Cache, Trigger, Impersonate: Enabling Context-Sensitive Honeyclient Analysis On-the-Wire

Teryl Taylor, Kevin Z. Snow, Nathan Otterness, and Fabian Monrose
University of North Carolina at Chapel Hill
{tptaylor, kzsnow, otternes, fabian}@cs.unc.edu

***Abstract*—Today's sophisticated web exploit kits use polymorphic techniques to obfuscate each attack instance, making content-based signatures used by network intrusion detection systems far less effective than in years past. A dynamic analysis, or *honeyclient analysis*, of these exploits plays a key role in initially identifying new attacks in order to generate content signatures. While honeyclients can sweep the web for attacks, they provide no means of inspecting end-user traffic *on-the-wire* to identify attacks in real time. This leaves network operators dependent on third-party signatures that arrive too late, or not at all.**

In this paper, we introduce the design and implementation of a novel framework for adapting honeyclient-based systems to operate on-the-wire at scale. Specifically, we capture and store a configurable window of reassembled HTTP objects network-wide, use lightweight content rendering to establish the chain of requests leading up to a suspicious event, then serve the initial response content back to the honeyclient system on an isolated network. We demonstrate the power of our framework by analyzing a diverse collection of web-based exploit kits as they evolve over a one year period. Our case studies provide several interesting insights into the behavior of these exploit kits. Additionally, our empirical evaluations suggest that our approach offers significant operational value, and a single honeyclient server can readily support a large campus deployment.

I. Introduction

Today, the rapid and wide-spread proliferation of browser-based exploits distributed via highly obfuscated web content is an all too familiar event. Sophisticated off-the-shelf exploitation toolkits detect vulnerabilities in victim's browsers and plugins prior to exploitation and use this information to dynamically and uniquely craft the next stage of attack, ultimately injecting highly targeted malicious code on the victim system. More concerning is that these kits can deliver malware without our knowledge while visiting legitimate sites; for example, by either identifying and exploiting vulnerabilities in a multitude of web servers, or by simply launching massive campaigns through advertising networks that monetize these sites and injecting redirections to their malicious web servers.

The status quo in defending networks from these attacks is the use of network intrusion detection systems (NIDS) that perform deep packet inspection to search HTTP traffic as it passes a network border. These systems perform signature matching, blacklisting, or statistical analysis to identify potentially malicious traffic. Sadly, attackers routinely thwart these defenses by rapidly changing their environment through 1) using polymorphic techniques on exploit payloads, 2) frequently moving exploit kits to new servers, 3) constantly changing domain names, and 4) morphing traffic to bypass signatures in an effort to look "normal" in the context of surrounding traffic.

Of late, honeyclient analysis has been used to address some of the aforementioned weaknesses, especially as it relates to detecting web exploit kits. The idea is to use a secure virtualized machine (VM) to navigate, render and execute potentially malicious web pages. Honeyclients dynamically track system state change caused by a specific application or website. System state change (*e.g.*, files written, processes created, etc.) has been shown to be an effective metric in classifying malicious applications [3]. Today, many security vendors routinely crawl the Internet with large clusters of VMs in an attempt to identify malicious websites [34, 10]. The result of these analyses are typically used to generate blacklists or other information deemed useful for improving a network's security posture.

However, the model of honeyclient analysis is not without drawbacks. Crawlers heavily depend on the quality of the URL seeding used to initially discover potentially malicious web pages, and there is no guarantee that crawlers will discover the same exploit kits that are visited by third-parties using a NIDS. Deploying any generated signatures can take days or weeks, often too late to be of use. Additionally, attackers use so-called cloaking techniques that redirect known crawlers to benign websites. Honeyclients also suffer from a number of other debilitating problems (as discussed later in more detail). For example, honeyclients are less effective if their system configuration does not match that of the targeted victim (*e.g.*, an exploit targeting Internet Explorer 11 will not be detected if the honeyclient is configured with Internet Explorer 10). Finally, honeyclients are notorious for requiring non-trivial amounts of time to complete a single analysis — easily on the order of minutes. For our purposes, such prohibitively long processing times make them poorly suited for live operational deployments. Indeed, Adobe Flash vulnerabilities have dominated other attack vectors in the last two years, but remain difficult to analyze dynamically due to the sheer volume of Flash files, exceeding hundreds of files per minute on our campus network, for example.

Motivated by a *real operational need* to tackle the threats posed by the significant rise in Flash-based attacks, we present

NDSS '16, San Diego, CA, USA.

http://dx.doi.org/10.14722/ndss.2016.23427

a framework that enables one to adapt an arbitrary honeyclient system to function on-the-wire by minimizing the impact of the aforementioned drawbacks. The approach described in this paper detects exploits by temporarily *caching* web traffic, *triggering* an analysis on a previously unseen exploitable file, *impersonating* the client and server that fulfilled the request, and *replaying* the traffic in a honeyclient to detect any malicious behavior. One major operational challenge we face is that the analysis we perform must be done without any human intervention and without storing personal information on non-volatile storage. These privacy restrictions are not unique to our environment, and it means that we (like many others) are left with no option but to process the fire hose of network data judiciously and expeditiously. Thankfully, we are able to leverage a few minutes of recently seen network traffic stored in an in-memory cache. A second major operational challenge is that many web-based exploit files (*e.g.*, Flash) will only elicit malicious behavior if the proper parameters are passed in by the loading website. As a result, we must provide the proper context in order to detect these files.

In designing, deploying and evaluating this framework, we overcame several obstacles and make the following contributions that we believe will be of value to the greater networking and security community:

- A network-based exploit kit detector that uses behavioral analysis to detect malicious exploits in the context of the websites that load them.
- A new fuzzy-hash based technique for filtering redundant exploitable *trigger* files, allowing for a scalable and online honeyclient behavioral analysis.
- A two-level semantic cache for storing and compressing HTTP network traffic based on URLs requested.
- A novel chaining algorithm that traces web exploit requests back to their origin by storing minutes worth of network traffic, replaying URL request paths, and impersonating both the client and server in order to coax the exploit into behaving maliciously.
- A set of recommendations for an improved honeyclient system based in part on the identification of code injection and code reuse payloads used in an exploit as well as a set of behavioral features.
- A case study that highlights recent trends in deployed exploit kits.

The remainder of the paper is organized as follows. We present background information and related work in §II. Our framework for enabling the use of honeyclients on-the-wire is presented in §III. We provide a performance evaluation, as well as a case study of real-world attacks, in §IV. Limitations and future work are discussed in §VI. We conclude in §VII.

II. Related Work

Over the past decade, the web has become a dominant communication channel, and its popularity has fueled the rise of web-based infections. Provos et al. [25] examined the ways in which different web page components are used to exploit web browsers and infect clients through drive-by downloads. That study was later extended [26] to include an understanding of large-scale infrastructures of malware delivery networks and showed that ad syndication significantly contributed to the distribution of drive-by downloads. Grier et al. [10] studied the emergence of the exploit-as-a-service model for drive-by browser compromise and found that many of the most prominent families of malware are propagated from a handful of exploit kit flavors. Thomas et al. [34] provide a more thorough analysis of prevalence of ad injection and highlight several techniques being deployed by ad injectors.

By far the most popular approach to detecting malicious websites involves crawling the web for malicious content starting from a set of known malicious websites [11, 15, 16, 8, 34]. The crawled websites are verified using statistical analysis techniques [15] or by deploying honeyclients in VMs to monitor environment changes [26]. Other approaches include the use of a PageRank algorithm to rank the "maliciousness" of crawled sites [16] and the use of mutual information to detect similarities among content-based features derived from malicious websites [37]. Eshete and Venkatakrishnan [8] identified content and structural features using samples of 38 exploit kits to build a set of classifiers that analyze URLs by visiting them through a honeyclient. These approaches require massive cloud infrastructure to comb the Internet at scale, and are susceptible to cloaking and versioning issues [36].

Gassen and Chapman [9] examine Java JARs directly by running applets in a virtualized environment using an instrumented Java virtual machine looking for specific API calls and behaviors such as file system accesses. Since the approach analyzes JAR files in isolation, it is unable to detect malfeasance when parameters are passed into the applet. Other approaches involve analyzing the source code of exploit kits to understand their behavior. For example, De Maio et al. [7] studied 50 kits to understand the conditions which triggered redirections to certain exploits. Such information can be leveraged for drive-by download detection. Stock et al. [32] clustered exploit kit samples to build host-based signatures for anti-virus engines and web browsers.

More germane to our own work are approaches that try to detect malicious websites using HTTP traffic. For example, Cova et al. [6] designed a system to instrument JavaScript runtime environments to detect malicious code execution, while Rieck et al. [27] described an online approach that extracts all code snippets from web pages and loads them into a JavaScript sandbox for inspection. Unfortunately, parsing and executing all JavaScript that crosses the boundary of a large network is not scalable without some mechanism for pre-filtering all the noise produced by benign scripts. Further, simply executing JavaScript without interfacing with the surrounding context, such as relevant HTML and other intertwined contents, makes evading such systems trivial. Our approach addresses both of these issues.

