Detecting Malicious Exploit Kits using Tree-based Similarity Searches

Teryl Taylor†, Xin Hu‡, Ting Wang∗, Jiyong Jang‡, Marc Ph. Stoecklin‡, Fabian Monrose†, and Reiner Sailer‡
† University of North Carolina at Chapel Hill, {tptaylor, fabian}@cs.unc.edu
‡ IBM T.J. Watson Research Center, {huxin, jjang, mpstoeck}@us.ibm.com
∗ Lehigh University, ting@cse.lehigh.edu

ABSTRACT

Unfortunately, the computers we use for everyday activities can be infiltrated while simply browsing innocuous sites that, unbeknownst to the website owner, may be laden with malicious advertisements. So-called malvertising, redirects browsers to web-based exploit kits that are designed to find vulnerabilities in the browser and subsequently download malicious payloads. We propose a new approach for detecting such malfeasance by leveraging the inherent structural patterns in HTTP traffic to classify exploit kit instances. Our key insight is that an exploit kit leads the browser to download payloads using multiple requests from malicious servers. We capture these interactions in a "tree-like" form, and using a scalable index of malware samples, model the detection process as a subtree similarity search problem. The approach is evaluated on 3800 hours of real-world traffic including over 4 billion flows and reduces false positive rates by four orders of magnitude over current state-of-the-art techniques with comparable true positive rates. We show that our approach can operate in near real-time, and is able to handle peak traffic levels on a large enterprise network — identifying 28 new exploit kit instances during our analysis period.

1. INTRODUCTION

Today, our computers are routinely compromised while performing seemingly innocuous activities like reading articles on trusted websites [43] (e.g., the NY Times). All too often, these compromises are perpetrated via complex interactions involving the advertising networks that monetize these sites. Since crime typically follows the money, it is not too surprising then that miscreants have turned their attention to exploiting advertising networks as a way to reach wider audiences. In 2012 alone, web-based advertising generated revenues of over $36 billion [28], and its wide-spread reach makes it an excellent target for fraudsters and deviants. Furthermore, the many players in the online advertising industry — publishers (who display ads), advertising networks (who deliver ads), and advertisers (who create content) — offer a multitude of vantage points for attackers to leverage, and many of these compromises can go unnoticed for extended periods. A well known example is the widely publicized case involving advertising networks from Google and Microsoft that were tricked into displaying malicious content by miscreants posing as legitimate advertisers [16]. Sadly, such abuses are not isolated incidents and so-called *malvertising* has plagued many popular websites [29], exploited mobile devices [33], and have even been utilized as vessels for botnet activity [3]. For the most part, these exploits are delivered over HTTP, and detecting and defending against such attacks require accurate and efficient analytical techniques to help network operators better understand the attacks being perpetrated on their networks.

Many of these HTTP-based attacks are launched through the use of exploit kits [9, 6], which are web-based services designed to exploit vulnerabilities in web browsers by downloading malicious Java, Silverlight, or Flash files. Exploit kits, such as Fiesta and Blackhole represent an entire software-as-a-service subindustry. The exploitation of a user's system typically follows a four-step process wherein a user navigates to a website (e.g., CNN) that — unbeknownst to the user — contains an external link (e.g., an advertising link) with an injected `iframe` that in turn directs the user's browser to an invisible exploit kit landing page. At that point, information about the victim's system is passed along to the attacker's server, which is then used to select a malicious exploit file that is automatically downloaded. The downloaded file exploits a vulnerability on the system that allows the attacker to install a malicious binary or otherwise control the victim's machine.

Security analysts typically defend enterprise networks from these attacks using network monitoring devices (such as intrusion detection systems) that search HTTP traffic as it passes through the network's edge for signature matches, statistical patterns or known malicious domain names. Unfortunately, the attack landscape constantly changes as the attackers attempt to hide their nefarious web-based services and avoid blacklisting. As a result, current approaches typically incur high false positive and negative rates.

In this paper, we explore a network-centric technique for identifying agile web-based attacks with a focus on reducing false positives over existing approaches while maintaining or improving false negatives. We improve detection rates by leveraging the structural patterns inherent in HTTP traffic to classify specific exploit kit instances. Our key insight is that to infect a client browser, a web-based exploit kit must lead the client browser to visit its landing page (possibly through redirection across multiple compromised/malicious servers), download an exploit file and download a malicious payload, necessitating multiple requests to several malicious servers. Our approach captures the structure of these web requests in a tree-like form, and uses the encoded information for classification purposes. To see how this can help, consider the example where a user visits a website, and that action in turn sets off a chain of web requests that loads various web resources, including the main page, images, and advertisements. The overall

CODASPY'16, March 09-11, 2016, New Orleans, LA, USA
© 2016 ACM. ISBN 978-1-4503-3935-3/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2857705.2857718

structure of these web requests forms a tree, where the nodes of the tree represent the web resources, and the edges between two nodes represent the causal relationships between these resources. For instance, loading an HTML page which contains a set of images might require one request for the page (the root node) and a separate set of requests (the children) for the images. When a resource on a website loads an exploit kit, the web requests associated with that kit form a subtree of the main tree representing the entire page load. Also, the exploitation is a multi-stage process involving multiple correlated sessions. By providing context through structure, we can capture the correlation among sessions, thereby providing improved detection accuracy.

