WebCapsule: Towards a Lightweight Forensic Engine for Web Browsers

Christopher Neasbitt[†], Bo Li[†], Roberto Perdisci[†ϟ], Long Lu[‡], Kapil Singh[◦], and Kang Li[†]
[†]Department of Computer Science, University of Georgia
[ϟ]College of Computing, Georgia Tech
[‡]Department of Computer Science, Stony Brook University
[◦]IBM Research
{cjneasbi,lubao515}@uga.edu, perdisci@cs.uga.edu, long@cs.stonybrook.edu
kapil@us.ibm.com, kangli@cs.uga.edu

ABSTRACT

Performing detailed forensic analysis of real-world web security incidents targeting users, such as social engineering and phishing attacks, is a notoriously challenging and time-consuming task. To reconstruct web-based attacks, forensic analysts typically rely on browser cache files and system logs. However, cache files and logs provide only sparse information often lacking adequate detail to reconstruct a precise view of the incident. To address this problem, we need an *always-on* and *lightweight* (i.e., low overhead) forensic data collection system that can be easily integrated with a variety of popular browsers, and that allows for recording enough detailed information to enable a full reconstruction of web security incidents, including phishing attacks.

To this end, we propose *WebCapsule*, a novel *record and replay forensic engine* for web browsers. WebCapsule functions as an always-on system that aims to record all non-deterministic inputs to the core web rendering engine embedded in popular browsers, including all user interactions with the rendered web content, web traffic, and non-deterministic signals and events received from the runtime environment. At the same time, WebCapsule aims to be *lightweight* and introduce low overhead. In addition, given a previously recorded trace, WebCapsule allows a forensic analyst to fully replay and analyze past web browsing sessions in a controlled isolated environment. We design WebCapsule to also be *portable*, so that it can be integrated with minimal or no changes into a variety of popular web-rendering applications and platforms. To achieve this goal, we build WebCapsule as a self-contained instrumented version of Google's Blink rendering engine and its tightly coupled V8 JavaScript engine. We evaluate WebCapsule on numerous real-world phishing attack instances, and demonstrate that such attacks can be recorded and fully replayed. In addition, we show that WebCapsule can record complex browsing sessions on popular websites and different platforms (e.g., Linux and Android) while imposing reasonable overhead, thus making always-on recording practical.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/XXX.XXXXXXX.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

General Terms

Security; Forensics

Keywords

Forensic Engine; Web Security; Browsing Replay

1. INTRODUCTION

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

The state-of-the-art methods for reconstructing web-based incidents generally follow two approaches. The first approach relies on analyzing the web browser's history, cache files, and system logs [16, 20]. However, cache files and logs provide only sparse information often lacking adequate detail to reconstruct a precise view of what happened during social engineering and phishing attacks that may have occurred days in the past. The second approach leverages access to full network packet traces, which may provide some indications of how an incident unfolded. However, the complexity of modern web pages results in a large *semantic gap* between the web traffic and the detailed events (e.g., page rendering, mouse movements, key presses, etc.) that occurred within the browser [18]. Such semantic gaps make it very difficult to precisely reconstruct what a victim actually saw and how she was tricked, and to identify what information was consequently leaked.

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

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

with particular focus on *attacks that directly target users*, such as social engineering and phishing attacks.

In this paper we propose *WebCapsule*, a novel *record and replay forensic engine* for web browsers. WebCapsule lays the foundations for web-based attack reconstruction and analysis while meeting all of the above stated requirements. Our main goal is to enable an always-on, transparent, and fine-grained recording (and subsequent replay) of potentially harmful web browsing sessions. As depicted in Figure 1, WebCapsule aims to record all non-deterministic inputs to the core web rendering engine embedded in the browser, including all user interactions with the rendered web content, web traffic, and non-deterministic signals and events received from the runtime environment.

WebCapsule allows an analyst to later replay previously recorded browsing sessions in a separate controlled environment, where no new external user inputs or network transactions are needed. This enables detailed analysis of security incidents that are (obviously) unexpected, and allows for reconstructing detailed information about incidents that may follow new, never-before-seen attack patterns. In addition, by replaying all non-deterministic inputs, including all content provided by the server, WebCapsule enables a full forensic investigation of incidents involving *ephemeral web content*, such as short-lived phishing or social engineering attack pages.

While some previous work has studied record and replay to assist the debugging of web applications [2,5,23], these studies do not focus on forensic analysis and, more importantly, do not satisfy the associated requirements listed above. For example, TimeLapse [5] is a debugging tool based on Apple's WebKit [30] that allows for recording and replaying web content. However, TimeLapse does not work as an "always on" system. Also, TimeLapse does not allow for transparent recording because it deeply modifies the internals of WebKit, for example to force a synchronous scheduling of threads such as the HTML parser thread [25], thus also impacting performance. In addition, TimeLapse currently only works on MacOS+Safari+WebKit [26], and is not easily portable to other operating systems and browsers. Conversely, WebCapsule can function as an *always-on* system (e.g., it can be configured to start recording at browser startup with no user intervention) to continuously and *transparently* record browsing sessions while introducing low overhead. Furthermore, WebCapsule is highly portable, can be embedded in a variety of web-rendering applications, and can run on a variety of platforms. Furthermore, unlike [2, 23], WebCapsule is not limited to only recording user interactions with web pages, but instead aims to record all non-deterministic events needed to fully replay browsing traces, including all previously rendered web content, without incurring high performance overhead (we further discuss differences with previous work in Section 8). This is very important for forensic analysis. In fact, most malicious web pages (e.g., phishing websites) are *short lived*. Therefore, because [2, 23] only record user interactions with pages without recording all other non-deterministic inputs (e.g., network traces, timing information, etc.), they do not enable an after-the-fact replay and investigation of security incidents. WebCapsule solves this problem.

To make WebCapsule *portable*, so that it can be easily embedded in a wide variety of web-rendering software (e.g., different browsers), we design and implement it as a self-contained instrumentation layer around Google's Blink web rendering engine [3], which is already embedded in a variety of browsers (e.g., Chrome, WebView, Opera, Amazon Silk, etc.) and can run on different platforms (e.g., Linux, Android, Windows, and Mac OS). To implement our WebCapsule forensic engine, we inject lightweight (i.e., low overhead) instrumentation shims into Blink and its tightly coupled V8 JavaScript engine [28] (see Figure 2) in a way that allows us to inherit Blink's portability.

