
<td>1,769</td>
<td>0</td>
</tr>
<tr>
<td>Andr.</td>
<td>craigslist</td>
<td>58</td>
<td>58</td>
<td>0</td>
<td>6,820</td>
<td>6,822</td>
<td>0</td>
</tr>
</tbody>
</table>

visited each website using WEBRR in *recording mode* on Windows 10, Ubuntu 20.04, and Android 12. During each visit, we heavily interacted with the website to simulate a typical visit to this website. For all of the websites, we verified that WEBRR properly recorded the browsing sessions.

*JavaScript Execution Determinism*

In this section, we focus on evaluating the JS execution and how accurately it was replayed during the experiments. After recording each attack and website used, we replayed the browser trace s using WEBRR in *replay mode*. Next, we compared the recorded JEU execution sequence to the replayed sequence. For the attacks, the results are shown in Table 4.2 in the *JEU Sequence Edit Distance* Column. We see that for all attacks, the edit distance between the recorded and replayed execution sequence was zero, which means the sequences were exact matches. Additionally, for all benign websites, the JEU execution sequence during the replay matched the recorded sequence, which is shown in Table 4.1. This shows that WEBRR can replay the JEU execution sequence exceptionally well for web-based attacks and popular benign websites.

The final portion of the evaluation verified that the sources of nondeterminism return
Table 4.2: Web-Based Attack Results. For each attack, we show the recording operating system (OS), the recorded JEU sequence length (Rec-JEU Seq. Len.), replayed JEU sequence length (Rep-JEU Seq. Len.), JEU sequence edit distance (JEU-Seq. E. D.), recorded API sequence length (Rec-API Seq. Len.), replayed API sequence length (Rep-API Seq. Len.), API sequence edit distance (API Seq. E. D.), and whether the attack was replayed correctly (Replayed).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lin.</td>
<td>Phish.</td>
<td>96</td>
<td>96</td>
<td>0</td>
<td>326</td>
<td>326</td>
<td>0</td>
</tr>
<tr>
<td>Lin.</td>
<td>Cred. Harv.</td>
<td>977</td>
<td>977</td>
<td>0</td>
<td>3,733</td>
<td>3,733</td>
<td>0</td>
</tr>
<tr>
<td>Lin.</td>
<td>KeyLogger</td>
<td>51</td>
<td>51</td>
<td>0</td>
<td>97</td>
<td>97</td>
<td>0</td>
</tr>
<tr>
<td>Lin.</td>
<td>Click. Jack.</td>
<td>189</td>
<td>189</td>
<td>0</td>
<td>131</td>
<td>131</td>
<td>0</td>
</tr>
<tr>
<td>Lin.</td>
<td>DriveBy</td>
<td>93</td>
<td>93</td>
<td>0</td>
<td>263</td>
<td>263</td>
<td>0</td>
</tr>
<tr>
<td>Lin.</td>
<td>StealthyPush.</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>73</td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td>Lin.</td>
<td>XXS-SW</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Win.</td>
<td>Phish.</td>
<td>96</td>
<td>96</td>
<td>0</td>
<td>326</td>
<td>326</td>
<td>0</td>
</tr>
<tr>
<td>Win.</td>
<td>Cred. Harv.</td>
<td>15</td>
<td>15</td>
<td>0</td>
<td>1389</td>
<td>1389</td>
<td>0</td>
</tr>
<tr>
<td>Win.</td>
<td>KeyLogger</td>
<td>55</td>
<td>55</td>
<td>0</td>
<td>61</td>
<td>61</td>
<td>0</td>
</tr>
<tr>
<td>Win.</td>
<td>Click. Jack.</td>
<td>38</td>
<td>38</td>
<td>0</td>
<td>259</td>
<td>259</td>
<td>0</td>
</tr>
<tr>
<td>Win.</td>
<td>DriveBy</td>
<td>298</td>
<td>298</td>
<td>0</td>
<td>2441</td>
<td>2441</td>
<td>0</td>
</tr>
<tr>
<td>Win.</td>
<td>StealthyPush.</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>73</td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td>Win.</td>
<td>XXS-SW</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Andr.</td>
<td>Phish.</td>
<td>55</td>
<td>55</td>
<td>0</td>
<td>5,127</td>
<td>5,127</td>
<td>0</td>
</tr>
<tr>
<td>Andr.</td>
<td>Cred. Harv.</td>
<td>211</td>
<td>211</td>
<td>0</td>
<td>5,504</td>
<td>5,504</td>
<td>0</td>
</tr>
<tr>
<td>Andr.</td>
<td>KeyLogger</td>
<td>92</td>
<td>92</td>
<td>0</td>
<td>80</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>Andr.</td>
<td>Click. Jack.</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>266</td>
<td>266</td>
<td>0</td>
</tr>
<tr>
<td>Andr.</td>
<td>DriveBy</td>
<td>353</td>
<td>353</td>
<td>0</td>
<td>2,571</td>
<td>2,571</td>
<td>0</td>
</tr>
<tr>
<td>Andr.</td>
<td>StealthyPush.</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>73</td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td>Andr.</td>
<td>XXS-SW</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

