WEBRR: A Forensic System for Replaying and Investigating Web-Based Attacks in The Modern Web

Joey Allen[⋆§], Zheng Yang[†], Feng Xiao[†], Matthew Landen[†], Roberto Perdisci[†‡], Wenke Lee[†]
[⋆]*Palo Alto Networks* [†]*Georgia Institute of Technology* [‡]*University of Georgia*

Abstract

After a sophisticated attack or data breach occurs at an organization, a postmortem forensic analysis must be conducted to reconstruct and understand the root causes of the attack. Unfortunately, the majority of proposed forensic analysis systems rely on system-level auditing, making it difficult to reconstruct and investigate web-based attacks, due to the semantic-gap between system- and web-level semantics. This limited visibility into web-based attacks has recently become increasingly concerning because web-based attacks are commonly employed by nation-state adversaries to penetrate and achieve the initial compromise of an enterprise network. To enable forensic analysts to replay and investigate web-based attacks, we propose WEBRR, a novel OS- and device- independent record-and-replay (RR) forensic auditing system for Chromium-based web browsers. While there exist prior works that focus on web-based auditing, current systems are either *record-only* or suffer from critical limitations that prevent them from deterministically replaying attacks. WEBRR addresses these limitation by introducing a novel design that allows it to record and *deterministically* replay modern web applications by leveraging *JavaScript Execution Unit Partitioning*.

Our evaluation demonstrates that WEBRR is capable of replaying web-based attacks that fail to replay on prior state-of-the-art systems. Furthermore, we demonstrate that WEBRR can replay highly-dynamic modern websites in a deterministic fashion with an average runtime overhead of only 3.44%.

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 [1]. To address this growing need, researchers have developed system-level state-of-the-art auditing systems [2–12], including whole-system record-and-replay approaches [4, 5, 13–15]. 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 [16, 17] and security researchers have begun studying web-specific auditing systems [18–20]. However, these systems (e.g., [18, 19]) 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 [8, 19, 20], 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 [21–25]. Unfortunately, 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 [21], can lead to a performance overhead of more than 4x (as shown in §5.4). Additionally, Jalangi [24] and Mugshot [22] rely on instrumenting the JavaScript (JS) code of web applications to

[§]Joey Allen was affiliated with Georgia Institute of Technology when this research was completed.

System	Design Goal	Instrumentation Layer	Deterministic Replay	Always On	Tamper Proof	Portable
RR [21]	Debugging	OS	✓	✗	✓	✗
Mugshot [22]	Debugging	JavaScript	✓	✓	✗	✓
Jalangi [24]	Debugging	JavaScript	✓	✗	✗	✓
WebCapsule [26]	Forensics	Browser	✗	✓	✓	✓
WebRR	Forensics	Browser	✓	✓	✓	✓

Table 1: A comparison of WEBRR with existing record and replay systems. It covers three main aspects: the forensic properties that existing systems lack, the instrumentation layer where these systems have been implemented, and the design objectives of these systems.

support record and replay. Unfortunately, instrumenting 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 [26], 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, [26], Section 5.4), WebCapsule is unable to replay a number of real-world attacks, as we demonstrate in our evaluation (§5.3).

In order to highlight the advantages of WEBRR over prior work more clearly, we have included a reference table, Tab. 1. This table lays out the limitations of the prior work most similar to WEBRR, detailing which forensic properties they lack. To address these fundamental limitations, 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 variety of sophisticated web-based attacks ([27–30]). 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 (§ 5.2), while also showing that prior approaches fail to correctly replay these attacks (§ 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 [31]. 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.

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.

2.1 Render Process Architecture

Most of WEBRR is implemented via render process instrumentations, which allow us to extend Chrome's DevTools [32, 33] with a custom *forensic* Inspector Agent [34]. 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 [35]. The platform layer is an additional middle 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 §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).

```
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 1: deliver.js

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 [27–30].

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 1, is inspired by how modern browser exploitation frameworks like BeEF [36] 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 [37–39]. 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 [26] fails to replay the payload delivered by `deliver.js` due to uncaptured nondeterminism 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.

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 executed. 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 nondeterminism 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 [26]. 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 [2–6, 8–12, 18, 19, 26, 40–46]). 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.

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 [32]. 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 execution into individual units using JS execution unit partitioning (discussed in §4.1). Next, during the replay we sequentially replay each JEU in a synchronous manner while also preserving the recorded execution order (discussed in §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 §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 §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 (JEU) 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` using a similar approach.

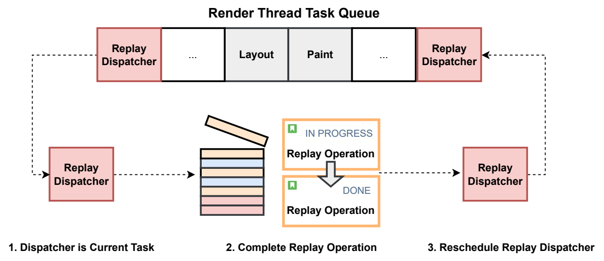

Figure 1: An overview of WebRR's Replay Dispatcher running on the render thread.

