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

Christopher Neasbitt[†], Roberto Perdisci[†‡], Kang Li[†], and Terry Nelms[≀‡]
[†]Department of Computer Science, University of Georgia
[‡]College of Computing, Georgia Institute of Technology
[≀]Damballa, Inc.
cjneasbi@uga.edu, {perdisci,kangli}@cs.uga.edu, tnelms@gatech.edu

ABSTRACT

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

Unfortunately, the modern web is becoming increasingly complex, serving highly dynamic pages that make heavy use of scripting languages, a variety of browser plugins, and asynchronous content requests. Consequently, the semantic gap between user-browser interactions and the network traces has grown significantly, making it challenging to analyze the web traffic produced by even a single user.

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

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Forensics; Network Traffic Replay

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright 2014 ACM 978-1-4503-2957-6/14/11...$15.00.
http://dx.doi.org/10.1145/2660267.2660268 .

1. INTRODUCTION

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

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

Among the applications enabled by full traffic captures, being able to automatically reconstruct user-browser interactions from archived web traffic traces would be useful in a number of scenarios. For example, malware infections through social engineering attacks [15] specifically target the browser and its user. Therefore, it is important to enable an after-the-fact analysis of such events, traveling back in time to study what actions the user performed *before* she arrived to the social engineering attack page. This may in turn allow us to better understand *browsing behavior* patterns related to such attacks, and suggest new defense mechanisms.

Besides computer forensics, reconstructing user-browser interactions from traffic traces may benefit other interesting applications, such as web-usage mining [14], whose main goal is to understand how users interact with web pages to enhance their browsing experience or to improve on personalized advertisement strategies [5, 11].

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

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

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

- Can we infer *what element in a web page was "clicked" by the user* to reach the next desired resource?

To answer these research questions and measure the reconstruction accuracy, we propose ClickMiner, a novel system for reconstructing user-browser interactions from network traces. We also compare ClickMiner to a "naive" referrer-based click inference (RCI) approach based on ReSurf [17], a recently proposed web traffic analysis system. For both ClickMiner and RCI, we assume to be given a (possibly partial) recording of the web traffic generated by a user within a given time window of interest. Our main goal is to reconstruct the sequence of "clicks" that the user performed within the browser. Notice that we focus only on user-browser interactions that cause the browser to initiate an HTTP request for a new web page, and that our definition of *click* goes beyond mouse clicks, and includes other events such as typing an URL directly, pressing `Enter` on the keyboard to follow a link, etc. (see Section 2).

The RCI approach is based on first constructing the *referrer graph* as in ReSurf [17], where a node represents an HTTP request, and two nodes are linked together by a directed edge according to their `Referer` request header fields. Then, this graph is pruned using a number of heuristics to filter out requests that were (likely) automatically generated by the browser during rendering (see Section 3.1).

Our ClickMiner system, instead, takes a very different approach: it reconstructs user-browser interactions by *actively replaying* the recorded HTTP traffic within an *instrumented browser*. At a high level, given the HTTP request-response pair (i.e. HTTP exchange) (q_0, r_0) related to an initial page in the trace, we force the browser to issue request q_0, and we feed it back response r_0 from the trace. Through the natural rendering of response r_0, the browser will issue another number of HTTP requests to retrieve page components. ClickMiner serves all the related responses to the browser from the trace. Essentially, the browser *consumes the traffic trace* until it reaches a *resting state*, whereby some action is needed for the browser to continue to consume the remaining HTTP traffic. At this point, we consider the next yet to be consumed HTTP exchange in the trace as a *candidate* user-browser interaction, feed it to the browser, and continue until all HTTP traffic has been consumed. In practice, the replay process used by ClickMiner is much more involved; we therefore defer the details to Section 5.

Contributions. We make the following contributions:

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

2. PROBLEM FORMULATION

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

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

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

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

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

HTTPS traffic. One may think that because many web services are transitioning to HTTPS, most recorded web traffic is going to be encrypted, and both the ReSurf-based RCI approach and ClickMiner will become less useful in the near future. However, it is important to notice that many modern enterprise networks already deploy web proxies that allow for SSL man-in-the-middle [7] (SSL-MITM) to enable the inspection of all HTTPS traffic (e.g., to enforce traffic filtering policies). Therefore, enterprise networks can already easily record the content of most HTTPS communications (sensible traffic capturing policies would avoid SSL-MITM for banking applications or other known sensitive activities).