the same values. These results are shown in the API Sequence Edit Distance column in Table 4.2 for the attacks and in Table 4.1 for the popular websites. We found that for all attacks, the APIs queried returned the same values as what was observed during the recording. For the popular websites, we see that for almost all APIs the values returned are the same as what was observed during the recording. However, we see that in a few scenarios (≤ 18 out of 15,397, or ≈ 0.1% of cases) some APIs returned different values during replay. We found that for these scenarios, we returned the wrong node type [115] for the Document node. This issue is related to the unique method we are using to reconstruct the DOM during replay, and could be resolved with some additional engineering effort. Despite this issue, we would like to emphasize that this did not have any side effects on the remaining portions of the replay. Since we were able to replay all remaining components and APIs in a fully deterministic manner, we argue this is a successful replay of the websites. Also, in our evaluation we discovered a few engineering-related challenges with replaying highly-optimized websites such as Amazon, which we discuss in section 4.6.
Table 4.3: The experimental results for visual determinism.

<table>
<thead>
<tr>
<th>Attacks</th>
<th>Screenshots</th>
<th>Avg. Cosine Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phishing</td>
<td>13</td>
<td>1.00</td>
</tr>
<tr>
<td>Credential Harvesting</td>
<td>23</td>
<td>1.00</td>
</tr>
<tr>
<td>KeyLogger</td>
<td>16</td>
<td>1.00</td>
</tr>
<tr>
<td>Clickjacking</td>
<td>12</td>
<td>1.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Popular Websites</th>
<th>Screenshots</th>
<th>Avg. Cosine Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>mozilla.org</td>
<td>39</td>
<td>1.00</td>
</tr>
<tr>
<td>wikipedia.org</td>
<td>10</td>
<td>.99</td>
</tr>
<tr>
<td>whitehouse.gov</td>
<td>10</td>
<td>1.00</td>
</tr>
<tr>
<td>craigslist.org</td>
<td>14</td>
<td>1.00</td>
</tr>
<tr>
<td>stackoverflow.com</td>
<td>22</td>
<td>.94</td>
</tr>
</tbody>
</table>

**Visual Determinism**

In this section, we focus on evaluating the visual component of WEBRR’s replay. For each attack, we collected screenshot pairs and generated embedding vectors for each screenshot using the approach described in subsubsection 4.5.1. The number of screenshots pairs created for each attack and the average cosine similarity between the pairs is shown in Table 4.3. We found that for the attacks, WEBRR performed exceptionally well and had an average cosine similarity of 1.00 for all attacks. This means the model found the screenshots to be exact matches. We also found that WEBRR can achieve a high visual determinism accuracy for the replayed popular benign websites. For stackoverflow, we found that the average cosine similarity dropped to 0.94 due to replaying SVG images. The underlying issue is that WEBRR currently does not support replaying SVG icons, which are heavily used on stackoverflow. The correct replay of SVG elements could be addressed with additional engineering effort, which we leave to future work.