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

4.2 Replaying JEUs

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 JEUs 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 JEUs recorded by the JEU Recorder are stored in the *replay queue*. The queue is a FIFO queue and 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 §4.2.1). 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 Fig. 1. First, when the replay begins, WebRR schedules the replay dispatcher by posting it as a task on the render thread's task

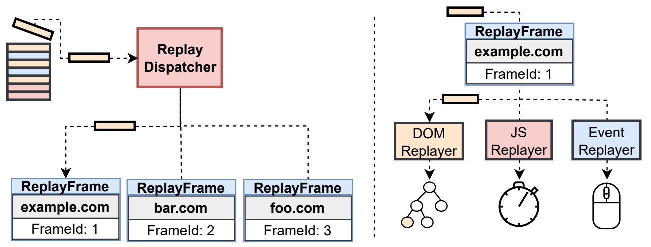

Figure 2: A high-level overview of the workflow that occurs when the replay dispatcher is executed on the render thread.

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 §4.2.1). 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.

4.2.1 Replay Operations

Internally, each operation is a map data structure that contains the necessary information for the replay engine to process the operation. In Fig. 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.

4.2.2 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 execute 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 that are in the ready state 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` using a similar approach.

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.

4.2.3 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.).

4.2.4 JS Microtasks & Asynchronous Functions

The final type of operation that WEBRR needs to address is the execution of JavaScript microtasks. Microtasks in JavaScript (e.g., Promises, queueMicrotask, try/catch, async/await) are brief, non-blocking tasks designed to run immediately after the completion of the current task (JEU) and before deferring to the event loop (for a more detailed explanation, interested readers may refer to [47]). For instance, when a promise is resolved, its *.then()* or *.catch()* handlers are added as microtasks. Furthermore, when a *async* function is invoked, it returns a promise. Any operations following the *await* keyword within the asynchronous function will be performed as a microtask.

WEBRR manages microtasks by allowing them to execute without interference. Specifically, WEBRR only schedules JEUs (scripts, callbacks, events) with its replay scheduler, without inhibiting or disrupting the execution of microtasks. This strategy is based on the principle that microtasks will invariably be performed after the JEU that scheduled them. In essence, during replay, a new JEU will not execute until the previous JEU and all of its associated microtasks have been completed. The correctness of this approach has been verified and validated in §5.6 of our evaluation.

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 *DOM-Replayer* 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 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.

4.4.1 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 response, WEBRR records the response's HTTP status, headers, and payload.

4.4.2 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 *Performance* module are dependent on the current execution speed, which differs across execution. Additionally, we consider APIs related to storage and cookies to be non-deterministic. The full list of non-deterministic APIs WEBRR records are provided at [48]. This list was developed by comparing the return values of several thousand web API invocations (222,118 to be specific) during our extensive experimental evaluations.

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 §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 [35]. 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 [49] and web APIs [50]. 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.

4.4.3 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.

4.4.4 Workers & Frame Communication

As previously discussed in §2, JS can run script operations on background threads using the Web Worker's Interface [51]. 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 main-frame) 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.

4.5 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 [52]). 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 we strategically designed WEBRR's recording engine to be stateless, so that its recording hooks could be called on different threads. This allows the the APIs and JEU Recorder to be able to record the SW and render threat simultaneously.

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?

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

5.1.1 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 non-determinism 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 [53] (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 (*rec*, *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 [54]. Specifically, we use SimCLRv2 [55], 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 similarlity among web pages, using a dataset of 7,203 web page screenshots collected by us from the Tranco 1K [31] list of websites. During the evaluation, we embed the corresponding recorded and replayed screenshot pairs (*rec*, *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.

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 Tab. 3. 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 [56], 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 §6 for a discussion of related limitations). The first attack (attack 1) is a phishing-based attack that was collected from OpenPhish [57]. The second attack (attack 2) is a credential harvesting attack inspired by a real-world attack carried out by the OceanLotus APT group [58].

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.

OS	Website	Rec-JEU Seq. Len.	Rep-JEU Seq. Len.	JEU-Seq. E. D.	Rec-API Seq. Len.	Rep-API Seq. Len.	API Seq. E. D.	Replayed
Lin.	stackoverflow	134	134	0	12,868	12,865	13	✓
Lin.	wikipedia	181	181	0	52,700	52,700	4	✓
Lin.	whitehouse	60	60	0	6,148	6,148	4	✓
Lin.	mozilla	141	141	0	8,078	8,078	0	✓
Lin.	craigslist	345	345	0	26,534	26,536	8	✓
Win.	stackoverflow	146	146	0	14,477	14,477	11	✓
Win.	wikipedia	173	173	0	6,790	6,794	14	✓
Win.	whitehouse	39	39	0	5,878	5,878	5	✓
Win.	mozilla	91	91	0	4,557	4,565	12	✓
Win.	craigslist	52	52	0	4,546	4,546	10	✓
Andr.	stackoverflow	161	161	0	15,391	15397	18	✓
Andr.	wikipedia	69	69	0	26,790	26,790	12	✓
Andr.	whitehouse	32	32	0	5,909	5,911	13	✓
Andr.	mozilla	67	67	0	1,769	1,769	0	✓
Andr.	craigslist	58	58	0	6,820	6,822	0	✓

Table 2: 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.), and replayed correctly (Replayed).