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

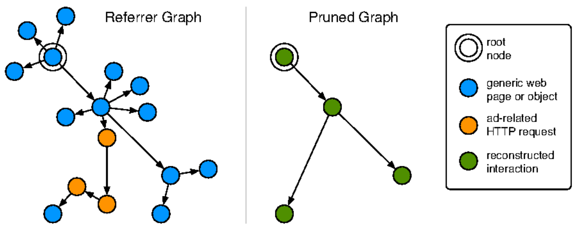

Figure 1: Example of referrer graph pruning (the length of the edges reflects the referrer delay).

the interaction by leveraging "artifacts" of q that may be observed in the network trace.

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

3. REFERRER-BASED INFERENCE (RCI)

In this section, we briefly explain how we perform referrer-based click inference (RCI, for short). Our RCI algorithm is based on ReSurf's approach [17]. Please, refer to [17] for further details on the construction and pruning of the referrer-based graph.

3.1 Building the Referrer Graph

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

Clearly, not all nodes in $\mathcal{G}$ are generated due to user-browser interactions. In practice, most of the nodes in the referrer graph are related to automatic requests issued by the browser during page rendering. In order to infer user clicks, we would like to derive a pruned graph $\mathcal{G}'$ that only contains nodes that are directly related to user-browser interactions. To this end, we use a number of heuristics (as in [17]) that allow for identifying and pruning away nodes that are likely due to automatically generated requests. After applying these pruning heuristics to obtain $\mathcal{G}'$, which typically contains many fewer nodes and smaller connected subgraphs than $\mathcal{G}$, we classify all nodes $v \in \mathcal{G}'$ as directly related to user-browser interactions. Figure 1, shows a visual example of the results of the pruning process.

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

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

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

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

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

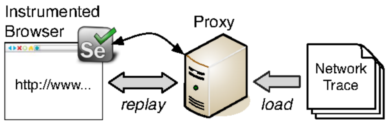

Figure 2: ClickMiner system overview.

4. CLICKMINER SYSTEM OVERVIEW

In this section, we give an overview of how ClickMiner works. A simplified view of the traffic replay process described below is provided in Figures 2 and 3.

In-Browser Replay: At a high level, ClickMiner works as follows (see Figure 3). Given a network trace containing full packet captures (1), we first reassemble all HTTP flows and the related HTTP exchanges. These flows are then loaded into a proxy application. We instrument a browser via a *browser driver* plugin implemented on top of Selenium WebDriver [4]. We chose to leverage Selenium WebDriver because it is compatible with most major browsers, making it easier to replay the traffic on different browser versions, e.g., chosen according to the user-agent string recorded in the network traces.

The instrumented browser is configured to use the proxy application as its HTTP proxy, and is responsible to replay the network traffic and reconstruct the user-browser interactions that took place during traffic capture. To initiate the reconstruction process, the browser driver will ask the proxy for the URL of the first HTTP request recorded in the trace that contains clickable elements within

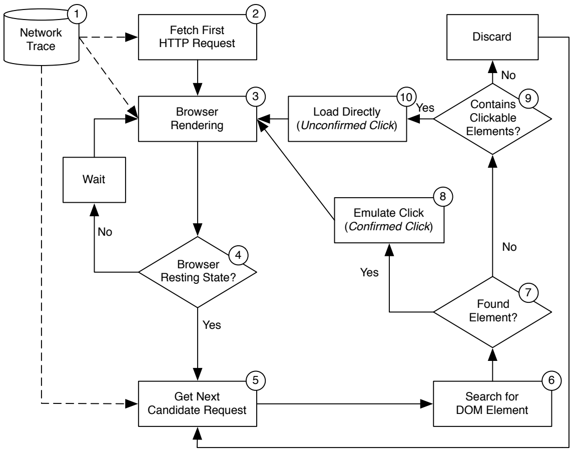

Figure 3: ClickMiner's process (simplified).

the body of its response (2). Then, the browser is instructed to load this URL as if a user typed it on the address bar. As the browser renders the web page (3), all related automatic object requests (e.g., to load images, frames, etc.) are sent to the proxy, which retrieves the responses from the recorded HTTP flows and passes them back to the browser (notice that the proxy is not allowed to retrieve content from other than the recorded network trace). This could be seen as forcing all browser requests to be served from a cache.