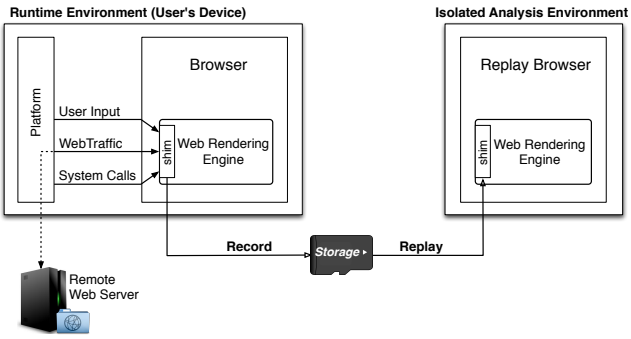

Figure 1: High-level overview of WebCapsule's record and replay capabilities. Non-deterministic inputs to the embedded web rendering engine are recorded, and can be fully replayed in an isolated forensic analysis environment where no new external user inputs or network transactions are received.

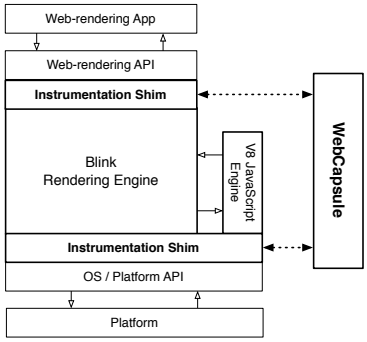

Figure 2: Overview of WebCapsule's instrumentation shims.

WebCapsule's portability has several advantages. Not only can it be readily deployed into existing Blink-based browsers and multiple platforms, but it also allows us to fully replay the browsing traces on a device (or virtual machine) whose platform may differ from the platform where the traces were recorded.

At the same time, our design choice of "living" strictly inside Blink imposes a number of constraints that make the instrumentation process challenging, especially for enabling the replay of complex browsing traces. For instance, one of the main challenges we address is how to inject lightweight instrumentation shims without altering the rendering engine's application and platform programming interfaces (APIs), so that we can fully inherit Blink's portability (the challenges we encountered are discussed in more details in Sections 3 and 5). Moreover, Blink is highly multi-threaded, making the replay of complex browsing traces challenging (e.g., Blink's main thread, HTML parser, and JavaScript WebWorkers could be scheduled differently during replay). Nonetheless, we are able to address these challenges (see Section 5) and, in turn, we can record and replay complex browsing sessions, including on popular and highly dynamic websites (e.g., Facebook).

In summary, we make the following contributions:

- We propose WebCapsule, a novel *record and replay forensic engine* for web browsers that enables an *always-on* and *transparent* fine-grained recording (and subsequent replay) of web browsing sessions. To the best of our knowledge, ours is the first work towards creating such an always-on and yet *lightweight* (i.e., low overhead) forensic engine.

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

2. PROBLEM DEFINITION AND GOALS

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

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

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

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

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

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

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

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

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

An additional approach is to appropriately choose encryption primitives, which could be implemented by the software agent that concretely collects the recorded data from WebCapsule and stores them on disk. For example, a different key may be generated to encrypt different parts of the browsing traces (e.g., one key per each new tab opened by the browser). The related decryption keys may be stored in a secure *key escrow*, and a specific key may be released only if a forensic investigation is properly authorized [9].

In this paper, we also do not focus on how to efficiently store the recorded traces to minimize storage use. However, we note that storage costs have been decreasing sharply, and that many enterprise networks already use commercial solutions to store full network packet traces for considerable periods of time [19], for security and compliance reasons. WebCapsule has the ability to *offload the recorded data in real-time* from the browser to a storage application. Thus, it may be possible to adapt current enterprise-level storage solutions to accommodate the recording of WebCapsule's browsing traces. In addition, in our future work we plan to study how the recorded data could be "aged" to reduce the granularity of historic traces, and measure the trade-off between the granularity of the recorded data and replay accuracy.

3. APPROACH AND CHALLENGES

Approach Overview. To make WebCapsule portable, we implement it by injecting lightweight instrumentation shims around Google's Blink web rendering engine [3] and the V8 JavaScript engine [28] without altering their application and platform APIs (see Figure 2).

This allows us to inherit the portability of Blink and V8, effectively making WebCapsule platform agnostic. In addition, because Blink and V8 are at the core of several modern browsers (e.g., Chrome, Opera, Amazon Silk, etc.) and of Android WebView [31], WebCapsule can be readily integrated with minor or no code changes in a wide variety of web-rendering software.

Our design and implementation of WebCapsule aims to minimize the amount of instrumentation code added into Blink. To this end, our record and replay capabilities are implemented in large part by extending Chrome's DevTools [7], which provide access to the internals of Blink (see Section 4 for more details). This allows us to inject only thin instrumentation shims at critical points in Blink's code without modifying any API, code interfaces (i.e., the members of Blink's classes) or data structures already implemented in Blink, thus obtaining a cleaner and more lightweight (e.g., low overhead) implementation of WebCapsule's functionalities.

It is worth noting that WebCapsule could be used to independently record and replay web content rendered in different browser tabs. More specifically, a browser that embeds Blink can spawn a new instance of the rendering engine for each separate tab, as it is done for example by Chromium's default process model[1] [22]. Consequently, each tab will also use its own independent instance of WebCapsule, and could therefore be recorded and replayed independently from other tabs. This is important for forensic analysis purposes, because the analyst may want to replay only a subset of the web content (i.e., only some tabs) visited by the user within a given time window.

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

In addition, Blink is highly multi-threaded, making the correct replay of complex browsing traces challenging (e.g., the main Blink thread, HTML parser, and JavaScript WebWorker threads could be scheduled differently during replay). To compensate for these problems, we implement a number of mechanisms to "re-synchronize" Blink's clock (e.g., by adjusting the result of calls to `currentTime()`) and to precisely pair network requests with the correct response drawn from the recorded traces (we discuss these mechanisms in more detail in Section 5).