Several approaches utilize statistical machine learning techniques to detect malicious pages by training a classifier with malicious samples and analyzing traffic in a network environment [27, 5, 4, 18, 19, 21, 22]. More comprehensive techniques focus on extracting JavaScript elements that are heavily obfuscated or `iframes` that link to known malicious sites [25, 6]. Cova et al. [6], Stringhini et al. [33], and Mekky et al. [21] note that malicious websites often require a number

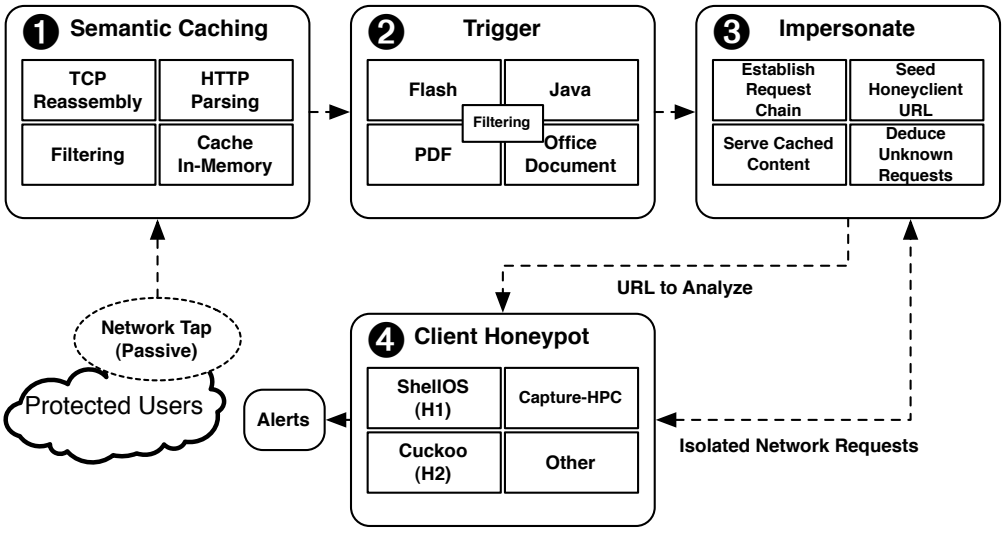


Fig. 1: Overall workflow of enabling an on-the-wire honeyclient.

of redirections, and build a set of features around that fact. Nelms et al. [22] studies the webpaths users take to malware downloads and builds a classifier to label them in the wild. Canali et al. [5] describes a static prefilter based on HTML, JavaScript, URL and host features while Ma et al. [18, 19] use mainly URL characteristics to identify malicious sites. Some of these approaches are used as pre-filter steps to eliminate likely benign websites from further dynamic analysis [26, 25, 5]. Unfortunately, these techniques take broad strokes in terms of specifying suspicious activity. As a result, Provos et al. [26] reported a 10% false negative rate and Canali et al. [5] reported a false positive rate of between 5% and 25%, while Provos et al. [25] only disclose that using obfuscated JavaScript as an indicator leads to a high number of false positives. These works also require large training sets that are not generally available. By contrast, our approach focuses on *behavioral* aspects of malware to help reduce false positives and false negatives.

Schlumberger et al. [29] extracts features related to code obfuscation and the use of Java API calls known to be vulnerable, then detects malicious applets using machine learning. Likewise, Van Overveldt et al. [35] instruments an open source Flash player and extracts similar features to detect malicious ActionScript. While these techniques are dynamically adaptable due to their use of machine learning, they still require a priori notions of how malicious code is constructed. For example, Van Overveldt et al. [35] implements features that are meant to determine whether code or data obfuscation has been used, and whether known vulnerable functions have been used. Intuitively, a previously unknown vulnerability, *i.e.*, a zero-day attack, present in an unobfuscated Flash file will not be detected. Additionally, highly obfuscated Flash exploits wherein the obfuscation itself is the only available feature cannot be reliably detected with this approach without false positives (2% in [35]) since obfuscation is commonly used by benign files. In contrast, our approach does not use obfuscation or known vulnerable functions to make a final decision, thus we have a lower false positive rate.

Finally, by far the most popular means of network protection are NIDS, such as Bro [23] or Snort [28], that passively monitor networks and apply content-based signatures to packets and sessions in order to detect malfeasance. These signatures are lightweight, but are evaded through the use of obfuscation and morphing techniques commonly utilized by attackers. They also are not effective against zero-day attacks. To help with forensic analysis, Maier et al. [20] extended Bro with time machine, a lightweight data store for packets, so that Bro could retrospectively query packets by their headers to perform further analysis on interesting events. Time machine has similar goals to our caching and replay mechanism; however, they attempt to achieve this goal at the network layer, storing up to N bytes per connection tuple in a packet trace. In contrast, our approach operates at the application layer by storing reconstructed web objects. For HTTP, this application layer approach achieves much greater compression, as a small number of unique web objects are frequently fetched by users (*e.g.*, Facebook, Google).

We argue that our framework provides the best of both worlds between statistical approaches and honeyclients by bringing the honeyclient to the network. As a result, we can identify new exploits on-the-fly and mitigate threats more swiftly than the current state of the art.

III. Our Approach

In short, our goals are to combine on-the-wire monitoring of network with the use of honeyclients in an attempt to address real-world challenges faced on a large network. We conjecture that such a combination significantly outperforms content-based signature approaches in terms of detection rates, and moreover, can be designed and implemented in a scalable manner. Working at scale, however, comes with several pragmatic challenges that must be addressed. For one, honeyclients are notoriously slow in analysis; therefore, we require mechanisms to drastically reduce the amount of traffic analyzed, but without basing these mechanisms on preconceived notions

as to the innocuity of the traffic in question. Other practical concerns involve finding robust ways to decide what contextual environment should be used for analyzing a potentially malicious event triggered by our framework. The high-level depiction of our workflow is given in Figure 1.

Intuitively, HTTP traffic is monitored at the network border or within an HTTP Proxy. In step ❶, a collector reassembles TCP sessions into bidirectional HTTP requests and corresponding responses. HTTP objects are extracted and cached in a two-level semantic cache. In step ❷, those objects that represent attack vectors (*e.g.*, Flash, PDF, Java, Silverlight) trigger additional analysis. In step ❸, our chaining algorithm selects the initial URL to be loaded by the honeyclient. Finally, in step ❹, the honeyclient transparently queries the two-level cache and monitors various system events to provide detection. In what follows, we discuss the challenges and solutions we provide for each component in our design.

A. Step ❶: Semantic Content Caching

The state-of-the-art application of honeyclient analysis requires that operators provide a seed list of URLs to the honeyclient, which in turn fetches each live URL within the analysis environment. Operating on-the-wire, however, we can not afford this luxury. Moreover, for privacy reasons, we can not simply log URLs observed on the network and use these URLs as the seed list; such URLs may contain end-user information embedded with parameters that instruct remote servers to perform some action such as purchasing items, posting written content, or verifying some event or action. Thus, we are left with no option but to perform *in-memory processing* of the fire hose of request content that enters the network, without human intervention or saving of information to non-volatile storage. We can, however, rely on a short window of time (*e.g.*, on the order of minutes) where recent browsing activity is retained in caches that can be queried.

In our approach, we opt for caching observed content at the application layer rather than at the network layer as proposed by Maier et al. [20]. As packets cross the network border, we reassemble them first at the TCP-level into matching {*request, response*} data streams. Duplicate or malformed TCP packets are discarded as specified by the TCP protocol. Then we reassemble these data streams at the HTTP-level, making each request header and associated response content transparent to our framework. As with TCP packets, malformed HTTP content is discarded in accordance with the protocol specification, and content for other application-layer services is filtered and ignored. Web objects (*e.g.*, HTML, JavaScript, Images, Flash, Java, etc.) are then extracted from the reassembled streams. Object types are determined by using a combination of the HTTP `Content-Type` header, the file extension specified in the URL, and the first 512 bytes of the payload (*i.e.*, the "file magic"). These objects are then placed in a two-level semantic cache to later be (potentially) queried by the chaining and honeyclient phases of the process (step ❹).

The key observation we made in designing our application-layer, two-level, semantic cache is that a significant percentage of network traffic is, in fact, identical content served from a few popular web sites (*e.g.*, Google, Facebook, YouTube). Thus, such a cache is capable of compressing data much more efficiently than at the network layer where each packet of data is more likely to be unique with client address information and different patterns of TCP and HTTP chunking. The first level of our cache is for web objects that are cacheable network wide – *i.e.*, objects that do not change frequently between client web requests. This cache works similar to a web proxy cache and caches objects using the `Expires` and `Max-Age` HTTP response headers and is implemented based on the web caching RFC 7234. We use a *least recently used* (LRU) caching data structure to hold these objects until they either expire, or are evicted because the cache is full. Globally cached web objects are stored on disk in order maintain the cache between application runs.

There are many objects that are not cacheable network wide because they provide dynamic content such as a personalized landing page on a social networking web site. As a result, these objects are stored in individual client-level caches keyed by IP address in volatile memory. This second level is an LRU cache composed of LRU caches, where client IP addresses are evicted after a tunable period of inactivity. The cache holds a tunable maximum of N client IPs by M objects to manage memory consumption. We revisit the effect these parameters have on memory consumption and the achievable throughput of our framework in §IV.

We later discuss how this cache is utilized for honeyclients in §III-C, but for now turn our attention to how one can use this information to hone in on potentially malicious web traffic in an overwhelmingly benign sea of traffic flows.

B. Step ❷: Filtering and Triggering

One significant challenge in the design of our framework lies in the ability to scale to provide a timely analysis of each observed request. Indeed, honeyclient analyses typically require on the order of minutes to complete depending on the specific techniques employed. Furthermore, large networks may observe on the order of thousands of requests per second. Our framework addresses this problem by selectively analyzing only specific types of requests — *those that eventually lead to the download of a commonly exploited file format* — and then we additionally filter those analyses using a file format specific mechanism.