To validate these results, we separately computed the similarity between negative samples. Specifically, we compared the cosine similarity of 7,203 pairs of web page screenshots taken from different popular websites, to confirm that our model correctly assigns a high score only to pages that are indeed visually very similar, and low scores to pages that are visually dissimilar (because they belong to different sites with different designs). On aver-
age, the similarity between pages from different popular sites was 0.03, showing that the model makes a clear distinctions between screenshots from the same and different sites. We believe this results further support the validity of our visual determinism evaluation.

4.5.3 Comparison with Existing Systems

In this section, we compare WEBRR to WebCapsule, a state-of-the-art forensic-based record-and-replay system for Chromium-based browsers. Unfortunately, evaluating WebCapsule directly using the metrics discussed in subsubsection 4.5.1 is not possible. The reason is that this would require us to instrument WebCapsule’s source code, which was built on top of Chromium version 36.0.1932.0 (released in 2014). After several attempts, we determined that, due to its age, WebCapsule can no longer be compiled without major refactoring (e.g., because of third-party dependencies being deprecated or no longer available). Furthermore, porting WebCapsule to a more recent Chromium version would require very extensive software engineering efforts, due to how much the Chromium code base has changed since 2014. Thus, this prevented us from adding the necessary instrumentation that is required to record the execution sequence and API calls, which are needed to calculate the evaluation metrics discussed in subsubsection 4.5.1.

To overcome this and perform the evaluation, we relied on the compiled binary version of WebCapsule provided by its authors [116]. Specifically, we used the binary version to answer the following question: Can WebCapsule properly replay the web-based attacks described in Section subsection 4.5.2?

For each attack, we considered the replay to be successful if i) the visual component (e.g., the social-engineering component) of the attack was displayed during the replay and ii) any user interactions, such as clicks or key events, with the visual component were also correctly replayed. The results for these experiments are shown in Table 4.4. We found that WebCapsule was only able to replay one of the attacks, while WEBRR successfully replayed them all. We discuss the results in the remainder of the section.
Table 4.4: Evaluation Results of WebRR and WebCapsule (WC) correctly replaying the web-based attacks. A ✓ in the Visual Component column means the system correctly replayed the visual component of the replay. A ✓ in the Interactions section means the system correctly replay the user interactions with the attack. A ✓ in the Attack Replayed means the visual component and the user-interactions were both correctly replayed. * means the attack was backported and – means the attack could not be recorded.

<table>
<thead>
<tr>
<th>Attack</th>
<th>Visual Component</th>
<th>Interactions</th>
<th>Attack Replayed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WebRR WC</td>
<td>WebRR WC</td>
<td>WebRR WC</td>
</tr>
<tr>
<td>Keylogger</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Phishing</td>
<td>✓ – ✓ –</td>
<td>✓ – ✓ –</td>
<td>✓ – ✓ –</td>
</tr>
<tr>
<td>Clickjacking</td>
<td>✓ – ✓ –</td>
<td>✓ – ✓ –</td>
<td>✓ – ✓ –</td>
</tr>
<tr>
<td>Credential Harvesting</td>
<td>✓ – ✓ –</td>
<td>✓ – ✓ –</td>
<td>✓ – ✓ –</td>
</tr>
<tr>
<td>Phishing*</td>
<td>✓ ✓ ✓ ✗</td>
<td>✓ ✓ ✗ ✗</td>
<td>✓ ✗ ✗ ✗</td>
</tr>
<tr>
<td>Clickjacking*</td>
<td>✓ ✗ ✓ ✗</td>
<td>✓ ✗ ✓ ✗</td>
<td>✓ ✗ ✗ ✗</td>
</tr>
<tr>
<td>Credential Harvesting*</td>
<td>✓ ✗ ✓ ✗</td>
<td>✓ ✗ ✓ ✗</td>
<td>✓ ✗ ✗ ✗</td>
</tr>
</tbody>
</table>

For the Phishing, Credential Harvesting, and Clickjacking attacks we found that these attacks were originally failing during recording, due to the fact that the version of V8 that WebCapsule relies on does not support ECMAScript6 (ES6), which was causing the browser to throw syntax errors when loading these attacks. To address this issue, we adapted these attacks by backporting them such that they only used functionality that predates ES6. We then verified for each adapted attack that the attack worked as intended while using WebCapsule in recording mode. However, we found that WebCapsule failed to replay the backported-versions of the attacks.