Popular Websites Used. We used the benign websites in the *Website* column in Tab. 2. The diverse set of websites we selected were all found on the Tranco 1k list [31]. We 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.

5.2.1 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 Tab. 3 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 Tab. 2. 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 the same values. These results are shown in the *API Sequence Edit Distance* column in Tab. 3 for the attacks and in Tab. 2 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 $\approx 0.1\%$ of cases) some APIs returned different values during replay. We found that for these scenarios, we returned the wrong node type [59] 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 §6.

OS	Attack	Rec-JEU Seq. Len.	Rep-JEU Seq. Len.	JEU-Seq. E. D.	Rec-API Seq. Len.	Rep-API Seq. Len.	API Seq. E. D.	Replayed
Lin.	Phish.	96	96	0	326	326	0	✓
Lin.	Cred. Harv.	977	977	0	3,733	3,733	0	✓
Lin.	KeyLogger	51	51	0	97	97	0	✓
Lin.	Click. Jack.	189	189	0	131	131	0	✓
Lin.	DriveBy	93	93	0	263	263	0	✓
Lin.	StealthyPush.	10	10	0	73	73	0	✓
Lin.	XXS-SW	2	2	0	10	10	0	✓
Win.	Phish.	96	96	0	326	326	0	✓
Win.	Cred. Harv.	15	15	0	1389	1389	0	✓
Win.	KeyLogger	55	55	0	61	61	0	✓
Win.	Click. Jack.	38	38	0	259	259	0	✓
Win.	DriveBy	298	298	0	2441	2441	0	✓
Win.	StealthyPush.	10	10	0	73	73	0	✓
Win.	XXS-SW	2	2	0	10	10	0	✓
Andr.	Phish.	55	55	0	5,127	5,127	0	✓
Andr.	Cred. Harv.	211	211	0	5,504	5,504	0	✓
Andr.	KeyLogger	92	92	0	80	80	0	✓
Andr.	Click. Jack.	32	32	0	266	266	0	✓
Andr.	DriveBy	353	353	0	2,571	2,571	0	✓
Andr.	StealthyPush.	10	10	0	73	73	0	✓
Andr.	XXS-SW	2	2	0	10	10	0	✓

Table 3: Web-Based Attack Results. See Tab. 2 for column descriptions.

5.2.2 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 §5.1.1. The number of screenshots pairs created for each attack and the average cosine similarity between the pairs is shown in Tab. 4. 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 deter-

		Screenshots	Avg. Cosine Similarity
Attacks	Phishing	13	1.00
	Credential Harvesting	23	1.00
	KeyLogger	16	1.00
	Clickjacking	12	1.00
Popular Websites	mozilla.org	39	1.00
	wikipedia.org	10	.99
	whitehouse.gov	10	1.00
	craigslist.org	14	1.00
	stackoverflow.com	22	.94

Table 4: The experimental results for visual determinism.

minism 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. However, we would like to emphasize that the screenshots taken still appeared extremely similar, despite the lower than normal accuracy (our visual similarity metric is highly sensitive to rendering changes). 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 average, 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.

5.3 Comparison with Existing Systems

In this section, we evaluated the performance of existing systems like WebCapsule [26], Mugshot [22], and RR [21] by their ability to replay the attacks used for assessing WEBRR. We considered two criteria to determine the success of a replay. Firstly, the visual aspects of the attack, for example, the social engineering components, had to be displayed accurately during the replay. Secondly, any user interactions, such as clicks or key inputs, with the visual components had to be replayed correctly.

WebCapsule. is a forensic-based record-and-replay system for Chromium-based browsers. Unfortunately, evaluating WebCapsule directly using the metrics discussed in § 5.1.1 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 §5.1.1.

Attack	WebRR	WC	Mugshot	RR
Keylogger	✓	✓	✓†	✓
Phishing	✓	–	✓†	✓
Clickjacking	✓	–	✓†	✓
Credential Harvesting	✓	–	✓†	✓
Phishing*	✓	✗	✓†	✓
Clickjacking*	✓	✗	✓†	✓
Credential Harvesting*	✓	✗	✓†	✓

Table 5: Evaluation Results of WEBRR, WebCapsule, Mugshot, and RR correctly replaying the web-based attacks. * means the attack was backported and – means the attack could not be recorded. † means the recording could be disabled during the attack.