To guide our efforts in designing file format specific filters, we measured the observed downloads on our campus network over the course of a single school day (see section §IV). Only JavaScript, Flash and Portable Document Format (`pdf`) exceeded an average of one observation per minute. Executable, Java and Silverlight file formats proved to be relatively rare and hence we do not design filters for these formats, as it is unnecessary. We observed an average of 7.4 `pdf` files a minute. Fortunately, filtering based on unique file content hashes alone drops the number of `pdf` files requiring analysis to less than one per minute, which can be easily handled by a stock version of ShellOS [30].

Unfortunately, the same cannot be said for JavaScript files. We observed a staggering 3,628 JavaScript files (on average) per minute with peak rates of over 8,000 per minute. Parsing the content of all these scripts in an effort to design an appropriate filter results in packet loss in the HTTP parsing phase of semantic caching. Hence, we believe a potential route

to filtering JavaScipt is to leverage meta-data for each script, such as the source IP and domain combined with a reputation-based approach [2]. Given the current challenges in analyzing Flash, we leave JavaScript filtering as future work.

As noted earlier, we observed hundreds of Flash objects per minute; large enough to require filtering, but not so large that the mere act of parsing them all causes packet loss. Hence, an additional filtering mechanism was required to reduce the overall number of Flash files analyzed. The academic literature offers a few options that we considered. For instance, Ma et al. [18] use URL features to classify requests as malicious, while Cova et al. [6] uses code obfuscation, specific API calls, and number of `iframes` as features. These features are effective, but fall short when a new zero-day exploit surfaces that is not in line with the predefined feature set. In short, existing approaches for filtering Flash files take a *blacklisting* approach, that unfortunately, are evaded during the period of time when attackers exploit a new vulnerability without giving those systems other hints of their malicious intent (*e.g.*, such as multiple layers of obfuscation). We return to that discussion later in §IV.

Instead, we opted for a *whitelisting* approach in line with our goal of using honeyclients to detect previously unseen, or zero day, attacks. Our approach, which is based on file popularity, does not make the same assumptions about feature sets as in prior work. The key insight is that the vast majority of Flash files seen on a network are from advertising networks that utilize a relatively few number of unique Flash files to display ads. These ads also flow along the network in a bursty pattern as a web page will typically load multiple advertisements.

Given these insights, we make use of two filters. The first filter takes a 16-byte hash of each Flash file and checks a key-value store of known popular Flash hashes. If the hash appears in the data store it is not analyzed. This basic check eliminates the need to analyze ads wherein the Flash files themselves are identical, but they serve different ad content through the use of different parameters supplied to those files. On the other hand, some ads have their content directly built into the Flash file itself. Our approach to handling this second type of ad is more involved. More specifically, we make the simplifying assumption that a small number of libraries are in use and that some subset of that code is used in each Flash file. Given that assumption, we parse Flash files observed on the network and extract individual function byte-code. We hash the byte-code at the function level to create a piecewise or fuzzy hash [14]. Then, for each new Flash file we only trigger an analysis if it has at least one function that is not in our function-level hash store. If an attacker attempts to masquerade their Flash exploit as a benign ad, we still trigger an analysis based on the fact that some new code must be added to exploit a vulnerability.

Using these filters, the average number of Flash files analyzed per minute drops to less than 10 (from over 100 observed per minute). Even so, Flash offers some interesting challenges, and so to focus our presentation, we center on an in-depth analysis of Flash exploits in §IV. At this point we have a cache of web objects and a desire to perform a honeyclient analysis based on the observation of a potentially malicious Flash file. We now turn our attention to the details of how all the information collected up to this point comes together to "replay" content for honeyclient analysis without ever contacting live exploit kit servers.

C. Step ❸: Client and Server Impersonation

Given some recently observed network traffic containing the interaction of a client and server, the immediate goal at this stage in the overall architecture is to provide an environment in which we can observe client system state changes, *e.g.*, to enable honeyclient analysis. The central challenge is to do so without further interaction with either the client or the server. The observant reader would note, however, that one can rarely analyze a web-based exploit file like Flash in isolation. This is due to the fact that the surrounding context of HTML and JavaScript provide requisite input parameters that enable the exploit to successfully operate. To overcome this obstacle, we recreate that context and replicate client and server configuration based on the previously observed information in the interaction between the client and server.

Client Impersonation: On the client-side there are two primary challenges: (1) replicating client system configuration and (2) determining the originating HTTP request that resulted in the chain of requests leading up to exploit file. To tackle the former challenge, our framework implements an independent *network oracle* that collects browser and plugin information about every client on the network. Collecting client browser information is a popular activity for attackers [1], which we turn into a valuable resource for our own purpose. Due to data collection limitations on our campus network, we are limited to collecting browser information through the `User-Agent` and `X-Flash-Version` fields of HTTP requests, which provides browser, OS and Flash versioning information. In corporate enterprise networks, one can use more sophisticated collection techniques using JavaScript [1]. Nevertheless, our empirical results show that even such limited information provides enough detail to assist with the dynamic configuration of honeyclients to allow them to be successfully exploited.

Tackling the latter client-side challenge turned out to be far more involved. One reason is because a client may have multiple web browser tabs open loading multiple web pages, or a single page loading several other web pages that do not lead to the observed exploit file. To resolve the originating web page of an exploit file we introduce a new algorithm, dubbed the *chaining algorithm* (Algorithm 1), that operates as follows. First, during the two-level caching step of our workflow (step ❶ §III-A), the URL from each cached object is timestamped and stored in a list keyed by the corresponding client's IP address. Only URLs that represent HTML documents are added to the list. When a web object (*e.g.*, Flash file) triggers an analysis, the URL list for the corresponding client IP address is traversed, and request URLs that are within a tunable time threshold are sent to the next step.

Next, Algorithm 1 iterates through each request URL in the list, and loads them one-by-one into an instrumented headless browser (lines 8–20) given the client's browser and IP address information. A headless browser is a web browser without any graphical user interface that allows rapid HTML parsing and JavaScript execution without the overhead of an entire virtual environment. The headless browser uses the two-level semantic cache as a proxy to request corresponding web resources. It

Algorithm 1 The chaining algorithm searches for the root web page that loads the trigger to be analyzed in the honeyclient.

```
1: URLList ← List of URLs within timing threshold of trigger.
2: TriggerURL ← URL of target trigger object.
3: ProxyAddr ← URL of web cache.
4: ClientConfig ← Client's browser information.
5: browser ← HeadlessBrowser(ClientConfig, ProxyAddr)
6: CurrentBestMatch ← ⊥
7: BestMatchURL ← ⊥
8: for all ( do Url ← URLList)
9:     ObjectTags ← browser.SearchForObjectTags(Url)
10:    Match ← FindTriggerInTags(TriggerURL, ObjectTags)
11:    if Match == EXACT_MATCH then
12:        CurrentBestMatch ← Match
13:        BestMatchURL ← Url
14:        BREAK
15:    end if
16:    if Match > CurrentBestMatch then
17:        BestMatchURL ← Url
18:        CurrentBestMatch ← Match
19:    end if
20: end for
21: if CurrentBestMatch ≠ ⊥ then
22:    SubmitToHoneyClient(ClientConfig, BestMatchURL)
23: end if
```

parses web content and executes any JavaScript searching for `object`, `applet`, and `embedded` HTML tags (line 9) that are used to load Flash, Java, and Silverlight files. These tags are scanned for absolute and relative references to the exploit file URL (line 10). If the exploit file reference is found in these tags, the request URL is selected as the originating request (lines 10-15). Where available, the triggering web object's referrer can be used to prioritize URL selections for the algorithm.

If no URL leads to an exact match, then the best near-match or potentially malicious match is selected as the originator. We determine near matches through domain, or by domain and path. A potentially malicious match is determined through observed JavaScript behavior, including checks for anti-virus plugins, accesses to known exploitable APIs, or attempts to load files on the local hard drive (see §V, for example).

One of the major challenges in our approach is that client browser caches can store highly cacheable web objects, such as JavaScript, for days or months. As a result, the network monitor may not see all requested web objects during the course of analysis. In order to deal with this situation, the web cache acts as a proxy, retrieving web objects known to be JavaScript and caching them. All proxy requests are sanitized of any client personal information.

It is prudent to note that there are cases where a single chain of HTML resources can lead to multiple Flash files. Thus, before sending a URL list to the chaining algorithm for analysis, the network monitor waits several seconds to allow other Flash files to be cached. Each Flash file is then sent with its corresponding URL list to the chaining algorithm for analysis. A request URL is only scanned once, and if it is found to lead to multiple Flash files the remaining chains associated with those files are not re-executed. The honeyclient uses the request URL to load all Flash files and analyzes them all at once (line 22).

Server Impersonation: The most significant challenge with respect to impersonating the server-side of the connection is that it is the headless browser and honeyclient—not the original network client—that makes the web requests to the web cache. As a result, we must pass the client IP to the web cache along with the URL. This is done by encoding the client IP into the URL of the initial web request before passing it to the honeyclient. The web cache decodes the URL, extracts the client IP, and maps the address to the honeyclient's IP to handle subsequent related web requests. Next, the web cache uses the URL to check the network-wide cache. If the URL is not present, the client-level cache is checked. If no web object is found, a 204 status code is returned.