Once the browser fetches all necessary objects and renders the current page, it will reach a *resting state* (4), whereby no further previously recorded HTTP requests would be issued by the browser without explicit user interaction (see Section 5.3 for a more accurate definition of resting state). At this point, the browser driver queries the proxy for the next HTTP request in the network trace that has not yet been "consumed" by the browser (5). The returned request represents a *candidate click*. To verify if this request was actually caused by a user click, the browser driver inspects the DOM of the pages currently rendered by the browser (6). If a *clickable element* is found (7), whose attributes (e.g., the `href` attribute of an <a> tag) match the URL of the candidate click, we consider this to be a *confirmed click*, which reports information including the URL of the page currently rendered in the browser (i.e., the URL in the address bar), the path in the current page DOM of the found clickable element, and the URL of the next Web resource that would be requested as a consequence of clicking on that element. In Section 5 we will explain in details how (and why) our URL matching process uses an *approximate matching* approach to compensate for dynamically generated URLs.

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

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

5. CLICKMINER SYSTEM DETAILS

5.1 In-Browser Traffic Replay

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

Then, our browser driver instructs the browser to replay request q_j. This request is received by ClickMiner's proxy, which responds by serving r_j (see Section 5.2). As the browser renders r_j, it may issue a set of subsequent requests $\{q_r\}$, which are sent to the proxy and served accordingly from the trace.

5.2 The Role of ClickMiner's Proxy

ClickMiner's proxy (see Figure 2) is mainly responsible for reassembling TCP flows and extracting HTTP exchanges from previously recoded network traces, and for serving the recorded web content upon request to the instrumented browser. When the browser (which is setup to use our proxy application as its *HTTP proxy*) sends an HTTP request q', the proxy extracts the corresponding absolute URL u' from q', and searches the list of exchanges, l, to find the first occurrence of an exchange whose query URL matches u'. Let (q_k, r_k) be such a pair. At this point, the proxy retrieves the content of response r_k from the trace, sends it to the browser, and marks the pair (q_k, r_k) as *consumed*.

(Approximate) Matching of Requests and Responses. Due to the highly dynamic nature of modern web pages, during replay some HTTP requests issued by the browser may not match any of the originally recorded traces. This may occur for example when URL parameters encode data that is either system- or time-dependent, or is randomly generated. To attempt to satisfy these requests, the proxy applies an *approximate matching* algorithm. Let q_u be the request issued by the browser during replay. We measure the similarity between q_u and all not-yet-consumed requests in the trace under the same domain name (or IP address) as q_u. Let q_a be one of such yet to be consumed requests. To compute the similarity between q_u and q_a, we measure the similarity between their URL path strings, the similarity of the set of parameter names, and the number of matching parameter values. We also consider q_u and q_a to be more similar when the timestamp of q_a is closer to the timestamp of the last request that was successfully consumed from the network trace. We then combine all these similarity scores, and match q_u with the most similar q_a in the trace. In the eventuality that the browser issues an HTTP request that does not match any not-yet-consumed request in the trace, the proxy will simply respond with a `404 Not Found`.

It is possible (though we found to be rare in practice) that due to approximate matching a request may be mistakenly consumed, potentially causing ClickMiner to miss a "click" later on during trace replay. The effect of this type of inappropriate match is akin to missing clicks due to the effect of the browser cache. Section 5.6 details this scenario as well as ClickMiner's recovery mechanisms.

5.3 Browser Resting State

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

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

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

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

Accordingly, we apply these three rules.

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

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

(R3) It is possible for a page to take a rather large amount of time to reach a resting state based upon (R1) and (R2) alone, given a significantly long trace. Therefore, we put an upper limit on the number of polling attempts we make before a browser window is considered to be in a resting state. We conservatively set the upper limit to $25 \cdot t_{poll}$ in our experiments.

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

5.4 Reconstructing Interactions

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

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

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

Challenges. There are a number of practical complications that make it difficult to locate the URL u within the DOM. For example, u may have been dynamically generated via JavaScript as a product of the user clicking on a DOM element with an `onclick` or `onmousedown` attribute. In addition, u may have been requested as a result of the user's interaction with an embedded object, such as a Flash object. Lastly, u may have been requested from a page satisfied from the browser cache, making the page containing u unsearchable as it may not have been recorded in the trace. These scenarios are addressed in Sections 5.5 and 5.6.