The Phishing and Clickjacking attacks both failed due to executional divergence related to JS scheduling. This is because the order in which scripts were executed during replay was different than what was observed during recording. For the Clickjacking attack, we found that this lead to the attack appearing to not be interacted with, because the attack was using a random value to determine when to display the attack and due to executional divergence, a different random value was returned. This caused the user-input to become “out of sync” with the JavaScript execution and giving the appearance that user did not interact with the attack. For the Phishing attack, the attack failed to load at all, because the wrong
network request was returned due to the number of heartbeats in the attack being different in the recording and the replay. This lead to WebCapsule returning a payload related to the heartbeat instead of the payload response. We provide an video demo of the phishing experiment\(^1\). For the Credential Harvesting attack, the attack relied on storing data in the browser using the `localStorage` API. We found that the attack used this data to determine if the user was targeted by the attack [5]. If the user was considered targeted, the attack would be deployed. Otherwise, no attack occurred at all. Unfortunately, during the replay we found that WebCapsule did not return the values observed during the recording, which lead to the attack not being deployed during the replay, despite the user being targeted. The underlying issue is that WebCapsule does not consider the web APIs to be a source of non-determinism, since it attempts to treat Blink as a blackbox. We provide a video recording of the Credential Harvesting experiment\(^2\).

4.5.4 Runtime Overhead

To evaluate the runtime performance of WEBRR, we measured the page load for the websites listed in the Tranco 1k [97]. We measure the page load time because previous studies have shown that a slow page load time can lead to frustrated users and drive websites’ revenue lower [76]. To measure the page load, we leveraged Chromium’s `TRACE_EVENT` instrumentation infrastructure for profiling [117]. This macro injects code to emit a timestamped event at the beginning and end of each recording hook. Using these timestamps, we can calculate the amount of time within each recording hook. Next, we measured the page load time for every website on the Tranco 1k list and determined how much overhead was induced by WEBRR’s runtime hooks. For each website, we loaded the page into the browser 10 times and recorded the median page-load overhead from these 10 runs. We provide a 5-number-summary of the results in Figure 4.3 in terms of percentages (left) and absolute time (right). The results show that WEBRR’s recording engine impact on the

\(^1\)Phishing Experiment: https://youtu.be/JGddenliISs

\(^2\)Credential Harvesting Experiment: https://youtu.be/7yxKcbhBqeQ
Figure 4.3: The runtime performance overhead induced on the page load by WEBRR for the Tranco 1k. (a) The runtime overhead increase for the page load. (b) The absolute time induced by WEBRR. Whiskers are set to (0%, 95%).
Figure 4.4: The runtime resource usage induced on the page load by \textsc{WebRR} for the Tranco 1k. (a) the distribution of the CPU usages. (b) provides distribution of the memory consumed by \textsc{WebRR} per process. \textsc{WebRR} induced negligible CPU overhead and at most 150MB memory (with 15MB overhead maximum) when visiting the Tranco 1k.

page load time is extremely low with a median increase of only 3.14%, which results in a 0.04s increase of the page load time. When looking at the outliers, we found that websites that had extremely large HTML documents were more impacted by \textsc{WebRR}’s recording hooks, because we record every DOM insertion made by the parser. For example, the website \texttt{hxxp://www.wp.pl} is an outlier and has 11.13% overhead. This is because the website’s homepage requires us to record 20,000 insertions into the DOM tree. However, the overhead induced by our DOM recorder is only two microseconds per insertion, which is low considering the tracing precision is one microsecond. Nevertheless, for 95% of the websites, \textsc{WebRR}’s runtime overhead was less than 7.24%, which shows that \textsc{WebRR}’s recording is capable of being deployed in real-world enterprise settings.