Lastly, web objects are cached with their original HTTP headers. However, since objects are reassembled and decompressed in the cache, some header information (*e.g.*, `Transfer-Encoding`) is deleted or altered (*e.g.*, `Content-Length`) before being served to the client.

D. Step ❹: Honeyclient-based Detection

Once a URL is selected for analysis in step ❸, the associated client IP is encoded into the URL and the new URL is sent to a honeyclient. In this context, we define a honeyclient as any software posing as a client that interacts with a server with the goal of determining whether that server is malicious. The framework is designed to be modular allowing for any honeyclient that supports interacting with a proxy server.

Our experiments in §IV make use of unmodified versions of Cuckoo Sandbox[1] and ShellOS [30, 31]. We chose these two approaches due to the fact that they collect very different metrics and have different runtime requirements. Specifically, ShellOS analyzes a virtualized environment for evidence of injected code (or shellcode) by executing potential instruction sequences from an application memory snapshot directly on the CPU. Thus, ShellOS monitors the programmatic behaviors of a malicious payload. ShellOS labels a sample as malicious if any of the following are true:

- The process memory contains a code injection or code reuse payload.
- The process memory exceeds a tunable threshold (500MB in our analysis), *e.g.*, a heap spray is likely to have occurred.
- The process terminates or crashes.

By contrast, Cuckoo monitors changes to a virtualized environment primarily by API hooking. API hooking is the process of intercepting function calls, messages, and events in order to understand application behaviors. We use Cuckoo Sandbox to label a sample as malicious if any of the following is true:

- The process uses known anti-detection techniques.
- The process spawns a another process.
- The process downloads an exe or dll file.
- The process accesses registry or system files.

[1] http://www.cuckoosandbox.org/

- Network traffic contacts non-application related hosts.
- The process accesses potentially sensitive information in the browser process.
- The process modifies system security settings.

In order to separate the honeyclient approaches from their specific implementations, we refer to ShellOS as H_1 and Cuckoo as H_2 in §IV. Our evaluation shows that monitoring system state with either of these approaches significantly improves detection performance over content-based signatures.

E. Prototype Implementation

Our prototype implementation consists of 8192 lines of custom C/C++, Java and Golang code. The *libnids* library provides TCP reassembly. We implemented a Go IO reader interface for *libnids* to adapt Go's in-built HTTP request and response parsing to captured network traffic. The resulting HTTP objects are stored using a multi-tiered hash map keyed by client IP address and the URL requested, as described in §III-A. The global web cache and Flash filters are stored in the *rocksdb* key-value store, while triggers are implemented with a combination of both response MIME-type and the "file magic" indicating a file type of interest.

The sheer volume of Flash requests observed on our campus network necessitated filtering for Flash file triggers, as described in §III-B. Our Flash parsing and fuzzy hashing is all custom code written in Go, as is the implementation that impersonates the attack server. For our headless browser, we use HTMLUnit[2], an open source implementation written in Java that incorporates the Rhino JavaScript Engine. HTMLUnit can mimic Internet Explorer, Firefox and Chrome and is controllable programmatically. Furthermore, the browser is extensible allowing for the addition of customized plugins and ActiveX objects to simulate various versions of Java, Flash, and Silverlight. Framework modules communicate with one another using a web-based REST messaging service in addition to *Redis*, a key-value cache and store.

IV. Evaluation

To demonstrate the efficacy of our framework we conducted both an offline evaluation with known exploit kit traces and an online analysis on a large campus network. In short, our findings suggest that on-the-wire honeyclients consistently out-perform signature-based systems by discovering exploited clients days and weeks ahead of those systems. We also show that a single on-the-wire honeyclient server is capable of keeping pace with a large campus network at its boundary.

The evaluation focuses on Flash files as triggers due to the sheer volume of Flash on the network (see Table I). File types such PDF and EXE are typically self contained and can be analyzed directly within a sandbox without loading a full website [30]. Like Flash, Silverlight and JAR files both require the context of the loading website. With all the recent Java security vulnerabilities, Java is disabled in all browsers requiring the user to directly allow a class or JAR file to run — our framework does not support user interaction. Finally, as shown in Table I, both Java and Silverlight are seen in such low numbers that they do not pose the same operational challenges as Flash and are thus not considered further.

JavaScript is one of the most (Table I) prevalent web objects on a network, and as such, presents significant scalability challenges. While we do not address JavaScript-only drive-by-download attacks in this paper, we can detect malicious JavaScript that is used to load a trigger file (*e.g.*, Flash). Furthermore, the lessons learned from analyzing Flash will be invaluable in our future work on full scale JavaScript analysis.

Silverlight	108
JAR	322
EXE	871
PDF	10,637
Flash	97,576
JavaScript	5,224,412

TABLE I: Number of instances of various file types seen on campus on a busy school day.

A. On Detection Performance

Experiments in this section are conducted on a Dell Optiplex desktop with a 4 core i7-2600 CPU at 3.40GHz and 16GB RAM. Two different honeyclients are used for each sample – H_1 and H_2 – as described in the previous section, with their default installations using Qemu and Virtual Box virtual machines, respectively, on Ubuntu Linux 14.04 64-bit. The analysis time for H_1 is set to 30 seconds, while H_2's timeout is 5 minutes. Each honeyclient uses the same VM configuration – Windows 7 32-bit, either Internet Explorer (IE) 8 or IE 10, and one of 8 different versions of Adobe Flash Player configured dynamically based on information retrieved from the network oracle (see section §III-C). Honeyclient results are then contrasted to the results of 50 antivirus engines[3].

We inspected 177 HTTP publicly available packet trace samples of exploit kits[4]. Each trace represents a packet recording of all HTTP traffic between a Windows 7 virtual machine and a real-world website known to be injected with an exploit kit landing page, typically through an injected `iframe`. Over a year of traces were collected between April 2014 and June 2015 representing successful exploits from 10 unique exploit kit flavors that evolved over this one year period. Thus, our dataset is representative of the diversity of real-world attacks that would be encountered if our framework were to be deployed on any large network.

On-the-wire Performance of Honeyclient H_1: Table II shows the evaluation results for our framework using H_1 with a breakdown of how each exploit kit is detected. In all cases, the exploit file and originating request URL are identified (step ❷) and forwarded to the honeyclient for inspection (step ❹). Overall, this configuration has a 92% true positive rate. The vast majority of detections are from code injection payloads in process memory, suggesting that the use of code injection payloads is still a prominent means of exploitation, despite a multitude of commonly deployed endpoint defenses. The missed detections result from exploits that do not make

[2]Available for download at http://htmlunit.sourceforge.net/

[3]Using analysis available at http://www.virustotal.com

[4]Samples available at http://www.malware-traffic-analysis.net

Exploit Kit	Uses Payload	Crashes	Heapsprays	Terminates	Misses	Total Detections	Total Instances
Nuclear	24	0	1	1	3	25	28
Angler	32	1	0	0	0	33	33
Magnitude	4	2	1	0	1	6	7
Sweet Orange	21	0	0	0	0	21	21
RIG	16	8	0	2	0	18	18
Neutrino	9	1	2	0	0	9	9
Fiesta	28	1	0	0	9	29	38
Null Hole	1	1	0	0	0	1	1
Flashpack	7	8	1	1	1	12	13
Infinity	5	0	0	4	0	9	9
	147	**22**	**5**	**8**	**14**	**163**	**177**

TABLE II: Detection results for our framework when using honeyclient H_1 on the 10 exploit kits by detection type.

Exploit Kit	Process Launch	File Drop	Browser Crash	File Access	Misses	Total Detections	Total Instances
Nuclear	3	1	5	5	14	14	28
Angler	0	0	4	20	9	24	33
Magnitude	2	0	0	0	5	2	7
Sweet Orange	2	0	0	1	18	3	21
RIG	3	0	7	0	8	10	18
Neutrino	2	0	0	0	7	2	9
Fiesta	26	26	0	0	12	26	38
Null Hole	0	0	1	0	0	1	1
Flashpack	5	0	5	0	3	10	13
Infinity	2	1	5	0	1	8	9
	45	**28**	**27**	**26**	**77**	**100**	**177**

TABLE III: Detection results for our framework when using honeyclient H_2 on the 10 exploit kits by detection type.

use of traditional code injection. Rather, they use a memory disclosure vulnerability to leak system API addresses and then dynamically construct the injected code using this information. As a result, the so-called `PEB` heuristic [24] used by H_1, which identifies the API address lookups of injected code, is never triggered. H_2, on the other hand, uses a disjoint set of features such as monitoring file drops, process launches, and registry and file accesses through function-level hooking.