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

4. RECORDING: DESIGN AND IMPLEMENTATION

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

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

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

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

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

Figure 3: Extending DevTools: instrumentation example (from `InspectorInstrumentation.idl`).

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

- Inputs that are injected by the embedder software (e.g., a browser) directly into Blink via the web-rendering API (see Figure 2). This category of non-deterministic inputs includes page control messages, such as scroll or resize the web page, as well as any user interaction and gestures via mouse, keyboard, or touchscreen interface.
- Information requested by Blink/V8 via synchronous calls to the underlying system. This information is requested via the platform API (see Figure 2) which abstracts the details of the underlying system upon which the embedder software is executing. Information in this category includes the current system time, the user-agent string describing the embedding software, total available memory, etc.
- Lastly, Blink can also request information from the platform API to be returned asynchronously via callback interfaces. Requests for remote resources, including network requests, are primarily handled using this functionality.

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

4.1 Extending DevTools

As mentioned in Section 3, one of our design goals is to create our instrumentation shims with as small a footprint as possible on the original codebase of Blink and V8. To this end, we implement a significant portion of WebCapsule's functionalities by extending Blink's built-in instrumentation facility known as DevTools [7]. This allows us to leverage existing quality code and at the same time minimize performance overhead.

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

It is possible to extend the existing DevTools functionalities by "hooking" events one would like to listen to. As shown in the example in Figure 3, this can be done by modifying `InspectorInstrumentation.idl`, which is written using a mix of IDL and C++ code. This allows us to define a special `InspectorAgent`, which we use to add WebCapsule's instrumentation shims around the web-rendering API, as explained below.

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

4.2 Recording User Input

Most user inputs (e.g., mouse movements, gestures, and key presses) are sent to Blink via its `WebViewImpl::handleInputEvent()` API. A `WebInputEvent` parameter is passed to this function carrying high-level information describing the input instance, such as its type and location on the page. To record the input, we define an instrumentation shim called `handleInputEvent`, with WebCapsule's `InspectorAgent` declared as the shim handler agent. Consequently, during execution our shim is called for each user input event. When WebCapsule is running in *record mode* the `WebInputEvent` passed to the shim is copied and stored, so that it can be re-injected *as is* during replay (see Section 5). On the other hand, when WebCapsule is not set to operate in recording (or replay) mode, then all of its shims perform no operation, letting Blink function as if WebCapsule was not at all present.

Target Element and DOM Snapshots. One of WebCapsule's goals is to provide the forensic analyst with a detailed recording of the state of the page at critical moments during the user's browsing experience. To this end, every time an input event occurs we also record the URL of the page where the event occurred. In addition, for all key presses and mouse clicks, we record the HTML representation of the element in the DOM tree that is the target of the user input. Furthermore, for events that are the main "cause" of a page transition, such as a mouse click or an "Enter" keypress (which may trigger a form data submission), we also take a full snapshot of the page DOM, including the DOM of all nested frames embedded within the page. We do so in a "blocking" fashion, so that the user input event is not propagated to any other software module, such as V8, that may alter the DOM itself before it's recorded (we intercept these events by injecting thin instrumentation shims within `WebCore::EventHandler`). While taking a snapshot of the DOM imposes some overhead, in Section 6 we show that in average the overhead is acceptable and does not significantly affect the user's experience.

4.3 Non-Deterministic Platform Calls

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

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

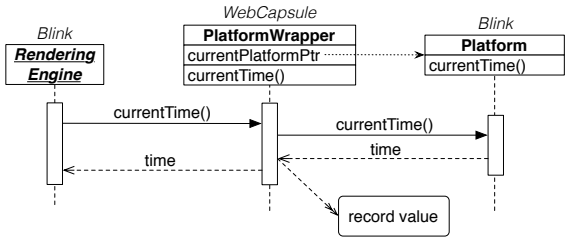

Figure 4: Simplified view of WebCapsule's platform wrapper in *record mode*. `PlatformImpl` is the actual implementation of the current underlying system platform.

4.3.1 Instrumenting Blink's Platform API

Blink provides a `Platform` interface that abstracts the provided platform API from its actual underlying implementation. To establish what specific `Platform` it is currently running on, Blink first calls a `Platform::current()` function, which returns a pointer to a static singleton instance of `Platform`. We leverage this "platform discovery" mechanism to our advantage. Specifically, WebCapsule includes a `PlatformWrapper` class, which implements the `Platform` interface and internally stores a reference to the true underlying platform returned by `Platform::current()`, as shown in Figure 4.

When recording is initiated, WebCapsule's `InspectorAgent` initializes the platform wrapper in the following manner. First, the `InspectorAgent` retrieves the pointer to the current platform instance. Next, the agent instantiates a new `PlatformWrapper`. Lastly, it replaces the value of the `Platform::current()` pointer with the address of the newly created `PlatformWrapper`. From this point on, every time Blink performs a call to any platform API, it will actually use WebCapsule's `PlatformWrapper` as its platform (see Figure 4). This design allows WebCapsule to seamlessly instrument Blink's entire Platform API while confining all instrumentation code exclusively within the `PlatformWrapper` class. Furthermore, the platform API instrumentation is completely *transparent* from the point of view of its callers.

Our `PlatformWrapper` implements the `Platform` interface as follows. For deterministic functions, the parameters passed to the wrapper's function call are simply forwarded to the same function of the wrapped (true) platform instance. The return value, if there is one, is then directly passed back to the caller. However, for non-deterministic platform functions, their implementation within `PlatformWrapper` is slightly different. When WebCapsule is in *record mode*, we make copies of both the parameters passed to and the return value generated from the call to the wrapped platform instance. The recorded values are then stored in a data structure that allows them to be retrieved and later replayed (we explain in more detail how the `PlatformWrapper` operates during replay in Section 5).