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

5.5 Dynamic and Embedded Content

The algorithm described in Section 5.4 for finding user-browser interactions is not able, by itself, to reconstruct events for which the actual click is handled outside of the context of the page's DOM. For example, a click event may be handled via a JavaScript (JS) that dynamically compiles the URL of the next page to be loaded and triggers a `document.location` change. Correctly replaying all possible such cases is extremely challenging. What we describe in this section is a *best effort* approach to close some of the semantic gaps present in the network traces, thus further aiding the work of a forensic analyst.

JavaScript Mediated Clicks: One possible way to reconstruct clicks handled by JS would be to apply program analysis techniques to all scripts contained in the currently open pages. However, JS code embedded in highly dynamic pages can be very complex [13], and is often obfuscated to protect it from straightforward reverse engineering. Therefore, instead of leveraging program analysis techniques, we take a lightweight "network-oriented" approach to *approximately* reconstruct the clicks. The idea is to identify all clickable elements in a page that subscribe to events such as `onclick`, `ondoubleclick`, `onmousedown`, etc., and "test" them to check whether any of these elements leads to a particular URL. In practice, we proceed with emulating clicks to each one of these elements in the current page DOM under analysis. If, as the result of a particular click, the browser requests the next expected URL (i.e., the URL of the candidate interaction currently being considered), our proxy will simply serve the response. However, if the click causes the browser to request an URL that does not match the expected URL, the proxy will serve a `204 No Content` response, thus preventing the browser from inadvertently loading an "undesired" page. Still, there exists a challenge due to the fact that clicking on certain elements may occlude parts of the page, thus in a way changing the "state" of the page itself. However, our emulated clicks are based on programmatically "activating" DOM elements, and in practice the visual state of the page is often irrelevant.

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

5.6 The Effect of the Browser Cache

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

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

In general, though user-browser interactions that are not reflected in the network trace cannot be reconstructed with certainty, ClickMiner may still be able to infer them, as we discuss in Section 6.2. More importantly, missing one interaction does not jeopardize the reconstruction of other interactions that can be recovered from farther along the trace.

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

6. CLICK GRAPH ANALYSIS

In this section, we describe how we can analyze the results of ClickMiner by building a *click graph* (Section 6.1). Furthermore, we explain how we can combine ClickMiner's click graph with the referrer graph (see Section 3.1) to infer additional user-browser interaction that may have been missed during the in-browser replay process (Section 6.2).

6.1 Building the Click Graph

We represent a user-browser interaction reconstructed by ClickMiner as a tuple (p, e, q), where q is an HTTP request, p is a web page URL, and e is a DOM element in the (rendered) page p that when clicked caused the browser to issue a request for q. Let $m = \{(p_i, e_i, q_i)\}_{i=1...n}$ be the list of all user-browser interactions *mined* by ClickMiner via in-browser traffic replay, as explained in Section 4 and Section 5. We build a directed acyclic graph $C = (V, E)$, where each node in V represents a "click tuple" from the list m. A directed edge $((p_w, e_w, q_w) \rightarrow (p_y, e_y, q_y)) \in E$ exists if page p_y was reached as a consequence of request q_w, which in turn was issued by emulating a user click on an element e_w of page p_w, for example.

Notice that the list m of interactions reconstructed by ClickMiner includes both *confirmed* and *unconfirmed* clicks (defined in Section 5.4). Remember that an *unconfirmed* click U is a reconstructed interaction for which ClickMiner was not able to find (i.e., confirm) the existence of a related clickable DOM element. Formally, an unconfirmed click can be represented by a node $U = (p_u, e_u, q_u)$, where p_u and e_u are unknown values (in practice, p_u and e_u are *null*). However, in some cases the page of an unconfirmed click can be inferred from the referrer header field of q_u, as discussed below in Section 6.2.

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

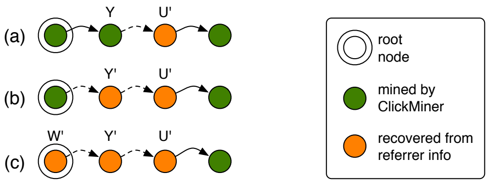

Figure 4: Examples of augmented click graph.