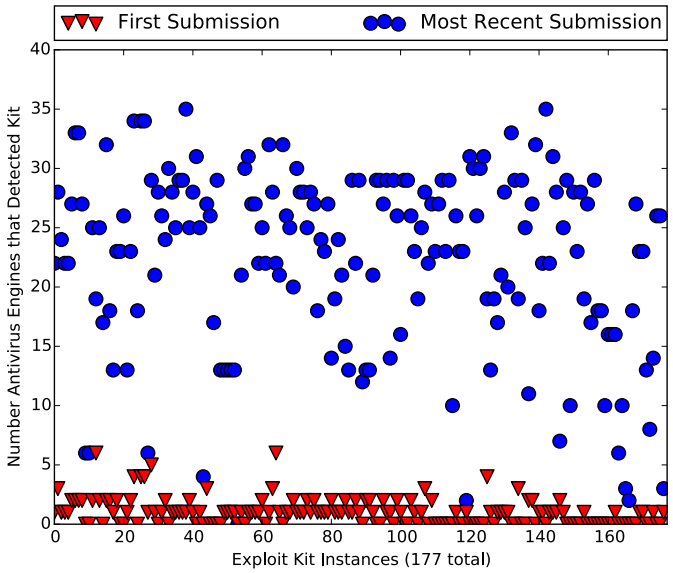
On-the-wire Performance of H_2: The results when using H_2 with our framework are shown in Table III. This configuration only resulted in a 56% true positive rate. One reason for this lower detection rate is that browser-based analysis is a relatively new feature in H_2 and IE 10 is not fully supported at the time of writing this paper. Digging deeper into the remaining missed detections, we found that the exploits are *unhooking* four Windows API calls (details in Section V) that are used by attackers to determine whether they are operating in a virtualized environment. In short, the exploits use injected code to first remove H_2's hooks, then call those APIs to determine if the system is virtualized. Attacks immediately cease when a virtualized environment is detected in these samples. Nevertheless, H_2's heuristics are still useful for exploit detection. For example, H_2 is able to detect the 14 exploit kits that H_1 misses by observing accesses to the filesystem, process launches and file downloads.

The results of our evaluation indicate that injected code detection is a robust feature for determining maliciousness. It is used by 83% of exploits, and does not require successful exploitation for detection. For example, exploits using injected code to detect virtualization are detected by H_1 even if they decide not to compromise the system. However, H_1 cannot handle virtualization checks that are done through JavaScript-based filesystem checks (§V) prior to constructing or unpacking the injected code. Indeed, Angler would have been undetectable by H_1 had it checked for files related to QEMU prior to unpacking the code injection payload. As a result, H_2's file and registry access hooks, as well as environmental change detection, are equally important. Using all features from both honeyclients enables the framework to achieve a 100% true positive rate. Even so, it may be possible for attacks to evade these honeyclients by combining unique methods of unhooking functions with injected code that does not perform API lookups.

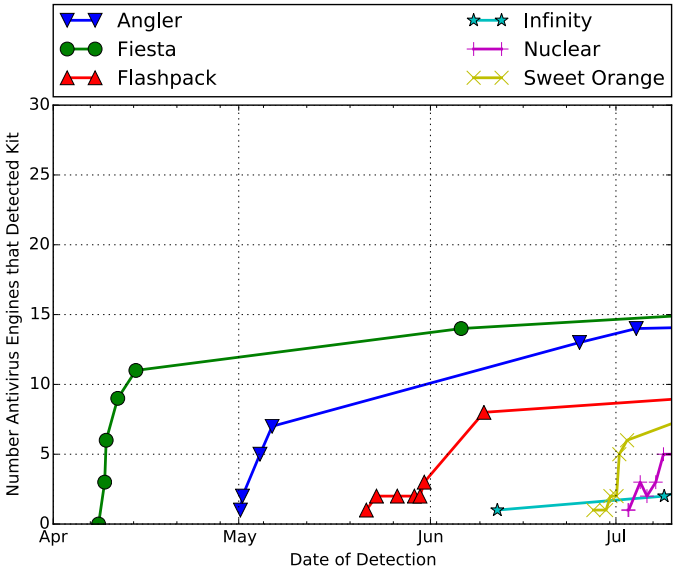
We reiterate that the design and implementation of specific honeyclient technologies is an ongoing research topic, but the primary goal of our work is to provide a framework that effectively leverages such advancements on-the-wire. To that end, these experiments confirm the efficacy of our approach by providing honeyclients H_1 and H_2 with all relevant information needed to replay and reproduce the attacks. Indeed, our framework achieves a 100% success rate in this context.

Content-based Signature Comparison: Next, we compare the performance of honeyclients using our framework with that of content-based signatures, *e.g.*, antivirus engines. We checked each exploit file associated with all 177 HTTP traces against 50 signature engines and found that on average 50% of these engines labeled the exploit file as malicious[5]. One could argue that perhaps some of these engines perform better than others and, indeed, three of the engines detect *all* of the given exploit files, *e.g.*, 100% true positive rate. However, we argue that such a comparison against a honeyclient is biased and incorrect in practice – The honeyclients operate with only the general knowledge of the behaviors they observe as they occur while content-based signature engines update their knowledge base per each newly observed malicious file. Indeed, there is little value in a system that does not detect a malicious file at the time it is used to attack one's network. We hypothesize that signature engine performance is significantly

[5]Note that some of these engines also incorporate a heuristic approach in their determination.



(a) Each exploit kit instance is represented by a point on the x-axis. The y-axis indicates how many signature-based engines detected an instance for the first and most recent submissions.



(b) Comparing detection rates of 6 Flash exploit instances over time.

Fig. 2: Analysis of the 177 exploits on VirusTotal.

worse than our on-the-wire honeyclient when comparing it to a signature engine using only those signatures available *at the time of the attack*.

Indeed, our experiments confirm the aforementioned hypothesis. The results of this analysis are depicted in Figure 2. Figure 2a shows that at initial attack time, 69 of the exploits go completely undetected by all engines. In other words, the best engine has no more than a 61% true positive rate. Another 70 are only detected by a single engine, meaning that 98% of engines have no better than a 21% true positive rate. More unsettling is that two different instances of the same exploit kit found a year apart still leads to at most 3 signature-engine detections. Thus, finding a single instance of an exploit file does not appear useful for these engines in finding newer exploit files from the same exploit kit, unless the files are exactly the same.

Another concerning revelation is how long it takes for signature-based engines to detect exploits after initial observation. We randomly selected six exploit kit instances from the sample set and analyzed how many engines detected the instance over time starting from the initial observation to the last, as seen in Figure 2b. In the case of Angler, Flashpack, Nuclear and Sweet Orange, 3 to 10 days passed before only 5 engines are able to detect the exploit. For Infinity, a month elapsed before signatures were distributed for each exploit instance. Unfortunately, with the rapidly moving and morphing nature of these kits, the instances are no longer active on the Internet by the time content-based signature engines have a rules to detect them. By contrast, honeyclients have no preconceived notions about what is malicious, but rather execute new files in a dynamic environment and monitor system state change and the factors described in section §III-D. As a result, our framework detects attacks on-the-wire when it matters – as they happen.

In summary, the use of H_1 and H_2 with our framework detects 100% of attacks in our diverse sample set, while the combination of 50 signature-based engines achieves 61% detection. Next, we present the results of live-testing on-the-wire and report on false positives.

B. On Live Traffic Analysis

We now turn our attention to detection in the face of significant background traffic. That is, experiments in this section demonstrate that our framework can successfully detect exploits from the larger haystack of benign traffic while maintaining a negligible false positive rate. To that end, we ran our framework on a campus network for a 5 day period in November 2015. The University has over 25,000 students, faculty and staff with an average network throughput of 1 Gbps (in the summer) and 7Gbps (during the school year) on a 10 Gbps link. Our tap utilizes an EndaceDAG data capture card on a Dell R410 rack-mounted server with 128 GB RAM and three 8-core Xeon 2100 CPUs. Furthermore, we used the H_1 honeyclient running with five VMs, allowing us to run five concurrent analyses supporting Chrome, Internet Explorer and Firefox browsers. The online analysis focuses on H_1 because we developed the platform and could easily modify it to support multiple browsers and Flash plugins, while debugging any load related issues. While we realize that not using H_2 will affect the overall detection rates, we believe our setup sufficiently demonstrates the utility of the approach in a live environment. Integrating the feature sets of H_1 and H_2 is left for future work.

On Flash Filters: Before we can run our online test, we must establish the Flash filters. To do so, we investigated the Flash file download patterns of the university network by monitoring the network for a three day period in July. We collected Flash file hashes, piecewise hashes (described in section III-B), and requested URLs.

Over 270,000 Flash files were downloaded by network clients, as shown in Figure 3. We observed that the ad-

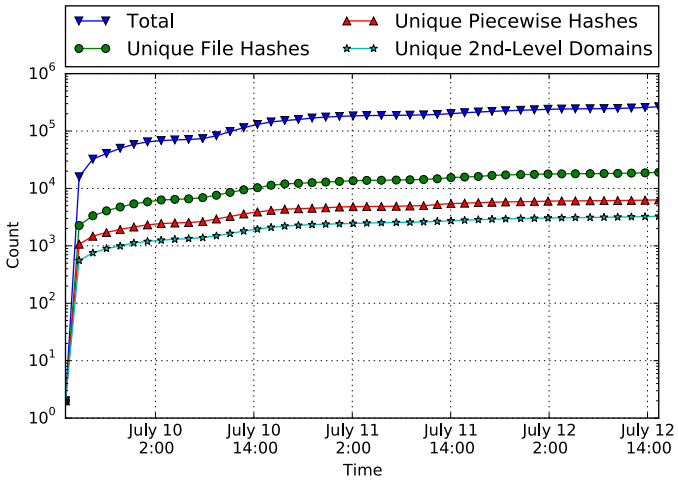


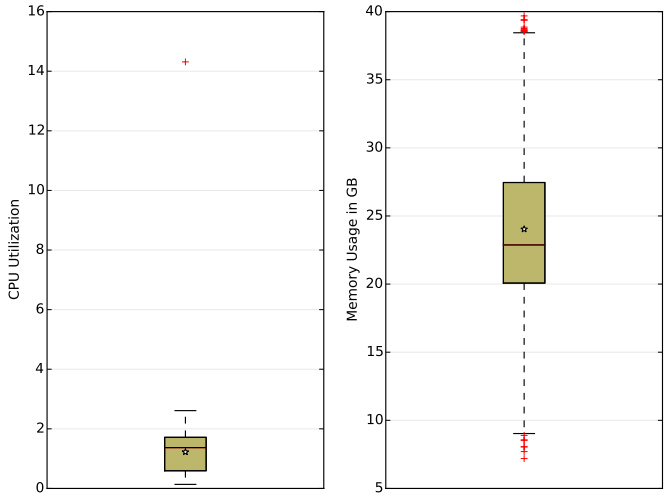
Fig. 3: The number of unique 2nd-level domain names, Flash files, and Piecewise hashes seen on the network.

related domains serving the most total Flash files actually serve relatively few *unique* Flash files, suggesting that ad sites reuse identical Flash files, but pass different parameters in order to render different ad content. For example, `adap.tv` generates 21% of all Flash traffic on the network, but does so with only 13 unique files. As depicted in Figure 3, only 19,000 unique Flash files are served during the test period. Further, only 6,000 of those unique Flash files contain distinctive function-level opcodes, as captured by our piecewise hashing.