To overcome this and perform the evaluation, we relied on the compiled binary version of WebCapsule provided by its authors [60]. Specifically, we used the binary version to answer the following question: *Can WebCapsule properly replay the web-based attacks described in Section 5.2?*

The results for these experiments are shown in Tab. 5. We found that WebCapsule was only able to replay one of the attacks, while WEBRR successfuly replayed them all. We discuss the results in the remainder of the section.

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 JS execution and giving the appearance that the 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 a 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 [58]. 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].

Mugshot. Mugshot [22] is a system designed for debugging that records a web app's execution on a user's device and enables a developer to replicate it on their own machine. Although Mugshot's source-code is not available, we discovered a system called Reanimator [61] that was inspired by Mugshot (stated in README). We then used Reanimator to replay the attacks, but initially, it failed to reproduce the clickjacking, credential harvesting, or phishing attacks accurately. This issue was due to its lack of support for recording local storage values and replaying callbacks registered via *Window.requestAnimationFrame* or *Window.requestIdleCallback*. To rectify this, we enhanced the tool to support replaying these features, adopting techniques described in the Mugshot paper. Following these adjustments, Mugshot was able to replay all the attack scenarios correctly, which was anticipated. However, when we factored in a scenario where the attacker attempted to disable the recording, we found that Mugshot was unable to replay any of the attacks. This is because since Mugshot's main design goal is debugging, it was not designed to be tamper proof. Specifically, Mugshot records and replays by inserting the "mugshot.js" script into the application and executes at the same privilege level as a potentially malicious web app itself. This makes it susceptible to tampering if an adversary is aware of its existence. They can easily interfere with the logs and override the recording hooks. We found, for instance, that simply setting the logging object to undefined could destroy the logs, and the hooks could be easily overwritten given their shared execution context with the web application.

RR. RR [21] is a system-level debugging utility tailored for Linux applications. As anticipated, it was able to replay all the tested attack scenarios on a Linux device. However, unlike WEBRR, RR lacks the capability to replay these attacks on both Windows and Android platforms. More so, RR's practicality in a real-world setting is limited due to its considerable effect on runtime performance. Specifically, it leads to over a four-fold (459%) increase in page load times while recording, an issue we elaborate on in §5.4.

5.4 Runtime Overhead

To assess the efficiency of our system, we measured the impact WEBRR has on the page load time for websites listed in the Tranco 1k [31]. This particular aspect is crucial to study because previous research indicates that slower page load times can lead to user dissatisfaction, ultimately resulting in reduced revenue for websites [62]. To carry out this experiment, we calculated the page load time by finding the time difference between when the navigation starts (`navigationStart` event) and when the page is fully loaded (`loadEventEnd` event). To minimize the influence of network conditions on our study, we conducted our tests in a simulated network environment with a latency of 100 milliseconds and a download speed of 15Mbps (similar to prior approaches [63, 64]). Next, we visited the landing page of each website ten times, using both WEBRR and a standard version of Chromium 83.0.4103.97. After performing these tests, we averaged the load times of the ten visits. This gave us a reliable measure of how the use of WEBRR impacts the time it takes for a page to load. Finally, we ran a similar experiment using RR [21] to calculate its impact on the page load time.

Our experiments found that WEBRR's recording engine impact on the page load time is extremely low with a median increase of only 2.94% which results in a 0.037s increase of the time. When looking at the outliers, we found that websites that had extremely large HTML documents were more impacted by WEBRR's recording hooks, because we record every DOM insertion made by the parser. For example, the website *hxxp://www.wp.pl* is an outlier and has 10.54% overhead. This is because the website's homepage requires WEBRR 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, WEBRR's runtime overhead was less than 9.56%, which shows that WEBRR's recording is capable of being deployed in real-world enterprise settings.

We have also completed the same experiment for RR to compare its impact on page load times while recording. We found that RR had a median increase in page load overhead for the website by 402%. This significant increase is somewhat expected since RR's main design goal is not forensics, but instead debugging. Since it does not need to maintain the *always-on* property, RR can constrain each thread in a process to a single core (refer to Section 7 in [21]). This allows it to achieve a deterministic replay, but it significantly

[1]Phishing Experiment: `https://youtu.be/JGddenliISs`
[2]Credential Harvesting Experiment: `https://youtu.be/7yxKcbhBqeQ`

List of Websites Used in Storage Evaluation
cnn.com, yahoo.com, theguardian.com, ebay.com, yelp.com, irs.gov, wikipedia.org, washingtonpost.com, cbs.com, nytimes.com

Table 6: The 10 websites used for the storage overhead evaluation (§5.5) of WEBRR.

Run	Website	Rec-JEU Seq. Len.	Rep-JEU Seq. Len.	JEU-Seq. E. D.	Rec-API Seq. Len.	Rep-API Seq. Len.	API Seq. E. D.	Replayed
1	microtest-eval	1279	1279	0	42157	42157	0	✓
2	microtest-eval	1553	1553	0	52457	52457	0	✓
3	microtest-eval	1039	1039	0	35017	35017	0	✓
4	microtest-eval	1781	1781	0	59142	59142	0	✓
5	microtest-eval	1333	1333	0	44277	44277	0	✓

Table 7: Evaluation of replayed microtasks website. See Tab. 2 for column descriptions.