6.2 Augmented Click Inference

Ideally, the click graph C will include the entire sequence of clicks a user made during a browsing session. However, there are some situations in which ClickMiner may fail to detect a user-browser interaction (see Sections 5.5 and 5.6), thereby causing the generation of an incomplete click graph. To recover from some of the missing clicks, we augment the click graph produced by ClickMiner with information extracted from the HTTP referrers in the network trace, as follows.

Assume $U = (p_u, e_u, q_u)$, with $p_u = null$ and $e_u = null$, is an unconfirmed interaction mined by ClickMiner, and let ref_u be the referrer carried by q_u. In this case, we can infer that p_u should be equal to ref_u. Therefore, we can replace U with $U' = (\text{ref}_u, null, q_u)$. At this point, if the click graph C contains a node $Y = (p_y, e_y, q_y)$ where q_y's URL equals ref_u, we can draw an edge $Y \rightarrow U'$. In other words, we inferred that the page p_u was missing in the click graph C, and we were able to derive it by leveraging referrer information, as shown in the example Figure 4(a).

Assume now that ClickMiner failed to mine the node Y. That is, $Y = (p_y, e_y, q_y)$ is missing from the click graph C. Also, suppose that the network trace contains an exchange (q_y, r_y), for which q_y's URL equals ref_u. In this case, we can further infer that the click graph C should have contained a node $Y' = (p_y, e_y, q_y)$, where p_y and e_y are unknown. We can repeat the process described above until the last inferred node can be connected to an existing node in C, as shown in Figure 4(b), or a "root node" $W' = (null, null, q_w)$ is recovered from referrer information, whereby q_w carries no referrer, as shown in Figure 4(c).

We can also apply some of the pruning techniques used for RCI (Section 3.1) to the augmented click graph, thus further refining it. For instance, we can apply ad and social widgets pruning to the augmented click graph in a manner similar to that described in Section 3.1. In addition, the referrer delay pruning can also be applied to the augmented click graph, allowing for a direct comparison between ClickMiner and RCI, which we discuss in Section 7 (see Figure 5).

Differences w.r.t. RCI. It is worth noting that, unlike the RCI method (Section 3), ClickMiner selectively leverages referrer information *only to fill in some gaps* between interactions in the click graph that were mined via in-browser traffic replay. On the other hand, RCI "naively" uses the entire set of referrers found in the traffic traces.

7. EVALUATION

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

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

7.1 Recording User-Browser Interactions

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

Each user interacted with an instrumented Firefox browser that allowed us to record most UI-level user-browser interactions. For example, every time the user performed a mouse click within a page, we recorded a tuple $(t_e, \text{page}_e, \text{elem}_e, \text{dst}_e)$ containing the timestamp of the click event e, the URL of the main page where the click happened, the DOM element that was clicked, and the destination page URL. We also recorded key-press events. At the same time, we recorded the related full packet traces, and a video of all UI events (essentially, a video of the entire Desktop). A few users offered to record more than one 20-minutes browsing session, and overall we collected 24 different browsing traces that we used in our experiments.

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

Table 1 and 2 report the number of HTTP requests and recorded user-browser interaction for each trace (a complete description of the table content is given in Section 7.2.2).

7.2 Reconstructed User-Browser Interactions

7.2.1 RCI Results

To evaluate the RCI method and compare it to ClickMiner, we first built a referrer graph (see Section 3.1) from each of the network traces recorded during our user study, and then pruned the graphs as explained in Section 3.1. Assume that $v_i = (q_i, r_i)$ is a node in the pruned RCI graph generated from a network trace. We compare the URL of request q_i with the set of user-browser interactions recorded during the related user browsing session. For example, if q_i matches the destination dst_e of a click event e, we consider q_i as a *true positive*, and mark event e as "consumed" to avoid matching the same recorded interaction more than once. On the other hand, if q_i does not match any of the recorded user-browser interactions, we label it as a *false positive*.