4.3.2 Instrumenting V8's Platform API

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

Platform Calls in V8 vs. Blink. To allow for accurate recording and replay, WebCapsule must capture non-determinism introduced by JavaScript that could affect page rendering within Blink. It turns out that because of how V8 is coupled to Blink, some of the

Blink instrumentations described earlier allow us to also record certain JavaScript-driven web events. For instance, JavaScript `XMLHttpRequest` network transactions are actually satisfied by Blink and utilize the same network functionality that WebCapsule already instruments. Therefore, we can record `XMLHttpRequest` transactions as any other network request (see Section 4.4). Furthermore, many of Blink's platform API functions are passed to V8 as function pointers during initialization. Therefore, some of V8's platform API calls are actually calls to Blink's platform API, which we already record via WebCapsule's `PlatformWrapper`, as discussed earlier. However, there are also a number of sources of non-determinism that reside solely within V8, and that could indirectly affect Blink's rendering. These sources include V8's own platform API and certain JavaScript functions, such as `Math.random()`.

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

Leveraging JS-to-C++ Calls. Let us now consider JavaScript's `Math.random()`. The random number generator exposed by `Math.random()` is one of the primary sources of non-determinism internal to V8. Unlike, `Date()`, V8 internally implements `random()` entirely in JavaScript. However, V8 defines several C++ preprocessor macros, which are used to define C++ functions callable from JavaScript code. We implement a new function called `HandleMathRandomVals`, which takes the return value of `random()` as a parameter. We then altered `random()` to call `HandleMathRandomVals` before returning, which in turn passes the values to be recorded to our V8 platform wrapper, (see function calls starting with '`%`' in Figure 5).

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

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

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

Figure 5: WebCapsule's instrumentation of V8's `Math.random()` implementation (from `v8/src/math.js`). Function calls starting with '`%`' are used to call C++ functions internal to V8 from JavaScript code.

4.4 Recording Network Events

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

As shown in Figure 6, WebCapsule records network events by leveraging the `PlatformWrapper` described earlier. When Blink, via a `ResourceLoader` instance, requests the platform to create a new URL loader, WebCapsule's `PlatformWrapper` returns a pointer to a `ForensicURLLoader`, which itself is a wrapper to the `WebURLLoader` actually provided by the underlying platform API. In addition, our `ForensicURLLoader` implements the `WebURLLoaderClient` interface, and passes itself as the client to the true `WebURLLoader`. Therefore, as network data arrives, the `ForensicURLLoader` is called first, records the desired information, and then calls back into the `ResourceLoader`, so that Blink can parse and render the response.

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

Browser Cache Considerations. WebCapsule's design for recording network transactions has the added benefit of abstracting the actual data source which satisfies resource requests. Obviously, the browser could satisfy Blink's network requests from a physical network, but it could also satisfy a request from the browser-level cache or from the associated resources of a browser extension. However, from the point of view of Blink these different data sources are invisible, in that where the network response is coming from does not really matter. Specifically, by recording the results of each resource request as explained above, we can replay not only network transactions, but also transparently recreate any browser cache hits without having to explicitly record the cache state.

5. REPLAY: DESIGN AND IMPLEMENTATION

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

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

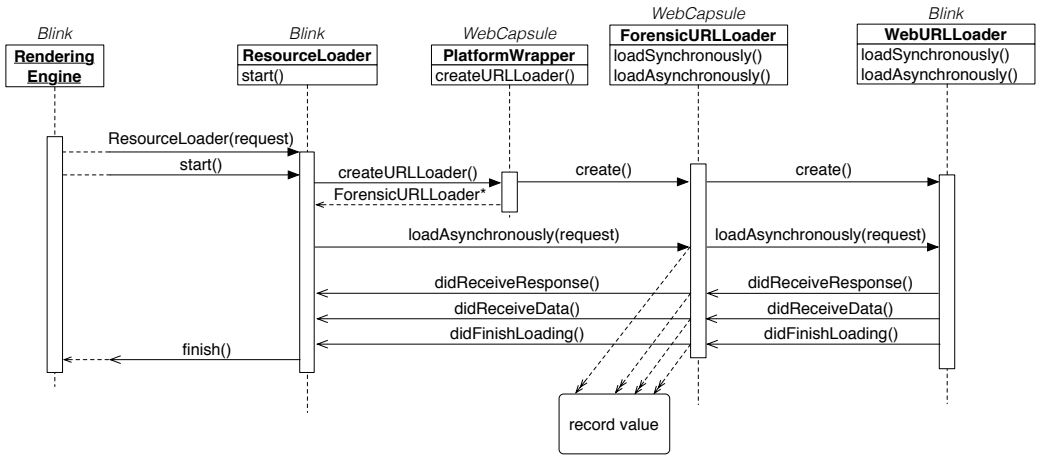

Figure 6: Simplified view of how WebCapsule records asynchronous network transactions.

replay by instructing Blink to load the first page URL in the trace and then replay the browsing events, as described in detail below.

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

Replay Strategy Overview. WebCapsule employs two replay strategies, depending on the source of the recorded data. As shown in Figure 7, inputs which originated from Blink's web-rendering API are replayed by explicitly re-calling the event handler function from which the input was recorded, effectively forcing the input (e.g., a mouse movement or keypress) to be re-injected into (and processed by) Blink. On the other hand, non-deterministic inputs obtained through the platform API are primarily replayed by simply waiting for Blink and V8 to call the platform as a consequence of the replay of the web-rendering API inputs and the rendering process. For each platform API call issued during replay, we identify the corresponding call observed during recording and directly return the previously recorded return value without having to call the true underlying system platform. Notice also that all inputs to be re-injected are timestamped, and are replayed following a precise event timeline.

For instance, as a mouse click on an HTML anchor element is re-injected into Blink via the web-rendering API, the rendering engine will start issuing the necessary network requests (via the platform API) to navigate to and render the new page. As the network requests are received by WebCapsule's `PlatformWrapper` (see Section 4.3), we return the previously recorded network response to Blink. A more detailed explanation of WebCapsule's replay mechanisms is provided below.