hampers performance, particularly for applications that are heavily reliant on multi-threading or multi-process apps, such as Chromium. Therefore, RR is unfit for forensic settings as it fails to maintain the crucial forensic property of being *always-on* in a real-world environment.

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 Tab. 6. 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.

5.6 Asynchronous Function Execution

In these experiments, our goal was to validate the approach of our system for managing asynchronous functions and microtasks (this is elaborated on in §4.2.4). We aimed to confirm that the microtasks were executed at the same time during the playback as they were during the recording. For this, we created a website designed to run a large number of callbacks (the exact number for each recording is provided in the "Rec-JEU Seq. Len" column of Tab. 7) in a short-period of time. When activated, these callbacks would trigger anywhere from 1 to 100 microtasks using methods such as Promises, the async/await keywords, and the queueMicrotask function. This led to several thousand microtasks being executed in a brief time span. Next, we recorded and replayed the website five times to check if the execution order of callbacks and microtasks during replay was consistent with the original recording. We replayed it multiple times because we were executing numerous callbacks quickly using various registration techniques, such as setTimeout, registerIdleCallback, and requestAnimationFrame. As a result, the order in which these callbacks were invoked varied across runs.

To verify the correctness of the replay, we employed the method detailed in the Metrics section (§5.1.1) and compared the recorded and replayed sequences of JEUs. The findings are shown in Tab. 7. In every test, we observed that the edit distance between JEUs (column JEU-Seq. E.D.) and APIs (column API Seq. E.D.) sequences was zero. This indicates that all JS functions and APIs calls happened in the exact same order during both the recording and the replay. Our experiments highlight our system's ability to accurately replay microtasks in the right sequence. Furthermore, we analyzed the benign websites from our study and found that all of them, barring Wikipedia, employed numerous promises and asynchronous functions. All these websites replayed correctly based on our experiments.

Version	Ported Version Release Date	Functions Modified
83 ->84	Jul 14, 2020	0
84 ->85	Aug 25, 2020	12
85 ->86	Oct 6, 2020	10
86 ->87	Nov 17, 2020	13

Table 8: WEBRR's maintainability evaluation results.

5.7 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). The implementation of WEBRR is also self-contained in two directories, limited to 5,824 lines of C++ code, and 44 functions were hooked in the Chromium code base to support its functionality.

To gauge its maintainability, we checked how often the code with WEBRR's hooks was updated across different versions. We did this by comparing consecutive versions to see how many functions with WEBRR's hooks had been modified. We check for changes by diffing the files where we had made modifications and then manually verifying if these changes were located within the functions where our hooks were. Next, if we identified a change inside the functions where our hooks were located, we investigated if this change would impact our hooks. We then repeated this experiment 4 times (for 4 different code versions) and measured how often changes impacted hooked functions. The results are reported in Tab. 8. We found that among the functions where WEBRR's hooks were located, the median number that underwent modifications was 11. Our investigation revealed that the modifications had minimal or no effect on WEBRR's hooks. For instance, in patch version 85, the *force_xhtml* input variable was eliminated from the *LocalDOMWindow::InstallNewDocument* method. While WEBRR's hook used to record this value when new documents were created, its removal means we need to update this hook. For the versions we analyzed, the process of verifying changes took around 2-3 hours, and we estimate that porting WEBRR to another version would require a single developer approximately a few days to complete (including code changes and testing/debugging). Obviously, future versions that include large structural changes may require a more involved refactoring to port WEBRR. However, this would be expected as part of the overall Chromium and DevTools development process.

6 Discussion & Limitations

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 §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.

Correctness. We studied WEBRR's limitations by examining 10 popular websites that posed challenges during replay. For each of these websites, we conducted an in-depth manual analysis to understand why they did not replay correctly. From our investigation, we identified three primary issues: unsupported callbacks, problems with message channels that lead to replay discrepancies, and unsupported components.

First, we observed that WEBRR naturally struggles to accurately replay websites that use callback registration methods that it does not yet support. For instance, when Reddit's page loads, it forms a custom root HTML element using the CustomElementRegistry.define method. This method requires a constructor callback. Because WEBRR does not yet support the replay of this particular method of callback registration, the page does not load correctly. Additionally, WEBRR currently fails to replay *Forbes* because it does not yet support replaying MutationObservers. However, we would like to point out that, with additional engineering effort, WEBRR can be extended to replay additional callbacks using the same hooking techniques used for the callbacks WEBRR currently supports.