\textbf{Resource Overhead.} To evaluate \textsc{WebRR}’s resource usage overhead, we measured the CPU and memory usage for the websites listed in the Tranco 1k [97]. It is challenging to separately measure the precise resource consumption of \textsc{WebRR}’s components, because this would require sophisticated code instrumentation to calculate how much memory is
allocated and how many CPU cycles are consumed. Therefore, we leverage an alternative approach that allows us to estimate the resource usage overhead. We use the `ps` command to continuously record the CPU and memory usage of the browser processes (with 100ms granularity) while visiting the home page of every website in the Tranco top 1k list ten times (i.e., 10k page loads in total), using both vanilla Chromium and WEBRR. Every time a page is visited, we wait for the page to be fully loaded, and then wait another 10 seconds before visiting the next page. To compare the resource usage of vanilla Chromium and WEBRR, we summarize the results as Cumulative Distribution Function (CDF) graphs in Figure 4.4. As can be seen from Figure 4.4, WEBRR induces negligible CPU overhead and limited memory usage overhead. The slight increase in memory usage is due WEBRR’s need to perform data serialization and buffer browser data objects that are then recorded to the WEBRR’s trace files.

4.5.5 Storage Overhead

To measure the disk space overhead, we ran WEBRR in recording mode for a 50-minute browsing session and visited 10 heavily dynamic and popular websites. The websites used in the evaluation are shown in Table 3.10. The compressed version of WEBRR’s recording logs was only 873 MB for the entire 50-minute session. This means that, on average, the disk space requirement for WEBRR is only 17 MB per minute for highly active browsing sessions. If we assume WEBRR is deployed in a standard enterprise environment, it would only require 8.38 GB of storage space for a typical 8-hour workday. Additionally, assuming a standard 262-day work year, only 2.2TB of disk space will be required to store WEBRR’s recording logs for a single work year. WEBRR also has the capability of logging data of each tab to its own individual file, which essentially means during an investigation, the investigator would only need to decompress the logs related to the browsing sessions of interest. Finally, we want to point out that WEBRR’s recording logs are currently include a significant amount of debugging information, and the numbers presented in this section
represent an upper bound.

4.6 Discussion & Limitations

**Maintainability.** We implemented WebRR as an InspectorAgent in the DevTools Framework, thus WebRR requires a similar amount of maintenance as existing DevTools Agents (it is comparable in terms of LoCs to existing Agents). WebRR's implementation is also self-contained in two directories and its is limited to 5,824 lines of C++ code.

**Transparency.** WebRR’s instrumentation does not guarantee transparency. Instead, we designed WebRR to be tamper-proof, such that a web application cannot disable our instrumentation even if it can detect it (we assume an uncompromised browser, as discussed in our threat model section 4.3). Notice that if a malicious web application decides not to behave maliciously because WebRR’s presence has been detected, this will play in the defender’s favor.

**Limitations.** Currently WebRR only supports the main methods used to register callbacks from JS code (e.g., `setTimeout`, `setInterval`, `requestIdleCallback`, and `requestAnimationFrame`). In practice, there are a few other methods that can register callbacks, such as the Intersection Observer API [119]. Due to engineering time constraints, WebRR currently does not support replaying these callbacks. However, these rarely used callback registration APIs can easily be supported with additional engineering effort using the approach used for the supported registration techniques. Unfortunately, due to this limitation, we were unable to replay some highly optimized websites such as Facebook and Amazon.

WebRR has limited capability of replaying the portions of driveBy attacks after the exploitation of the browser occurs. Specifically, WebRR can replay up until the point in which the browser itself was compromised. At this point, a sophisticated attacker could potentially thwart WebRR’s replay by tampering with the recording hooks. Despite only being able to replay up until the point of exploitation, this information is still useful for a analyst to understand how the payload was deployed. Additionally, there are cases where
WEBRR can replay the entire driveBy attack, including the post-exploitation phase, as demonstrated in our evaluation (subsection 4.5.2). However, we believe the success of replaying driveBy attacks after the point of compromise will be on a case-by-case basis.