5.1 Replaying Web-Rendering API Events

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

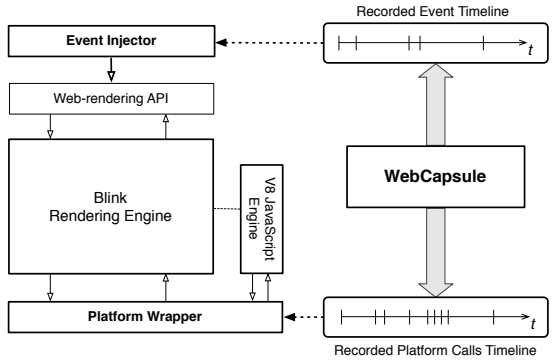

Figure 7: Simplified view of WebCapsule's replay strategy.

5.2 Replaying Platform Calls

As user inputs are re-injected into Blink and web content is rendered, the rendering and JavaScript engines will issue calls to the underlaying platform, such as network requests, calls to obtain the current system time, etc. As these calls are made, WebCapsule's `PlatformWrapper` and `JSPlatformWrapper` (defined in Section 4.3) can return the value that the same call had returned during recording. To identify the correct return value within all recorded calls of a given function, we use the combination of parameters passed to the function during recording as a key. To break "ties" on possible key collisions, the value returned during replay is simply the next unconsumed recorded return value from the related function call (with the same key) chosen in chronological order.

Challenges. As mentioned in Section 3, Blink is highly multithreaded, making the replay of some platform API calls challenging, especially for functions that take no input parameters (and therefore have no obvious key), such as Blink's `Platform:: currentTime` or V8's `OS:: TimeCurrentMillis`. During replay, depending on the (non-deterministic) scheduling of the threads, Blink and V8 may make some API calls at different "speed", compared to what happened during recording. One way to address this problem would be to record the non-determinism introduced by the thread scheduler. Unfortunately, precisely recording (and replaying) thread scheduling information from within Blink is extremely challenging. Alternatively, one may attempt to manipulate thread scheduling from "outside" of Blink. However, this is not an

option for us, because it would violate our main goal of not altering any code outside of Blink (to completely inherit its portability) and of introducing only low overhead so that WebCapsule can be used as an "always on" forensic data collection system.

Proposed Solution. To address these challenges, we use a best effort approach that we found to work well in practice. During replay, whenever `currentTime()` is called (either by Blink or V8), we return the previously recorded time value that is *closest* to the current replay time delta. More specifically, let S_{rec} and S_{rep} be the time when recording and replay started, respectively, and v_{rec} be the list of `currentTime()` return values stored during recording. Suppose that during replay Blink calls `currentTime()` at time t_{rep}. To choose the return value, we first compute the replay time delta $\delta_{rep} = t_{rep} - S_{rep}$. We then find the return value, $t_{rec} \in v_{rec}$, such that $\delta_{rec} = t_{rec} - S_{rec}$ is the closest to δ_{rep} (i.e., we minimize $|\delta_{rec} - \delta_{rep}|$). Because the web-rendering API events (e.g., mouse movements and keypresses) are re-injected respecting the relative time deltas observed during recording (see Section 5.1), the approach described above has the effect of loosely "re-synchronizing" the `currentTime()` replay clock to the user input events, thus improving replay accuracy.

5.3 Replaying Network Events

To replay network responses, WebCapsule leverages the `ForensicURLLoader` class discussed in Section 4.4. During replay, when a network resource is requested WebCapsule's URL loader finds the recorded response for that request (which may include redirection chains or error messages) using the request's URL and some other request parameters (e.g., HTTP request headers) as a key. Once the response is located, a series of WebCapsule's re-execution events are created, representing each of the `WebURLLoaderClient` callbacks to be executed (see Section 4.4). Using this technique, we are able to return the desired network response to Blink.

Challenges. In some cases, the URL of network requests driven by JavaScript (e.g., `XMLHttpRequest`) may be created dynamically. As a concrete example, consider a search box element on Amazon.com's front page[3]. Every time the user enters a character, a piece of JavaScript code issues an `XMLHttpRequest` to retrieve a set of search term suggestions. While the structure of the URL is always the same, some of the URL query parameters change. For example, the URL contains the partial search term entered by the user and a timestamp (retrieved via a call to V8's platform API). During replay, as the keypresses are reinfected into the search box, the related `XMLHttpRequest` are re-issued. However, the timestamp appended to the URL may cause a key mismatch, which would not allow us to easily find the correct response to be re-injected into Blink. As explained in Section 5.2, WebCapsule is able to "re-synchronize" the `currentTime()` replay clock to the user input events, which alleviates this problem. However, in some cases our `PlatformWrapper` may return a time value that is a few milliseconds off, compared to what was observed during recording for the same `XMLHttpRequest`, thus still causing a mismatch.

Proposed Solution. To address the above problem, we use a best effort approach that works very well in practice. During recording, every time a network request is issued, we determine if it was initiated (directly or indirectly) by JavaScript. If that's the case, we reconstruct the JavaScript call stack to identify exactly what JavaScript function caused the network request to be issued, and store this information in the browsing trace. Then, for those replay events in which there is a URL mismatch and we cannot easily identify the related network response data, we analyze the JavaScript call stack of all the *not-yet-consumed* responses in the recorded browsing trace, and return the "closest" response. Specifically, let $R_{rec} = (q_i, r_i)_{i=1...n}$ be such a set of unconsumed network requests, q_i, and related responses, r_i. Also, let $q_i^{(rep)}$, be the network request issued during replay that we are trying to match with a response. We then find the request $q_{i^*} \in R_{rec}$ whose JavaScript call stack matches the call stack associate to $q_i^{(rep)}$, and whose timestamp is the closest to $q_i^{(rep)}$'s timestamp (computed as a delta from the replay start time and record start time, respectively). Then, we return the related response r_{i^*}. In the rare cases in which the network response search still fails, we return an empty response with HTTP code `204 No Content`.

5.4 Divergence Detection and Self-Healing

The replay approaches described in Section 5.2 and 5.3 work well in practice. Nonetheless, there may be (rare) cases in which we still fail to correctly replay an event (e.g., due to complex thread scheduling issues), causing the replay to differ slightly from the recorded browsing trace. As a concrete example, assume that during recording the user clicked on an element on a given web page, P_1, and the browser navigates to page P_2. Suppose that during replay a problem occurs, and the same re-injected click does not cause the expected transition from P_1 to P_2. To recover from these problems, WebCapsule implements a replay *self-healing* approach. As mentioned in Section 4, during recording each recorded user input event includes the URL of the page where the input occurred. Therefore, if during replay the browser does not navigate to page P_2, this will cause a mismatch between the URL associated to the next input event to be replayed (i.e., P_2's URL), and the URL of the current page rendered on the browser (which erroneously remained on P_1). WebCapsule is able to detect such a mismatch, and responds to these cases by forcing the browser to load P_2, before continuing with normal replay of the remaining events on the trace.