The next limitation that we found with WEBRR is due to *MessageChannels*. MessageChannels[3] are objects that allow for communication for scripts running in different browsing contexts. Each channel has two attributes *MessageChannel.port1* & *MessageChannel.port2*. The app that creates the channel uses *port1* and passes *port2* to another window using *window.postMessage*. The receiver of *port2* can then receive messages through this channel and send messages through the channel using *MessagePort.postMessage*. Currently, WEBRR has a limitation, where it can only replay messages to *port2*, but it cannot replay messages that are sent back to the creating context and received on *port1*. This is because during the replay WEBRR will reconstruct the MessageEvent, and if it discovers that a MessagePort was passed during the recording, it will create a *MessagePort* object and store it into the *MessageEvent* and then fire this MessageEvent into the receiving context. This allows us to replay messages that were passed from the creating context. However, since we recreate this MessagePort in the replay engine, it is not associated with a *MessageChannel* object. Therefore, we cannot send a message back to the creating context. To summarize, WEBRR's current approach is limited to unidirectional communication for MessageChannels, where it can only replay messages that are sent from the creating context to the external context. Unfortunately, this issue causes us to fail to replay *google, zillow, amazon* and *yahoo*. Overall, we believe this issue can be addressed by instead of reconstructing message events, to instead intercept them on the fly and then replay the created message event. By intercepting the message events, it would allow us to replay the message event in the correct order, but also allow the message port to be attached to the *MessageChannel* created in the original context.

The final issue we found was related to unsupported components, such as WASM, WebRTC, SVG, and IndexedDB. For example, WEBRR failed to replay *twitch* and *youtube*, because it does not currently support WebRTC. Additionally, WEBRR currently does not support IndexedDB events, which prevents it from replaying *Salesforce* & *MSN*. However, we did observe that WEBRR can replay WebGL, which relies on *animation* callbacks, which are supported by our current WEBRR release. Finally, we believe that these components could be replayed using similar design techniques that WEBRR uses. For example, since websites instantiate and invoke WASM through JS and WEBRR treats V8 as a blackbox, it would be fairly straightforward to support replaying WASM modules with additional engineering effort.

To conclude, we bound the completeness of WEBRR to the components that are discussed in detail in the paper, which include key pieces of the rendering engine, service workers, and replaying JS that is executed via scripts and the supported

[3]https://developer.mozilla.org/en-US/docs/Web/API/MessageChannel

callbacks that are discussed (i.e., event handlers, timeouts, idle callbacks, and animation callbacks). Despite these limitations, we would like to point out that WEBRR was able to replay several real-world complex benign websites and real-world attacks, as demonstrated in § 5.2 on Linux, Android, and Windows. With the kind of resources available to the Chromium Devtools development team, these limitations could be addressed in a reasonably small time through additional engineering effort.

7 Related Work

Attack Investigations. The recent increase in enterprise data breaches has led to a significant amount of novel forensic analysis systems [2–6, 8–12, 40, 43]. Protracer [3] provides fine-grained provenance information about a system's execution history, and MPI [40] 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 [41] 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 [8] 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 [18–20]. For example, JSGraph [19] 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 debugging and testing [21–25, 65–67]. For example, Jalangi [24] 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 § 1). To overcome this limitation, researchers have proposed several record and replay systems that are designed for forensic analysis. For example, RAIN [4] 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 [4, 5, 13–15] 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.

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 3.44% increase on the page load time on websites in the Tranco 1k.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd for their helpful and informative feedback. This material was supported in part by National Science Foundation (NSF) under grants No. CNS-2126641; the Office of Naval Research (ONR) under grants N00014-17-1-2179, N00014-17-1-2895, N00014-15-1-2162, and N00014-18-1-2662; and Cisco Systems under an unrestricted gift. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflects the views of NSF or ONR.

References

[1] Tarun Yadav and Arvind Mallari Rao. "Technical Aspects of Cyber Kill Chain". In: *CoRR* abs/1606.03184 (2016). arXiv: 1606.03184. URL: http://arxiv.org/abs/1606.03184.

[2] Samuel T King and Peter M Chen. "Backtracking intrusions". In: *ACM Transactions on Computer Systems (TOCS)* 23.1 (2005), pp. 51–76.

[3] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. "ProTracer: towards practical provenance tracing by alternating between logging and tainting". In: *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, Feb. 2016.

[4] "RAIN: Refinable Attack Investigation with On-demand Inter-Process Information Flow Tracking". In: *Proceedings of the 24rd ACM Conference on Computer and Communications Security (CCS)*. Dallas, Texas, Oct. 2017.

[5] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1705–1722.

[6] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. "High Accuracy Attack Provenance via Binary-based Execution Partition". In: *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, Feb. 2013.

[7] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F Ciocarlie, et al. "MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation." In: *NDSS*. 2018.

[8] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. "UISCOPE: Accurate, Instrumentation-free, and Visible Attack Investigation for GUI Applications". In: (2020).

[9] Sadegh Milajerdi, Rigel Gjomemo, Birhan Eshete, R. Sekar, and V.N Venkatakrishnan. "HOLMES: Real-time APT Detection through Correlation of Suspicious Information Flows." In: *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA, May 2019.

[10] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. "Towards a Timely Causality Analysis for Enterprise Security." In: *NDSS*. 2018.

[11] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, C Gunter, et al. "You are what you do: Hunting stealthy malware via data provenance analysis". In: *Symposium on Network and Distributed System Security (NDSS)*. 2020.