Over the course of the experiment, the network starts to reach a steady state where fewer and fewer new Flash instances are observed. In 98% of the minutes analyzed, we see four or fewer new files, while in 57% of the minutes we see no new files at all.

On Packet Drops, CPU and Memory Usage: We used the hashes gathered during the July experiment, and augmented them with 745 file and 722 piecewise hashes of popular ads for our 5-day test in November — in total, the Flash filter contains 38,904 file and 11,091 piecewise hashes. During the test we observed an average of 23,000 unique IP addresses per day with up to 1,000 concurrent users. Throughput averaged 14,128 TCP flows per minute with peak periods of 35,000 flows. Our implementation reassembled TCP streams, parsed HTTP flows, and cached all web objects (step ❶) without dropping a single packet, but did observe 4.25% TCP reassembly errors.

Figures 4a and 4b show the average CPU and memory usage per minute for the network semantic caching and triggering module. The module works by using a single packet collection and reassembly thread, which launches a thread to parse and decompress each new TCP session. Parsed web objects are then passed to a different thread for caching. As shown in Figure 4a, the collector averages a modest CPU utilization of about 1.7 (170%), but can peak to 14 (1400%) for small time periods. CPU utilization refers to the percentage of CPU cycles used by the process; therefore, the collector uses on average the equivalent of 1.7 processor cores. Given the threading model, we recommend a system with many cores to best support the semantic caching and triggering module. Memory usage (Figure 4b) averages 22 GBs but can reach peaks of 40 GBs, suggesting that our caching model significantly reduces memory requirements over time.



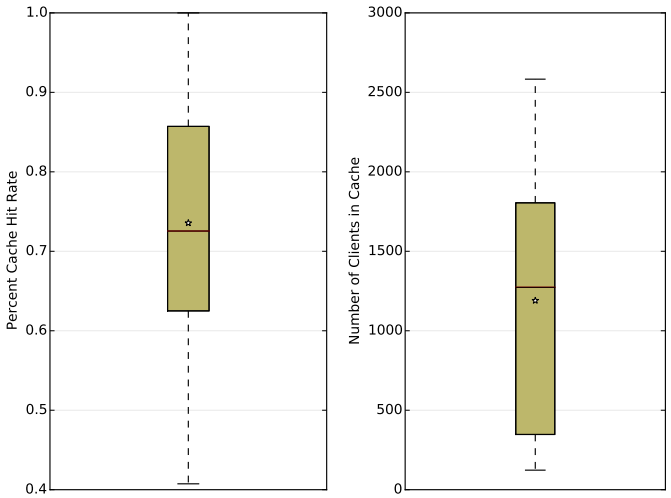
(a) Average CPU usage per minute.

(b) Average memory usage per minute.

Fig. 4: CPU and memory statistics for the semantic cache and trigger module.

On Cache Hit Rates and Chaining Algorithm Performance: Over the five day period, the collector reassembled 576,871 Flash objects of which 5,488 objects were analyzed using the chaining algorithm after filtering. Figure 5a shows the average cache hits per minute of the headless browser for all web requests to the two-level semantic web cache. The chaining algorithm had an average cache hit rate of 73% per minute with the majority of cache misses due to three main reasons. First, the TCP reassembly errors cause 4.25% of the web objects to be improperly processed and cached. Second, Flash can be loaded by other Flash files over intervals longer than the window set by the client cache meaning the corresponding webpages are no longer present in the cache (more on this in the following paragraph). Finally, a user may periodically visit a popular website that contains highly cacheable web objects such as images, JavaScript files. These files are cached by the user's web browser and thus might not be requested along with the Flash file. We mitigate these cache misses by retrieving missing JavaScript files from their source as discussed previously. On average there are 1,253 clients in the client cache with peak rates of 2,670 (see Figure 5b), while 90% of clients have less than 1,000 web objects in their cache at eviction. As a result, we found that setting the client LRU cache to size N=5,000 per client maintains a reasonable memory footprint.

Table IV shows that the chaining algorithm triggered a full sandbox analysis for 76% of all Flash files. Although this might seem low, the remaining Flash came from three distinct categories. First were those Flash files that require user interaction to load. For example, many Flash-based news sites will load an image for a news report video, and will not load the actual Flash video until the user clicks on the image. Another example is those pages that require user login credentials. Since we opt not to make use of any user credentials, Flash



(a) Average cache hit rate per minute.

(b) Average number of clients in the client cache.

Fig. 5: Two-level cache statistics.

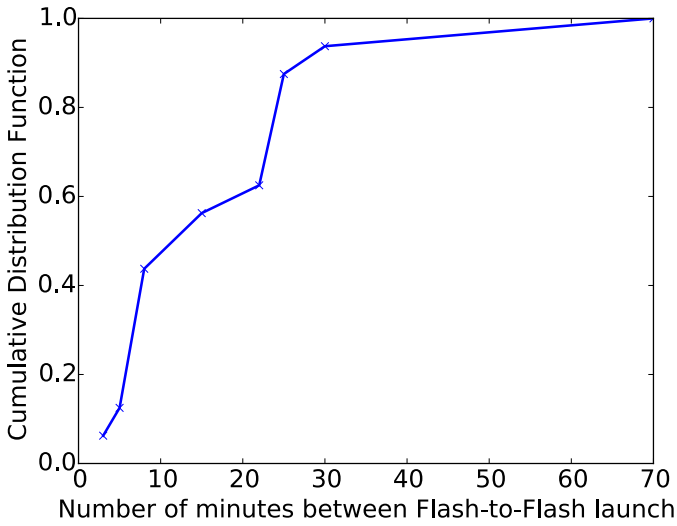


Fig. 6: Estimate of the amount of time elapsed between Flash to Flash file launches for those files not subjected to a full honeyclient analysis.

objects requiring credentials cannot be analyzed.

Triggered Full Sandbox Analysis	76%
Interactive	8 %
Flash in Flash	11 %
Errors	5 %

TABLE IV: Chaining algorithm match rate.

The second category is what we call "Flash within Flash" that occur over a time window larger than what is set by the client cache. For example, it is not uncommon that when a user watches a TV show using a Flash player, the player will load ads at various times throughout the show. As a result, the context web objects that loaded the Flash will no longer exist in the cache. In other cases, a page of ads may have been left undisturbed (*e.g.*, in another tab) for hours at a time while the ads cycle through various Flash files. Figure 6 shows an estimate for the amount of time elapsed between "Flash within Flash" file references for those Flash files that did not trigger a full sandbox analysis. Indeed, 90% of these flash files were loaded at least 8 minutes after their root Flash file. While we could increase the time windows to help identify the corresponding roots, we note that, as shown in the public dataset, attackers want to load exploits as quickly as possible, in order to increase the likelihood that the user will not navigate away from the site before infection. Our decision to not increase the window size is also tied to memory consumption. Admittedly, our approach is susceptible to low-and-slow attacks, but that limitation is not unique to this work.

Finally, 5% of the Flash files do not trigger a sandbox analysis due mainly to TCP reassembly errors that cause root webpages to be disregarded by the reassembler rather than cached. Furthermore, note that trigger files that are not properly reassembled are also disregarded from analysis meaning that we could miss a potentially malicious file. However, the errors in our proof-of-concept prototype originate from *libnids*, and we believe that these issues can be mitigated by using a more robust reassembler.

In 0.05% of the errors, an important JavaScript file is encrypted with SSL, which we currently do not support. Fortunately, many enterprises have the ability to inspect encrypted traffic at the border by using proxy servers specifically designed to decrypt and monitor encrypted traffic.

On Detecting Malicious Flash: As part of our online evaluation, we hosted a malicious landing page on an external network[6]. The exploit server automatically detects the victim's software configuration before serving one or more appropriate Flash exploits. In total, 11 unique Flash exploits are hosted (see Table V). Our "victim" system runs IE10 and Firefox on a Windows 7 VM within our campus network. We instrumented the victim to repeatedly visit the landing page with different versions of Flash, triggering each of the different exploits.

Since no packets are dropped in step ❶, it is not surprising that our framework detected *all* of these exploit instances in face of all the noise produced by the benign traffic. At the same time, no false positives were generated by our framework over the course of this 5 day period.