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

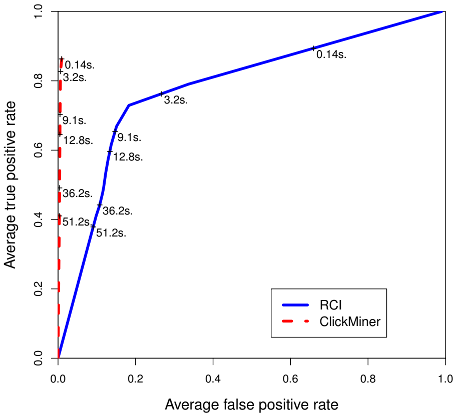

Figure 5: Trade-off between true and false positives.

The blue solid curve in Figure 5 shows the trade-off between the average true and false positive rates obtained by varying the referrer delay threshold, θ_δ. Per each trace, the true positive rate is computed as the fraction of user-browser interactions recorded during a user's browsing session that are represented in the pruned referrer graph, whereas the false positive rate is the fraction of graph nodes that do not match any recorded interaction. The average is computed across all 24 available user browsing traces.

7.2.2 *ClickMiner Results*

To evaluate ClickMiner, for each user browser session, we performed in-browser traffic replay as explained in Section 5. Then, we built the click graph, as explained in Section 6. Given the click graph, we match each node in the graph with the user-browser interactions recorded during our user study, in a way similar to what we discussed in Section 7.2.1 for the RCI method. Tables 1 and 2 report the results obtained using ClickMiner with the augmented click graph (see Section 6.2), but without any referrer delay pruning. On the other hand, the dashed red line in Figure 5 reports the average trade-off (across all traces) between true and false positives obtained by pruning the augmented click graph at different referrer delay thresholds, so that we can directly compare the results to those produced by RCI.

In all tables, the *Trace Number* is simply a trace identifier; *HTTP Requests* is the number of requests in the captured network traces; *Recorded Clicks* is the number of user-browser interactions recorded via the instrumented browser used by the participants of our user study; *Mined Clicks* indicates the number of user-browser interactions inferred as explained in Section 6.2, averaged across five runs per trace; *Matching Clicks* indicates the number of mined clicks that match a recorded click. Finally, *TPR* is the true positive rate, i.e., the percentage of recorded interactions that match an interaction inferred by our system, whereas *FPR* is the false positive rate, i.e., the number of mined clicks that do not match any recorded interaction.

It is worth noting that, to produce the results in Table 1 and 2, we replay each user browsing trace for five times, and compute the average and standard deviation for the click mining results obtained per each run. We do this because every time a traffic trace is replayed within the browser, the rendering of highly dynamic pages may introduce slight changes, such as the automatic generation of some new HTTP requests via JavaScript[2]

Table 1: ClickMiner results - Group1 (no-cache)

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

Table 2: ClickMiner results - Group2 (cache)

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

By comparing the average true and false positive rates reported in Tables 1 and 2, we can see that the browser cache certainly has an impact on the ability of ClickMiner to reconstruct user-browser interactions. For example, when the cache is disabled (Table 1), ClickMiner achieves more than 90% TRP in 7 out of 13 traces, and 100% in 3 traces. On the other hand, only 3 out of 11 traces recorded with the cached enabled can achieve more than 90% TRP. Nonetheless, even in the cache enabled case, ClickMiner in average can retrieve more than 81% of user-browser interactions with less than 1.16% of false positives. For comparison, using the *non-augmented* click graph (i.e., without augmenting ClickMiner's output with referrer information), we obtained 70.47% TPR at 1.72% FPR, for the traces in Group2.

We also analyzed the percentage of confirmed and unconfirmed user-browser interactions (see Section 5.4) that ClickMiner was able to discover. On average, ClickMiner was able to reconstruct

[2] To reduce execution time, in our current evaluation we did not turn on ClickMiner's system components describe in Section 5.5 that deal explicitly with dynamic and embedded content.

the actual DOM element clicked by the user for around 46% of the true positive interactions in Group1, and 48% in Group2.

ClickMiner vs. RCI Notice that the dashed red curve (ClickMiner) in Figure 5 does not reach the top-right corner because, unlike the RCI method, the number of user-browser interactions returned by ClickMiner is bound by what can be discovered through the in-browser replay process. Augmenting the click graph only fills some "gaps", as explained in Section 6. Nonetheless, at low false positive rates, ClickMiner clearly outperforms RCI. For example, we would have to tolerate more than 20% FPR, for RCI to reach a TPR higher than what obtained with ClickMiner.