[12] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. "NODOZE: Combatting Threat Alert Fatigue with Automated Provenance Triage". In: *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, Feb. 2019.

[13] Yonghwi Kwon, Weihang Wang, Jinho Jung, Kyu Hyung Lee, and Roberto Perdisci. "C 2 SR: Cybercrime Scene Reconstruction for Post-mortem Forensic Analysis". In: *Network and Distributed Systems Security (NDSS) Symposium 2021*. 2021.

[14] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. "Eidetic systems". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, Oct. 2014.

[15] Yang Ji, Sangho Lee, and Wenke Lee. "RecProv: Towards Provenance-Aware User Space Record and Replay". In: *Proceedings of the 5th International Provenance and Annotation Workshop (IPAW)*. Mclean, VA, 2016.

[16] *Talon Browser*. https://talon-sec.com/. 2022.

[17] *Island Browser*. https://www.island.io/. 2022.

[18] Phani Vadrevu, Jienan Liu, Bo Li, Babak Rahbarinia, Kyu Hyung Lee, and Roberto Perdisci. "Enabling Reconstruction of Attacks on Users via Efficient Browsing Snapshots." In: *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, Feb. 2017.

[19] Bo Li, Phani Vadrevu, Kyu Hyung Lee, Roberto Perdisci, Jienan Liu, Babak Rahbarinia, Kang Li, and Manos Antonakakis. "JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions." In: *NDSS*. 2018.

[20] Joey Allen, Zheng Yang, Matthew Landen, Raghav Bhat, Harsh Grover, Andrew Chang, Yang Ji, Roberto Perdisci, and Wenke Lee. "Mnemosyne: An Effective and Efficient Postmortem Watering Hole Attack Investigation System". In: *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, Oct. 2020, pp. 787–802. ISBN: 9781450370899. DOI: 10.1145/3372297.3423355.

[21] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. "Engineering Record and Replay for Deployability." In: *USENIX Annual Technical Conference*. 2017, pp. 377–389.

[22] James W Mickens, Jeremy Elson, and Jon Howell. "Mugshot: Deterministic Capture and Replay for JavaScript Applications." In: *NSDI*. Vol. 10. 2010, pp. 159–174.

[23] Brian Burg, Richard Bailey, Amy J Ko, and Michael D Ernst. "Interactive record/replay for web application debugging". In: *Proceedings of the 26th annual ACM symposium on User interface software and technology*. 2013, pp. 473–484.

[24] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. "Jalangi: a selective record-replay and dynamic analysis framework for JavaScript". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 2013.

[25] John Vilk, Emery D Berger, James Mickens, and Mark Marron. “McFly: Time-Travel Debugging for the Web”. In: *arXiv preprint arXiv:1810.11865* (2018).

[26] Christopher Neasbitt, Bo Li, Roberto Perdisci, Long Lu, Kapil Singh, and Kang Li. “Webcapsule: Towards a lightweight forensic engine for web browsers”. In: *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, Colorado, Oct. 2015.

[27] Panagiotis Papadopoulos, Panagiotis Ilia, Michalis Polychronakis, Evangelos P Markatos, Sotiris Ioannidis, and Giorgos Vasiliadis. “Master of web puppets: Abusing web browsers for persistent and stealthy computation”. In: *arXiv preprint arXiv:1810.00464* (2018).

[28] Karthika Subramani, Jordan Jueckstock, Alexandros Kapravelos, and Roberto Perdisci. “Categorizing Service Worker Attacks and Mitigations”. In: *arXiv preprint arXiv:2111.07153* (2021).

[29] Soroush Karami, Panagiotis Ilia, and Jason Polakis. “Awakening the Web’s Sleeper Agents: Misusing Service Workers for Privacy Leakage”. In: *Network and Distributed System Security Symposium*. 2021.

[30] Phakpoom Chinprutthiwong, Raj Vardhan, GuangLiang Yang, and Guofei Gu. “Security Study of Service Worker Cross-Site Scripting.” In: *Annual Computer Security Applications Conference*. 2020, pp. 643–654.

[31] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. “Tranco: A research-oriented top sites ranking hardened against manipulation”. In: *arXiv preprint arXiv:1806.01156* (2018).

[32] *Chrome DevTools*. https://developer.chrome.com/docs/devtools/. 2022.

[33] *Chrome DevTools Protocol*. [Online; accessed 20-January-2022]. 2022. URL: https://chromedevtools.github.io/devtools-protocol/.

[34] *Contributing to Chrome DevTools Protocol*. ContributingtoChromeDevToolsProtocol. 2022.

[35] *IDLCompiler*. https://source.chromium.org/chromium/chromium/src/+/master:third_party/blink/renderer/bindings/IDLCompiler.md. 2022.

[36] *The Browser Exploitation Framework*. https://beefproject.com/. 2023.

[37] Secureworks. *Cobalt Illusion*. https://www.secureworks.com/research/threat-profiles/cobalt-illusion. 2022.