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

6. EVALUATION

6.1 Experimental Setup

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

For all experiments, we used the Chromium[4] codebase. We deployed Chromium with our WebCapsule instrumentations on the desktop computer, and a ChromeShell APK [8] with WebCapsule enabled on the Nexus 7 (we were also able to perform some preliminary experiments with WebView + WebCapsule on Android,

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

[4]Git commit: `45eed524365a1cbc612aba31ab36aafd7788d825`.

Table 1: Functionality Tests

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

which are not reported here; we plan to expand and report the WebView experiments in our future work). All experiments were performed using the default browser process model [22], whereby each browser tab is handled in a different process (except in some corner cases). In addition, each process uses its own separate instance of Blink and V8. In this process model, WebCapsule could record multiple tabs independently. Therefore, all our results refer to experiments performed on one single tab.

Overall, our main code modifications to Blink and V8 (including the DevTools modifications and platform API wrappers discussed in Sections 4 and 5) consist of approximately 14,000 lines of code (primarily C++ code, plus a number of Python scripts for log analysis). We plan to release our WebCapsule prototype system and a variety of browsing traces collected for evaluation at `http://webcapsule.org`.

6.2 Functionality Tests

First, we performed a set of *functionality tests*, which aim to verify that WebCapsule's record and replay capabilities do not negatively impact Blink's functionalities (e.g., support for JavaScript and DOM manipulation functionalities). To this end, we leverage two popular web browser benchmarks that aim to test functional correctness. These benchmarks include Web Standards Project's Acid3 [1,32] and the Dromaeo Test Suite developed by Mozilla [10].

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

The above results show that in record mode WebCapsule is completely *transparent*, because it does not alter the core functionalities of Blink/V8. In addition, these tests show that WebCapsule can replay complex web page events with high accuracy (perfectly for Dromaeo, and 99% accuracy for Acid3), as shown in Table 1.

After investigating the browsing events related to `Test 80`, we found that the URL requested for that test had a timestamp embedded in the query string; during replay, the timestamp embedded in the requested URL was always a few milliseconds off, compared to the recording phase[6], thus causing a mismatch. We later[7] solved this URL mismatch problem with a minor adjustment to the implementation of the algorithm discussed in Section 5.3 for "resynchronizing" `currentTime()`'s return value. This allowed us to correctly replay Acid3 tests with 100% accuracy, and further improve the overall replay accuracy of WebCapsule.

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

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

[7]After initial submission

6.3 Evaluation on Phishing Attacks

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

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

To determine how well WebCapsule can record and replay phishing attacks, we measured the following quantities. For each attack trace, we wanted to quantify how well each trace could be replayed. To this end, every time a mouse click or keypress event occurred during replay, we compared the target DOM element of the replay event to the target DOM element of the same event observed during recording. For example, assume that during replay a mouse click m was injected on an anchor element, say e' =go. As discussed in Section 4, during recording not only do we store the internal details of m, but also its original target element e. Therefore, during replay we can perform a comparison between e' and e, and increment the number of target errors if the elements differ. Similarly, we recorded the URL of each page (specifically, the main frame URL) through which the user navigated while under phishing attack. Then, we compared the URLs observed during replay with the ones observed during recording, and counted differences in the page URL sequences.

The results are summarized in Table 2. As can be seen, the vast majority of traces (almost 90%) replayed perfectly. Specifically, WebCapsule was able to replay 106 out of 112 phishing traces with no page transition errors. The 6 page errors were primarily caused by corner case scenarios that are not currently handled by our proof-of-concept code and that could be fixed with additional engineering effort. For example, in some cases our network replay approach (see Section 5.3) failed to match a dynamically generated URL, and at the same time the JavaScript call stack matching algorithm described in Section 5.3 was not able to correctly recover the appropriate network response. We plan to add support for these corner cases in our next releases of WebCapsule.

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

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

Table 2: Replay correctness for phishing attack pages. Measurements performed over 112 phishing traces collected by 6 users.

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

Table 3: Performance test results. Overhead computed during recording of browsing activities on popular websites.

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

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

6.4 Record & Replay of Popular Websites

To further evaluate WebCapsule's record and replay functionalities and measure the overhead introduced by our instrumentation of Blink, we performed tests on several representative popular websites, on both Linux and Android devices (see Section 6.1 for details on device configurations).

Performance Analysis. For each website shown in Table 3, we performed a few minutes of "fast pace" browsing, during which we issued numerous input events, such as mouse clicks, touchscreen gestures, and keypresses, and navigated through several pages. During this test, we recorded the browsing activities and measured the overhead introduced by our WebCapsule instrumentation code over Blink. To accomplish this goal, we leveraged the profiling framework already implemented in Blink. Specifically, we added calls to `TRACE_EVENT` macros [27] within each single Blink function we instrumented, including around all web-rendering API (or Web API, for short), platform, and network-related "hooks" that we use to record non-deterministic inputs. This allowed us to precisely compute the CPU overhead introduced by our recording infrastructure. The results are reported in Table 3. We break down the overhead introduced by the code used to record Web API, platform, and network events, respectively.

Our results show that WebCapsule introduces reasonable overhead both on Linux and Android, making its use as an always-on system practical. For the Web API events overhead, which is always lower than 17% on Linux and 9% on Android, we need to consider that this corresponds to only a few milliseconds of overhead added to the processing of a user input event. During the recording of our browsing traces this delay was visually unnoticeable to the user. The platform overhead, which measures the added time spent to record the input and return values of calls to Blink's platform API calls, is always low on both platforms, never exceeding 2%. Finally, the time spent by WebCapsule to record network requests and responses has only a small impact on network latency, with an overhead always below 4%. The lower WebAPI performance overhead for the Android traces is likely due to the lower complexity of the mobile version of the websites (e.g., taking a snapshot of the DOM at every click is less expensive).