4.7 Related Work

**Attack Investigations.** The recent increase in enterprise data breaches has led to a significant amount of novel forensic analysis systems [16, 18, 19, 12, 10, 11, 15, 22, 14, 23, 24, 13, 31, 34, 29, 24, 29, 28, 21, 27, 30, 25, 33, 26]. Protracer [12] provides fine-grained provenance information about a system’s execution history, and MPI [14] improve the accuracy of provenance information by leveraging source-code annotations for execution partitioning. Unfortunately, these systems are limited in investigating web-based attacks due to the semantic-gap between system- and web-based semantics. OmegaLog [24] attempts to address this issue by correlating application-level logs to system-level audit logs. However, OmegaLog’s correlation technique currently does not support the browser. UIScope [23] attempts to provide additional semantic information by correlating GUI-information to system-level audit logs. Unfortunately, UIScope can only provided textual information and GUI metadata about the UI. In contrast, WEBRR allows the investigator to render exactly what was observed. This is extremely important in the web since styling and layout play a critical role in how the page is rendered.

To overcome the semantic gap, researchers have proposed systems that emit web-based logs [35, 37, 92]. For example, JSGraph [35] emits audit logs of a user’s browsing session for postmortem analysis. However, these systems are record-only systems and lack the capability of replaying the web attack, which limits these systems to only static analysis. WEBRR addresses these limitations by providing a deterministic replay that allows analysts to faithfully replay attacks, enabling a accurate, interactive and fine-grained investigation.

**Record & Replay.** There are several record and replay systems developed for debug-
For example, Jalangi [52] supports a record and replay mode that allows developers to attach debugging tools during the replay. Unfortunately, these systems were not developed with forensic analysis in mind and consequently, they cannot support the necessary properties that are required for a forensic analysis system (discussed in section 4.1). To overcome this limitation, researchers have proposed several record and replay systems that are designed for forensic analysis. For example, RAIN [10] is a system-level analysis systems that support record and replay of sophisticated APT attacks. However, a major shortcoming of these systems is that they are still limited in terms of providing a deterministic replay of the browser because they treat the browser [88, 10, 11, 89, 42] as a black box, which makes it challenging to ensure a deterministic replay due to the high parallelization and multi-process architecture of the browser. WEBRR overcomes this limitation by taking advantage of the single-threaded nature of JavaScript to avoid the need to synchronize threads during the replay.

### 4.8 Conclusion

In this work, we presented WEBRR, a novel forensic system for replaying and investigating web-based attacks in the modern web. We showed that WEBRR can replay a diverse set of web-based attacks including attacks that could not be replayed with prior state-of-the-art systems. Finally, WEBRR’s runtime performance is practical and only has a 3.14% increase on the page load time on websites in the Tranco 1k.
CHAPTER 5
CONCLUSION

In this dissertation, I have discussed the gaps that exist within the postmortem auditing space related to web-based attacks. I then proposed MNEMOSYNE, a web-based auditing framework that is capable of reconstructing watering hole attacks. I then showed that MNEMOSYNE was capable of greatly reducing the workload required by the forensic analyst to fully recover who was the victim of the attack. Next, I proposed WEBRR, a novel forensic system for replaying and investigating web-based attacks in the modern web. I then demonstrated that WEBRR can replay a diverse set of web-based attacks including attacks that could not be replayed with prior state-of-the-art systems. Finally, I showed that WEBRR’s runtime performance is very low and only has a 3.14% median increase on the page load time on websites in the Tranco 1k list.
REFERENCES


[54] Blink - the chromium projects, 2022.


[65] How was the attack on the pfsa and polish banks carried out, and who else was targeted by criminals? https://niebezpiecznik.pl/post/jak-przeprowadzono-atak-na-knf-i-polskie-banki-oraz-kto-jeszcze-byl-na-celowniku-przestepcow/.