[38] Yuri Ilyin. *Testing the mettle: legit tools for illicit cyberespionage*. https://www.kaspersky.co.za/blog/testing-the-mettle/15140/.

[39] Catalin Cimpanu. *Chinese Cyber-Espionage Group Hacked Government Data Center*. https://www.bleepingcomputer.com/author/catalin-cimpanu.

[40] Shiqing Ma, Juan Zhai, Fei Wang, Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. “MPI: Multiple Perspective Attack Investigation with Semantic Aware Execution Partitioning”. In: *Proceedings of the 25th USENIX Security Symposium (Security)*. Vancouver, BC, Canada, Aug. 2017.

[41] Wajih Ul Hassan, Mohammad A Noureddine, Pubali Datta, and Adam Bates. “OmegaLog: High-fidelity attack investigation via transparent multi-layer log analysis”. In: *Proc. NDSS*. 2020.

[42] Ashish Gehani and Dawood Tariq. “SPADE: support for provenance auditing in distributed environments”. In: *Proceedings of the 13th International Middleware Conference (Middleware)*. 2012.

[43] Wajih Ul Hassan, Adam Bates, and Daniel Marino. “Tactical Provenance Analysis for Endpoint Detection and Response Systems”. In: *Proceedings of the IEEE Symposium on Security and Privacy*. 2020.

[44] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. “UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats”. In: *Network and Distributed System Security Symposium*. 2020.

[45] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. “SLEUTH: Real-time attack scenario reconstruction from COTS audit data”. In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 487–504.

[46] Md Nahid Hossain, Sanaz Sheikhi, and R Sekar. “Combating Dependence Explosion in Forensic Analysis Using Alternative Tag Propagation Semantics”. In: *IEEE S&P*. 2020.

[47] MDN contributors. *In depth: Microtakss and the JavaScript runtime envrionment*. https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide/In_depth. 2023.

[48] https://github.com/jallen89/JSCapsule/blob/dev/forensic_utils/parser.py.

[49] *attributes.cc.tmpl*. third_party/blink/renderer/bindings/templates/attributes.cc.tmpl. 2022.

[50] *methods.cc.tmpl*. third_party/blink/renderer/bindings/templates/methods.cc.tmpl. 2022.

[51] "MDN Contributors". *"Web Workers API"*. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API. 2022.

[52] MDN Contributors. *Using Service Workers*. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers. 2022.

[53] *Levenshtein Distance*. https://en.wikipedia.org/wiki/Levenshtein_distance. 2022.

[54] Sahar Abdelnabi, Katharina Krombholz, and Mario Fritz. "VisualPhishNet: Zero-day phishing website detection by visual similarity". In: *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 2020, pp. 1681–1698.

[55] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey Hinton. "Big Self-Supervised Models are Strong Semi-Supervised Learners". In: *arXiv preprint arXiv:2006.10029* (2020).

[56] Verizon. *Data Breach Digest*. https://www.verizon.com/business/resources/reports/data-breach-digest-2017-perspective-is-reality.pdfx. 2017.

[57] *OpenPhish*. https://openphish.com/phishing_database.html. 2022.

[58] Dave Lassalle, Sean Koessel, and Steven Abair. *OceanLotus Blossoms: Mass Digital Surveillance and Attacks Targeting ASEAN, Asian Nations, the Media, Human Rights Groups, and Civil Society*. https://www.volexity.com/blog/2017/11/06/oceanlotus-blossoms-mass-digital-surveillance-and-exploitation-of-asean-nations-the-media-human-rights-and-civil-society/. Nov. 2017.

[59] *Node.nodeType*. https://developer.mozilla.org/en-US/docs/Web/API/Node/nodeType. 2022.

[60] *WebCapsule Github*. https://github.com/perdisci/WebCapsule. 2015.

[61] Lon Ingram. *reanimator*. https://github.com/WaterfallEngineering/reanimator.

[62] Sebastian Egger, Peter Reichl, Tobias Hoßfeld, and Raimund Schatz. ""Time is bandwidth"? Narrowing the gap between subjective time perception and Quality of Experience". In: *2012 IEEE international conference on communications (ICC)*. IEEE. 2012, pp. 1325–1330.

[63] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. "Adgraph: A graph-based approach to ad and tracker blocking". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 763–776.

[64] Umar Iqbal, Charlie Wolfe, Charles Nguyen, Steven Englehardt, and Zubair Shafiq. "Khaleesi: Breaker of advertising and tracking request chains". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 2911–2928.

[65] Oren Laadan, Nicolas Viennot, and Jason Nieh. "Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems". In: *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '10. New York, NY, 2010.

[66] Sudarshan M Srinivasan, Srikanth Kandula, Christopher R Andrews, and Yuanyuan Zhou. "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging". In: *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*. Boston, MA, 2004.

[67] *Replay: Your time travel debugger*. 2022. URL: https://www.replay.io/.