We also computed the amount of data that would need to be stored to archive WebCapsule's browsing traces. On average, using Chromium our browsing on popular websites produced 37.3kB/s of offloaded browsing events data, with network-related data being responsible for the vast majority (almost entirety) of the offloaded information. As we mentioned in Section 2 (see non-goals and future work), in this paper we do not focus on how to minimize storage use. However, it is worth noting that many enterprise networks already use commercial solutions to store full network packet traces for considerable periods of time (e.g., for compliance or security reasons). Therefore, it would be possible to adapt such solutions to store WebCapsule's traces.

Replaying Browsing Traces. Besides measuring the performance overhead introduced by WebCapsule in recording mode, we also tested how well the recorded traces could be replayed. As shown in Table 4 the vast majority of traces replayed correctly, with no visually noticeable difference in the rendering of the pages between recording and replay. On Linux, only Youtube caused a replay problem. Specifically, while replaying the Youtube browsing trace we encountered an assertion failure[9] on a part of Blink's code dedicated to "painting" the rendered page, causing the page, and our instrumentation agent, to freeze. We plan to further investigate and correct this issue in future versions of WebCapsule. We were also able to successfully replay searching/browsing on Google.com. However, replay was fully successful (on both Linux and Android) only with Google Instant predictions turned off [14]. Google Instant's JavaScript code seems to use a non-deterministic input that is not fully supported by our prototype. This causes one of the parameters of the URLs for network requests issued by Google Instant during replay to be slightly different from recording. While our JavaScript call stack matching algorithm (Section 5.3) helps us identify the correct (previously recorded) network response to re-inject into Blink, it appears that Instant's code performs some sort of "response content verification," which prevents the Instant search results to be correctly rendered. Because the relevant JavaScript code is heavily minimized, reverse-engineering Google Instant is fairly complex. Therefore, we plan to add support for Google Instant in future releases of WebCapsule.

For our Android experiments performed on the Nexus 7, we performed multiple tests on each of the sites listed in Table 4. For each site, we were able to record and fully replay the browsing activities. Only two sites caused some issues, and only for specific browsing scenarios. Specifically, on Youtube and Yelp our recording engine did not support the site's search functionality. Namely, after typing a search term and hitting Enter or the search button, the search results would load but in some cases the browser page would freeze. Performing other browsing activities on Youtube (with no search) did not cause any noticeable issues, and both record and replay worked with no problems. Recording and replaying Yelp also worked better when the site's search function was not used, though

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

Table 4: Record and replay tests on popular websites. Test are marked as follows: ✔ = successful test; ★ successful test with some divergence; ✖ test with problems; ✳ test with problems that we later fixed. Multiple symbols indicate different results depending on the type of browsing activity on the site.

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

we experienced some other less critical issues (e.g., a missed page transition) during replay, due to a fixable engineering issue in our prototype code.

To better understand the origin of the replay issues related to the search functionality on Yelp and Youtube, we performed an in-depth investigation of WebCapsule's execution traces. We found that the above mentioned problems were caused by a combination of a small bug within our WebCapsule code (now fixed) plus two known bugs within the version of Chromium we developed against. Our own bug was related to the implementation of the JavaScript call-stack reconstruction code (see Section 5.3), which would sometimes return an unhandled `null` value, causing WebCapsule to crash. Fixing this bug solved the problem with the search functionality on Yelp. However, Youtube's search functionality was still not working properly. After further debugging, we were able to confirm that this was entirely due to two separate bugs in Chromium itself (Issue-365858 and Issue-460328) that caused ChromeShell to crash whenever a key-stroke was entered into Youtube's search field. After implementing a workaround for both Chromium's bugs, WebCapsule was able to correctly replay Youtube's search functionalities as well.

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

Demos. To further demonstrate the record and replay abilities of WebCapsule, we have recorded five example "demo videos" that show a representative demonstration of how WebCapsule can be used to record and replay browsing activities. We produced two videos related to phishing attack traces, and three for highly popular websites (Flickr, Amazon, and Wikipedia). These videos have been posted before the submission deadline, as demonstrated by the post dates on Youtube. We also took care to remove any identifiable information from the videos themselves, so not to break anonymity during the paper submission and review process.

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

7. DISCUSSION AND FUTURE WORK

Besides forensic analysis of browsing activities, WebCapsule could benefit other applications, including the debugging of web applications and web usage mining [12], or be used as a compliance tool in some sensitive network environments (e.g., healthcare, banking, or government networks). In addition, besides social engineering and phishing attacks, WebCapsule may help to reconstruct other complex web-based attacks, including potentially helping to detect and differentiate between phishing and insider threat attacks.

While WebCapsule is currently Blink-specific, this was a pondered design decision. Our main objective was to enable accurate recording and full replay of browsing traces while maximizing portability. One alternative we have thought about was to implement WebCapsule as a browser extension. However, extension APIs are browser-specific, and in most cases do not allow for the fine-grained recording of internal rendering engine events that are critical for enabling full replay. In addition, most popular mobile browsers do not currently allow for installing "powerful" extensions, and this would have prevented the use of WebCapsule on some mobile devices. On the other hand, because Blink is already embedded in several popular browsers (e.g., Chrome, Opera, Silk, etc.), and also in Android's WebView library, implementing WebCapsule by instrumenting Blink enables its deployment in a variety of web browsers, web-rendering applications, and platforms, including on mobile devices.

Our current version of WebCapsule has some limitations. For example, as discussed in Section 3, the fact that Blink is highly multi-threaded imposes a number of implementation challenges for replay. During evaluation we did encounter some practical cases of complex web pages that cause replay divergence. However, as shown by the results reported in Section 6, WebCapsule performed remarkably well on a large variety of sites, considering that our implementation is an academic-level prototype system.

In our future work we plan to study how to further enhance WebCapsule, for example by using an approach inspired by the *logical thread schedule* proposed in DejaVu [6]. In addition, once the Blink scheduler [4] reaches a stable release, we plan to leverage it to be able to record the scheduling of Blink's internal tasks, and to replicate the same scheduling during replay.