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

We performed our experiments with ClickMiner on an off-the-shelf desktop machine with an Intel Core i7-870 CPU and 8GB of RAM. Overall, ClickMiner required an execution time between approximately two to five times the length of the browsing session. The median execution time for the traces was about 76 minutes for traces in Group1 and 34 minutes for Group2. After inspection, we noticed that ClickMiner's execution time is dominated by inspecting the DOM of each relevant web page rendered by the browser searching for the elements that were clicked by the users. The lower time needed to process traces in Group2 is likely due to the effect of the cache, because it lowers the number of HTTP requests to be considered as a possible interaction. In our future work we will investigate further optimizations that would allow us to reduce ClickMiner's execution time.

7.2.3 Discussion

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

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

Common causes of false positives. False positives are primarily represented by unconfirmed user-browser interactions reconstructed by ClickMiner as a result of HTTP requests that were not properly "consumed" during in-browser replay. For instance, assume the HTTP exchange (q, r) represents a web advertisement loaded through a JavaScript-driven request, and that response r contains clickable HTML content (the ad itself). Also, assume the user who generated the trace never actually clicked on the ad during her browsing session. It is possible that during ClickMiner's in-browser reply, request q will not be made, because the JavaScript running on the rendered page that was supposed to load the ad happens to dynamically construct a significantly different URL to be requested (e.g., for a different ad). Therefore q remains "unconsumed". At a certain point during trace replay, q may be considered as the next "candidate click", because it is an outstanding, unconsumed request in the trace. As no DOM element can be found relating to q's URL, ClickMiner will render r in a new window, and classify q as an unconfirmed click.

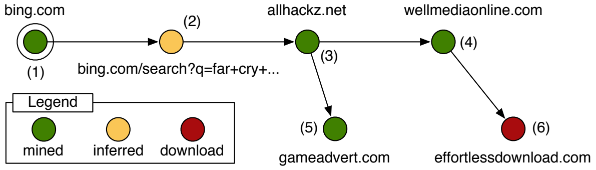

Figure 6: Reconstructed click graph (case study 1)

Common causes of false negatives. An example of "missed interactions" that is prevalent within our user study involves Google searches. The search results page is built dynamically as the user types, via asynchronous requests. ClickMiner is not able to infer all these UI events from the network traces, and therefore it is difficult to have the browser replay the same asynchronous requests. Consequently, the "replayed" DOM will be different from the one on which the user originally clicked (see related discussion in Section 5.4), and we will not be able to find the DOM element needed to mine a confirmed interaction.

Another source of false negatives is represented by exchanges that are simply missing from the trace, because of the caching effect. In addition, while more rare, it may also happen that an exchange is erroneously consumed during replay, as a consequence of ClickMiner's approximate URL matching process (see Section 5.2). This may "steal" the HTTP exchange from a correct match to an actual user interaction.

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

8. CASE STUDIES

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

8.1 Case Study 1

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

Summary of Results: By running ClickMiner on the network trace recorded during the user-generated browsing session described above, we were able to reconstruct the *full chain of user-browser interactions* that led the user to the malware download. In particular, ClickMiner reduced the amount of information to be analyzed by

[3] MD5: `c94f917fdc39dfb7245ebdd674b2bdf8`

a forensic analyst *from 316 HTTP exchanges to only 6 click graph nodes*, as shown in Figure 6. Additionally, ClickMiner was able to correctly identify what element the user clicked on to initiate the malicious download. Conversely, RCI did not perform as well. Even by adjusting the filtering and referrer-delay threshold to reduce RCI's false positives without missing any of the interactions, RCI produced a graph with 79 nodes, instead of only 6.