Aside from the injected metasploit malware, our approach flagged 6 malicious events, *i.e.,* 1 to 2 per day. To the best of our knowledge, these events were missed by our campus' Information Technology Service Office (ITS), which makes use of several commercial products to detect and block known malicious content on the network. The first event barred striking resemblance to the Magnitude samples examined in this paper. Two other instances were similar to Angler in that they checked for the installation of anti-viral and monitoring applications such as Norton and Fiddler. The final three instances were all heapspray incidents, with one emanating from an online TV site, while the others were site banners. Since the majority of

[6]Specifically, we used Metasploit's *browser_pwn2* module on an Amazon EC2 instance.

Metasploit Exploit	CVE Numbers	Flash Version Used
adobe_Flash_pixel_bender_bof	CVE-2014-0515	11.5.502.136
adobe_Flash_avm2	CVE-2014-0497	11.5.502.136
adobe_Flash_regex_value	CVE-2013-0634	11.5.502.136
adobe_Flash_uncompress_zlib_uaf	CVE-2015-0311	16.0.0.235
adobe_Flash_net_connection_confusion	CVE-2015-0336	16.0.0.235
adobe_Flash_worker_byte_array_uaf	CVE-2015-0313	16.0.0.235
adobe_Flash_pcre	CVE-2015-0318	16.0.0.235
adobe_Flash_nellymoser_bof	CVE-2015-3043, CVE-2015-3113	17.0.0.134
adobe_Flash_shader_job_overflow	CVE-2015-3090	17.0.0.134
adobe_Flash_shader_drawing_fill	CVE-2015-03105	17.0.0.134
adobe_Flash_domain_memory_uaf	CVE-2015-0359	17.0.0.134

TABLE V: List of exploits injected into the campus network and detected by the framework.

redundant Flash ads are filtered, the main sources of benign flash included online games, tutorials, news websites, online TV, online textbooks, website tracking, and adult content.

V. Case Study

In what follows, we perform a more in-depth analysis of the inner workings of the exploit kits in our empirical evaluation. Although we originally surmised that the landing pages would likely look like advertisements, we quickly noticed that the majority of pages were either composed of randomized English words or encoded character sets (or both). Indeed, these pages are never meant to be seen by the user, but rather hidden in a small `iframe`. Furthermore, buried in these pages are nuggets of data that the kit uses to help ensure it is not being run in isolation. For example, embedded JavaScript might only fully execute if the color of the third paragraph on the landing page is "red".

JavaScript is often the language of choice for would be attackers as it can be used to check browser configurations, and administer exploits either through browser or plugin vulnerabilities. The language is also ideal for obfuscation because objects and their functions are represented as hash tables making obfuscated code almost impossible to decipher without a debugger.

As mentioned above, almost all exploit kits conduct a reconnaissance phase to collect information about the browser and to determine whether it is operating in a legitimate environment. Browser configurations are determined using either the `navigator.plugins` API (Chrome, Firefox, and IE (11+)), or the proprietary `ActiveXObject` in older versions of IE. A kit will use browser vulnerabilities to determine whether it is operating in a virtualized environment, and will drop one or more exploit payloads onto the client system if the coast is clear. Below we describe some of the key characteristics of popular exploit kit families.

a) Fiesta: The Fiesta landing page is known for checking for a number of vulnerabilities in the browser and serving multiple exploits at once. The kit communicates with its server by encoding browser plugin information directly into the URL that is sent to exploit server similar to a command-and-control channel for a botnet. Fiesta's attack of choice is to abuse weaponized PDF documents to drop one or more malicious binaries onto the system. Indeed, we found one instance of the kit that dropped 12 binaries onto the system, while other instances launched `ping` or a command shell.

b) SweetOrange: SweetOrange likes to use JavaScript heapspray attacks, particularly by exploiting the rarely used VML API in Internet Explorer[7] to infect its victims. In three cases, the exploit kit launched the Windows Control Panel (`control.exe`) presumably to turn off key services.

c) Angler and Nuclear: Angler and Nuclear appear to be popular vectors for dropping so-called *Ransomware*. Recent versions (circa June 2015) of the kits are known to check for Kaspersky and Norton browser plugins and to use vulnerabilities in the IE browser to detect virtualization. For example, Figure 7 shows a snippet of JavaScript code from an instance of the Angler exploit kit (June 2015). The code uses the HTML `script` with an invalid language to check for commonly installed files related to VMWare, VirtualBox, Parallels, Kaspersky, and Fiddler. If any of the aforementioned applications exist, Angler will not exploit the system. Instances of Angler from April of 2015 do similar checking using JavaScript's `Image` object as a medium to gain disk access.

These exploit kits also like to embed JavaScript directly into the HTML of the landing page. Indeed, entire JavaScript libraries (like the script in Figure 7) are embedded inside HTML tags such as `p` as demonstrated by an Angler instance in Figure 8. The JavaScript is decoded by a number of obfuscated method calls, and the resulting code is executed using an `eval` function call. As a result, current generation exploits must be analyzed within the larger context of the website.

VI. Limitations

Many of the evasion techniques used against our system are inherent to honeyclients in general and are being actively researched in the security community. For example, as shown in our use case, exploits will often check for evidence that the environment is a virtual machine. In the short term, we can help combat this check by installing VM libraries in non-standard locations or by attempting to detect and flag potentially evasive behavior. In the long term, however, a better solution would be to adopt ideas from Kirat et al. [12, 13] to build sandboxes on "bare-metal" that are able to revert system changes without relying on hardware virtualization.

An obvious attack against sandbox-based approaches is for the attacker to inject delays into the exploit kit code in the hopes that the sandbox execution will timeout before the exploit is executed. Such timeouts can be risky for the attacker because the user of the targeted machine could surf to a new

[7]Described in the whitepaper at http://www.vupen.com/blog/20130522.Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php

```
function xTrue(rp1, rr) {
    var rs1 = 're',
        ac = ['QUPFE', 'PTKytUJ', setQuery,
                setDatabase];
    if (window[ac[rr]]) return;
    var el = document.createElement('script');
    if (!window['MSInputMethodContext'])
            el['language'] = 'some';
    el.onload = function() {
        ac[rr + 2]();
    };
    el.src = rs1 + 's://' + rp1 + '/#16/#1';
    el['onreadystatechange'] = function() {
        var sr = 'rea' + 'dyState',
            r = this[sr];
        if (r == 'complete' || r == 'loaded') {
            ac[rr + 2]();
        }
    };
    document.body.appendChild(el);
}
....

var path_sys32 = "\\Windows\\System32\\drivers\\",
  vm_s = ["vm3dmp", "vmusbmouse", "vmmouse",
                "vmhgfs", "VBoxGuest",
                "VBoxMouse", "VBoxSF", "VBoxVideo",
                "prl_time"];

 for (var i = 0; i < vm_s.length; i++) {
        xTrue(path_sys32 + vm_s[i] + '.sys', 0);
 }
```

Fig. 7: JavaScript snippet from Angler Exploit Kit that checks to see if files exist on the hard drive.

page before the delay has transpired. One way to combat such delays is by instrumenting our headless browser to record sleep times and ensuring that the sandbox runs for at least that time period. Sandboxes in general can also attempt to patch out sleep functionality or adjust the time value it presents to the software, but either of these techniques can still be defeated if malware uses external sources of time information, such as the Internet, to verify that the embedded delays have completed as expected[8]. Thwarting such attacks remains an active area of research and this limitation is not specific to our approach [17].

Attackers can also force a user to interact with the system in some way before triggering an exploit. Such an attack would be difficult to detect in a framework such as ours, which is designed to work without manual intervention. Extensions to the framework could simulate user interaction, such as automated button clicks, but this is left as future work. Also, if an attacker is willing to require user interaction in order to carry out an attack, many other non-exploit attack vectors exist, such as simply tricking a user into downloading and running an executable file.

Alternatively, an exploit could also alter URLs using some randomized token based on local settings. One approach to thwarting such attacks is to perform URL similarity matching (as done extensively in the literature [33]) while instrumenting