Another limitation of our system is due to differences in CPU speed between the recording and replay environments, which may affect rendering. While during replay we use the event timelines discussed in Section 5 to synchronize the re-injection of user inputs to the underlying platform API calls (e.g., network requests, calls to current time, etc.), theoretically a web application may generate DOM changes (e.g., rotate an image or rewrite a hyperlink) repeatedly and in an unconstrained way, at a speed bounded only by the available CPU cycles. In turn, this may cause a re-injected user input to interact with the wrong DOM element. We refer to this problem as *intrinsic non-determinism*, because it is mainly due to the difference in hardware characteristics between the recording and replying devices (though CPU load due to multiple applications running on the same device during recording may also contribute to this problem).

We believe this intrinsic non-determinism is difficult to solve completely unless we perform heavyweight record and replay at the instruction level [11], rather than at the API call level. Unfortunately, this would likely introduce a much higher performance overhead and degrade portability. At the same time, we need to consider the fact that in practice intrinsic non-determinism rarely affects replay, as demonstrated in part by the positive results of our replay evaluation reported in Section 6. We expect this to hold true especially when the same device type used for recording is also

used during replay. In our future work, we will attempt to further mitigate the effects of intrinsic non-determinism by implementing more sophisticated replay clock synchronization techniques (e.g., by synchronizing to DOM changes or rendering events).

8. RELATED WORK

Forensic analysis of web-based incidents generally relies on analyzing the web browser's history, cache files, and system logs [16, 20], or on web traffic traces [15, 18, 24, 33]. However, such approaches do not provide the ability to fully reconstruct and replay web security incidents, especially for incidents that directly involve the user (e.g., phishing and social engineering attacks), as also discussed in Section 1.

Record and replay is a commonly desired feature for debugging and troubleshooting complex software and systems, and a number of previous efforts have been explored to support record and replay at different levels. For example, approaches based on virtual machines [6, 11, 21, 29] have been proposed to record system level events (interrupts, thread scheduling, etc.) and replay application and system execution at the level of single instructions. While they are designed for generic application record and replay, VM-based approaches cannot be easily deployed with low performance overhead to resource constrained devices, such as smartphones and other mobile devices. In contrast, our WebCapsule system focuses on *always on* record and replay of web browsing traces, and provides a lightweight and practical solution that can be deployed with no changes in a variety of different web-rendering applications and platforms, including mobile devices (e.g., Android devices).

WebCapsule also differs significantly from prior work that focuses exclusively on replaying specific web components, such as JavaScript code [17], user gestures and other sensor inputs on mobile devices [13], or user interactions with web applications [2]. Unlike these previous studies, WebCapsule aims to record and replay the execution of a web browsing trace in its entirety, including network transactions, and non-deterministic calls to the underlying system platform. This allows us to replay in a completely isolated environment, without requiring new user or network inputs. This is especially important when there is a need to replay potential web-security incidents for which the server-side content is ephemeral and many not be otherwise available at the time of replay.

Another work related to ours is TimeLapse [5], a developer tool that focuses on the record and replay of human visible web events, and on the interoperability with existing web application debugging tools. As discussed in more details in Section 1, TimeLapse does not work as an always on recording system and is not easily portable to different browser and operating systems (it is currently limited to Safari+MacOS). In addition, Timelapse is not transparent, because ig deeply modifies WebKit (e.g., to force a synchronous scheduling of threads). In contrast, WebCapsule can perform low-overhead always on recording, and is also transparent and portable.

9. CONCLUSION

In this paper, we presented *WebCapsule*, a novel *record and replay forensic engine* for web browsers. WebCapsule's main goal is to work as an *always-on* and *lightweight* (i.e., low overhead) forensic data collection system that enables a full reconstruction of web security incidents, including phishing and social engineering attacks. We designed WebCapsule to be a *portable* system by instrumenting Google's Blink rendering engine without altering its application and platform APIs. Our experimental results on both Linux and Android systems show that WebCapsule can accurately record and replay complex web applications, including popular websites and real-world phishing attacks, with reasonable performance overhead, thus making always-on recording practical.

Acknowledgments

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

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

10. REFERENCES

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

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

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

[4] Blink scheduler. `https://goo.gl/wzqXgC` - `https://goo.gl/I8YGu3` - `https://goo.gl/RBkhCo`.

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

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

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

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

[9] DENNING, D. E., AND BRANSTAD, D. K. A taxonomy for key escrow encryption systems. *Commun. ACM 39*, 3 (Mar. 1996), 34–40.

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

[11] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation* (New York, NY, USA, 2002), OSDI '02, ACM, pp. 211–224.

[12] ETMINANI, K., DELUI, A., YANEHSARI, N., AND ROUHANI, M. Web usage mining: Discovery of the users' navigational patterns using som. In *Networked Digital Technologies, 2009. NDT '09. First International Conference on* (2009), pp. 224–249.

[13] GOMEZ, L., NEAMTIU, I., T.AZIM, AND T.MILLSTEIN. Reran: Timeing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 ICSE* (2013).

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

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

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

[17] MICKENS, J., ELSON, J., AND HOWELL, J. Mugshot: Deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, pp. 11–11.

[18] NEASBITT, C., R.PERDISCI, LI, K., AND NELMS, T. Clickminer: Towards forensic reconstruction of uesr-browser interactions from network traces. In *Proceedings of the 2014 ACM Computer and Communication Security Conference (CCS)* (2014).

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

[20] OH, J., LEE, S., AND LEE, S. Advanced evidence collection and analysis of web browser activity. *Digit. Investig. 8* (Aug. 2011), S62–S70.

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

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

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

[24] Tcpreplay. http://tcpreplay.synfin.net/.

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

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

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

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

[29] VMWARE INC. Replay debugging on linux, October 2009. http://www.vmware.com/pdf/ws7_replay_linux_technote.pdf.

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

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

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

[33] XIE, G., ILIOFOTOU, M., KARAGIANNIS, T., FALOUTSOS, M., AND JIN, Y. Resurf: Reconstructing web-surfing activity from network traffic. In *IFIP Networking Conference, 2013* (2013).