Details: Figure 6 shows that `bing.com` (1) was identified as the root. Node (2) is related to the `bing.com` page that shows the search results. This node is colored yellow because its presence in the click graph was inferred through the referrer reported by the next page request on `allhackz[dot]net`. The reason why node (2) is *inferred*, rather than directly reconstructed through ClickMiner's traffic replay, is that `bing.com` populated the search results page dynamically as the user typed the search terms, using asynchronous HTTP requests. As mentioned in Section 5.4, it is difficult to replay this type of traffic without knowledge of the semantics of the response content (e.g., a json object). Nonetheless, ClickMiner was able to infer that the user clicked on a search result link to access `allhackz[dot]net` (3). In addition, through the replay of the response for node (3), ClickMiner was able to correctly identify the DOM object that the user clicked on to reach node (4). More precisely, using the *click emulation* approach described in Section 5.5, ClickMiner identified the location in the DOM of the download button that would fire an event and call a JavaScript that opened page (4). Finally, while the user's browser was automatically redirected from node (4) to the actual file download hosted on node (6), the page on node (4) also contained two <a> tags linking to the URL of node (6). Therefore ClickMiner connected the two nodes and essentially reported that the user likely clicked on one of those two links. While this is not completely accurate, because the download happened through an automatic redirect, the click graph produced is correct. Notice also that if the page on node (4) contained no explicit hyperlink to (6), ClickMiner could still have inferred the link between the two pages through the referrer information, for example.

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

8.2 Case Study 2

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

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

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

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

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

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

In summary, ClickMiner was able to correctly or very closely derive the chain of interactions by which the user downloaded each malicious executable. In the cases above, the traffic corresponding to each search was not fully recorded; yet, ClickMiner was able to infer the user-browser interactions related to the search pages by constructing an augmented click graph (see Section 6). In addition, ClickMiner was able to reduce the input traffic from several hundreds HTTP exchanges, to only few that are highly related to the security incident being analyzed.

9. LIMITATIONS AND FUTURE WORK

One of ClickMiner's primary applications, as demonstrated in Section 8, is the after-the-fact replay of user-browser interactions before and up to the occurrence of a security incident. However, potential attackers could seek to thwart our analysis by reducing the ability to replay the recorded incidents. The following are some examples of possible techniques an attacker could use to this end.

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

[4] MD5: `e64bef0c045430dfdbc02a824cd19003`

[5] MD5: `25b1d21a555bbd05856555a01b5be2b4`

[6] MD5: `2d484f0614b1d720dfbccdd788a3ad9c`

tags, etc.) to prevent certain user-browser interactions from being correctly reconstructed during replay.

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

10. RELATED WORK

Traffic Replay: While a number of traffic replay systems have been proposed in the past [1, 9], most tools have focused on replaying traffic at an IP-level or TCP/UDP-level granularity. For example, Hong et al. perform interactive replay of internet traffic [9] by emulating a TCP protocol stack. ClickMiner is different in that it implements in-browser replay of application-level (HTTP) traffic.

In [6] the authors describe WebPatrol, a system for automated collection and replay of malware infection scenarios. Importantly, ClickMiner's purpose is very different and much more general, compared to WebPatrol. In fact, while WebPatrol is limited to replying malware infection scenarios that can be automatically collected by its own "honey clients", ClickMiner aims to reconstructing user-browser interaction from real-world traffic traces. Furthermore, ClickMiner is not limited to reconstructing events related to malware infections, and can instead aid the forensic analysis of other security incidents involving user-browser interactions, such as social engineering attacks, phishing, etc.

Traffic Analysis Tools: Over the last several years numerous network forensic analysis tools have been constructed [12]. These tools allow administrators to monitor, capture, analyze, and in some cases replay network traffic in order to aid in network crime investigations (i.e. malware infections, denial of service, etc.) and help generate appropriate incident responses. To the best of our knowledge, ClickMiner is the first tool that can automatically reconstruct detailed user-browser interactions from web traffic traces, and it could therefore be used as a component of a more comprehensive network forensic analysis framework.

Web Usage Mining: Web usage mining has been studied for example in [16, 10, 8]. Wu et al. [16] proposed a system named SpeedTracer which applies data mining techniques to web server logs in order to extract and group user interaction paths. Etminani et al. [8] propose the application of Kohonen's Self Organizing Maps to preprocessed web logs for extracting common interaction patterns. Also, ReSurf [17] aims to reconstruct web surfing activities from traffic traces via an analysis of referrer headers. Our work takes a different approach, because it aims to reconstruct user browsing activities from recorded network traffic via in-browser replay. Our approach has the advantage of mining complex cross-site activities invisible to techniques that rely on web server logs, and outperforms the RCI approach based on ReSurf [17], as we showed in Section 7.

11. CONCLUSION

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

12. ACKNOWLEDGMENTS

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

13. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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