[8]See *Sleeping Your Way Out Of The Sandbox*, SANS Institute Reading Room, accessed August 16 2015

```
<p id="aXzYFDTUynszpYYl">i3g5t(wwffos.dis 2gn7rgkfd{
    fsa 1 )1vf l
   gt '' w=f,ek'rl 2=i'eaKlfou;giintn  
    tocozydU u{( t(re
  n)Fmrn)   d IF][f2qZ y;}Wul{a teinxg t
  oTc()nr(IFdudreyFZmznUtny Wfint  3q ;o}) &
    nbsp;[uc]n{urte(t
   aratD)g eanI W[Fnd d(4FymyzZqU i tl;a ]mv=r
    } )ru  
    ....
</p>

<p id="aXzYFDTUmPsTzMivOvU"> P L]i'n I[=Pwdywe'
    o ngTr'c;'onetu il
  aft''hf d{([n]oi w)hQw oy[Ptwr=d Ii;un w'ePi.dw''
    ]ew's;   Lno=f l
  a5sg3j7sfstd2=fge2sggw.i;3w7sonf 5tds oidsl f;
    awsf=edwn .d
  1=3s2sfgf7t5sff gasg35ed;1.ntoi wsf2wgl}e s=fsa &
    nbsp; 2df7s;f g)t(n
  cfrn TotaueNi{57tsisn 2.sfwd3wggof cfn e=d}ut;r &
    nbsp;n utii)ls
Ipome...
</p>

 <p id="aXzYFDTUlIIsxbIijznJVBuz">
    wcaEqdjoiPP8WZwzng8. )e{rfou=(itntn
     cuyqIW yzrFU[(mFnndZd ol.O]w)1d 9i;
    n wY}40A00M
  Yc'F0=
    aSY66r90B0000F0C009100682DD0A0000306701000C9FFB
    ...'
</p>

<p id=" aXzYFDTURSYMdwuFLH "> ''hffd( [n]oi!
    wihQw)a(gtfou{Ti)ntn
   rec Q=' w[i{hw 'onh f]dt'eyPed;r[nLoi wuPIw
    'o3.f=w ]d 2i;'n
  ws'5f;legjsg d  f7=tasswtfssd3oigfd2sw5ng7.
    snso. ;f ief
  saw=dwl3 l agft21ssd7gf5=fse2sggw.i;3w7sonf 5tds
    ;f} fa2feful
   sd  =ntlmliuocIaispnet(i s w e{d )e &
    nbsp;arvt
  =naDueDtea(acv r ) tra;=ctrauol  de  l
    {nuD; ;w}
  nte=)aheDw(    i-edt a(l D trcee au&lt
    ; tuci;l  )i
  sl}mfnio   fUtfn)((xSet {
  i3wtfssd3oigfd2sw5ng7.nvlrvr e) n
  0rt;...
</p>
<script>
    KBUmJFFEgyrFy([pmlQVe][Cd](RTcV) + "RSYMdwuFLH");
</script>
```

Fig. 8: Example of a portion of a landing page for the Angler Exploit Kit. Note that paragraph tags with ids are used to house obfuscated and encoded JavaScript code. The code is then loaded and decoded by obfuscated code in <script> tags. In this small example, the KBUmJFFEgyrFy function is loading code from paragraph id aXzYFDTURSYMdwuFLH

the headless browser to pass file types to the web cache in order to improve the matching process.

An attacker could try to overwhelm the framework by loading several Flash files at once with only one of the files being malicious. Our chaining algorithm tries to mitigate this attack by analyzing URLs that lead to multiple exploitable files only once. This is by no means foolproof, but large spikes in Flash files could also be recorded and presented to the security analyst for further analysis.

Finally, as discussed in §IV, the framework does not directly support Flash loaded within other Flash files because the time window between file loads can be larger than the time window over which HTTP traffic is cached. In such a scenario, the attacker is relying on the user staying on a web page for a protracted period of time in a low-and-slow style attack.

VII. Conclusion

In this paper, we present a network-centric approach to accurately and scalably detect malicious exploit kit traffic by bringing a honeyclient to-the-wire. By caching, filtering and replaying traffic associated with exploitable files, our approach allows us to use our knowledge of the clients in the network to dynamically run exploits in a safe and controlled environment. We evaluated our framework on network traces associated with 177 real-world exploit kits and demonstrated that we could detect zero-day exploits as they occur on the wire, weeks before conventional approaches. We supplement these analysis with case studies discussing interesting aspects of the detected behaviors in the studied exploit kits. Lastly, a preliminary analysis in an operational deployment on a large university campus network shows that our techniques can handle massive HTTP traffic volumes with modest hardware.

Acknowledgments

We express our gratitude to Jim Gogan and Alex Everett of the Information Technology Service Office and the Computer Science networking staff (especially, Murray Anderegg and Bil Hayes) for their efforts in deploying the infrastructure used in this study. The researchers and the Technology Service Office have a longstanding memorandum of understanding in place to analyze anonymized network traffic on campus. The memorandum covers specific uses and types of networking data, as well as conditions for securing and accessing such data. We also thank Jan Werner, the anonymous reviewers, and our shepherd Thorsten Holz for their insightful comments. This work is supported in part by the National Science Foundation under awards 1421703 and 1127361 (with a supplement from the Department of Homeland Security under its Transition to Practice program). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Department of Homeland Security.

References

[1] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, "FPDetective: Dusting the web for fingerprinters," in *ACM Conference on Computer and Communications Security*, 2013.

[2] M. Antonakakis, R. Perdisci, D. Dagon, W. Lee, and N. Feamster, "Building a Dynamic Reputation System for DNS," in *USENIX Security Symposium*, 2010.

[3] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *Symposium on Recent Advances in Intrusion Detection*, 2007.

[4] A. Blum, B. Wardman, T. Solorio, and G. Warner, "Lexical feature based phishing url detection using online learning," in *ACM Workshop on Artificial Intelligence and Security*, 2010.

[5] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: a fast filter for the large-scale detection of malicious web pages," in *World Wide Web Conference*, 2011.

[6] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *World Wide Web Conference*, 2010.

[7] G. De Maio, A. Kapravelos, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "PExy: The Other Side of Exploit Kits," in *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2014.

[8] B. Eshete and V. N. Venkatakrishnan, "Webwinnow: Leveraging exploit kit workflows to detect malicious urls," in *ACM Conference on Data and Application Security and Privacy*, 2014.

[9] J. Gassen and J. Chapman, "Honeyagent: Detecting malicious java applets by using dynamic analysis," in *Malicious and Unwanted Software: The Americas (MALWARE), International Conference on*, 2014.

[10] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker, "Manufacturing compromise: the emergence of exploit-as-a-service," in *ACM Conference on Computer and Communications Security*, 2012.

[11] L. Invernizzi, S. Benvenuti, P. M. Comparetti, M. Cova, C. Kruegel, and G. Vigna, "Evilseed: A guided approach to finding malicious web pages," in *IEEE Symposium on Security and Privacy*, 2012.

[12] D. Kirat, G. Vigna, and C. Kruegel, "Barebox: Efficient malware analysis on bare-metal," in *Annual Computer Security Applications Conference*, 2011.

[13] ——, "Barecloud: Bare-metal analysis-based evasive malware detection," in *USENIX Security Symposium*, 2014.

[14] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, Supplement, 2006.

[15] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang, "Knowing your enemy: understanding and detecting malicious web advertising," in *ACM Conference on Computer and Communications Security*, 2012.

[16] Z. Li, S. Alrwais, Y. Xie, F. Yu, and X. Wang, "Finding the linchpins of the dark web: a study on topologically dedicated hosts on malicious web infrastructures," in *IEEE Symposium on Security and Privacy*, 2013.

[17] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Symposium on Recent Advances in Intrusion Detection*, 2011.

[18] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond blacklists: learning to detect malicious web sites from

suspicious urls," in *Conference on Knowledge Discovery and Data Mining*, 2009.

[19] ——, "Learning to detect malicious urls," *ACM Transactions on Intelligent Systems Technology*, vol. 2, no. 3, May 2011.

[20] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, "Enriching network security analysis with time travel," in *ACM Conference of the Special Interest Group on Data Communications*, 2008.

[21] H. Mekky, R. Torres, Z.-L. Zhang, S. Saha, and A. Nucci, "Detecting malicious http redirections using trees of user browsing activity," in *IEEE Conference on Computer Communications*, 2014.

[22] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, "Webwitness: Investigating, categorizing, and mitigating malware download paths," in *USENIX Security Symposium*, 2015.

[23] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, 1999.

[24] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Comprehensive shellcode detection using runtime heuristics," in *Annual Computer Security Applications Conference*, 2010.

[25] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu, "The ghost in the browser analysis of web-based malware," in *USENIX Workshop on Hot Topics in Understanding Botnet*, 2007.

[26] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, "All your iframes point to us," in *USENIX Security Symposium*, 2008.

[27] K. Rieck, T. Krueger, and A. Dewald, "Cujo: efficient detection and prevention of drive-by-download attacks," in *Annual Computer Security Applications Conference*, 2010.

[28] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *USENIX Conference on System Administration*, 1999.

[29] J. Schlumberger, C. Kruegel, and G. Vigna, "Jarhead analysis and detection of malicious java applets," in *Annual Computer Security Applications Conference*, 2012.

[30] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos, "Shellos: enabling fast detection and forensic analysis of code injection attacks," in *USENIX Security Symposium*, 2011.

[31] B. Stancill, K. Z. Snow, N. Otterness, F. Monrose, L. Davi, and A.-R. Sadeghi, "Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets." Symposium on Recent Advances in Intrusion Detection, 2013.

[32] B. Stock, B. Livshits, and B. Zorn, "Kizzle: A signature compiler for exploit kits," Microsoft Research, Tech. Rep. MSR-TR-2015-12, February 2015.

[33] G. Stringhini, C. Kruegel, and G. Vigna, "Shady paths: leveraging surfing crowds to detect malicious web pages," in *ACM Conference on Computer and Communications Security*, 2013.

[34] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. A. Rajab, "Ad injection at scale: Assessing deceptive advertisement modifications," in *IEEE Symposium on Security and Privacy*, 2015.

[35] T. Van Overveldt, C. Kruegel, and G. Vigna, "Flashdetect: Actionscript 3 malware detection," in *Symposium on Recent Advances in Intrusion Detection*, 2012.

[36] D. Y. Wang, S. Savage, and G. M. Voelker, "Cloak and dagger: Dynamics of web search cloaking," in *ACM Conference on Computer and Communications Security*, 2011.

[37] G. Wang, J. W. Stokes, C. Herley, and D. Felstead, "Detecting malicious landing pages in malware distribution networks," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2013.