Intuitively, identifying the malicious subtree within a sea of network traffic can be modeled as a subtree similarity problem. We demonstrate that we can quickly identify the presence of similar subtrees given only a handful of examples generated by an exploit kit. In order to do so, we build an index of malicious tree samples using information retrieval techniques. The malware index is essentially a search engine seeded with a small set of known malicious trees. A device monitoring network traffic can then query the index with subtrees built from the observed client traffic. The traffic is flagged as suspicious if a similar subtree can be found in the index. We note that our decision to use techniques from the field of information retrieval is motivated by the fact that these techniques are known to work well with extremely sparse feature sets (e.g., words and phrases), and the feature space for network analysis can be equally as sparse. Moreover, in information retrieval, the desire is to access a set of documents based on a user's query, and in most cases, the resulting set typically comprises a very small portion of the overall set of documents in the data store. Similarly, in network security, the malicious instances in the dataset tend to comprise only a fraction of the overall network traffic.

In the remainder of this paper, we present several contributions including a network-centric approach based on subtree similarity searching for detecting HTTP traffic related to malicious exploit kits on enterprise networks. We show that using the structural patterns of HTTP traffic can significantly reduce false positives with comparable false negative rates to current approaches. We also provide a novel solution to the subtree similarity problem, by modelling each node in the subtree as a point in a potentially high dimensional feature space. Finally, we utilize this technique to identify agile exploit kits found in a large network deployment.

2. RELATED WORK

Over the past decade, the web has become a dominant communication channel, and its popularity has fueled the rise of malicious websites [39] and malvertising as a vector for infecting vulnerable hosts. Provos et al. [26] examined the ways in which web page components could be used to exploit web browsers and infect clients through drive-by downloads. That study was later extended [27] to include an understanding of large-scale infrastructures of malware delivery networks, and provided overall statistics on the impact of these networks at a macro level. Their analysis found that ad syndication significantly contributed to drive-by downloads. Similarly, Zarras et al. [43] performed a large scale study of the prevalence of malvertising in ad networks. They showed that certain ad networks are more prone to serving malware than others. Grier et al. [9] 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 through drive-by downloads from a handful of exploit kit flavors.

Since then, detecting malicious landing pages has been a hot topic of research. The most popular approach involves crawling the web for malicious content using known malicious websites as a seed [15, 17, 18, 8]. The crawled websites are verified using statistical analysis techniques [17] or by deploying honeyclients in virtual machines to monitor OS and browser changes [27]. Other approaches include the use of a PageRank algorithm to rank the maliciousness of crawled sites [18] and the use of mutual information to detect similarities among content-based features derived from malicious websites [38]. Eshete and Venkatakrishnan [8] identified content and structural features using samples of 38 exploit kits to build a set of classifiers that can analyze URLs by visiting them through a honey client. These approaches are complimentary to ours, but require significant resources to comb the Internet at scale.

Other approaches involve analyzing the source code of exploit kits to understand their behavior. For example, De Maio et al. [6] 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. [34] clustered exploit kit samples to build host-based signatures for anti-virus engines and web browsers. Closer to our work are approaches that try to detect malicious websites using HTTP traffic. Cova et al. [5], for example, designed a system to instrument JavaScript run-time environments to detect malicious code execution while Rieck et al. [31] described an online approach that extracts all code snippets from web pages and loads them into a JavaScript sandbox for inspection. Unfortunately, these techniques do not scale well, and require precise client environment conditions to be most effective.

Other approaches focus on using statistical machine learning techniques to detect malicious pages by training a classifier with malicious samples and analyzing traffic in a network environment [31, 2, 1, 20, 21, 22, 24]. More comprehensive techniques focus on extracting javascript elements that are heavily obfuscated or `iframes` that link to known malicious sites [26, 5]. Cova et al. [5] and Mekky et al. [22] note that malicious websites often require a number of redirections, and build a set of features around that fact. Canali et al. [2] describes a static prefilter based on HTML, javascript, URL and host features while Ma et al. [20, 21] 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 [27, 26, 2]. Unfortunately, these techniques take broad strokes in terms of specifying suspicious activity, and as such, tend to have high false positive rates. They also require large training sets that are often not available. By contrast, we provide a framework for detecting various flavors of exploit kits, and utilize the interactions between HTTP flows to reduce false positives from a small seed of examples.

Yegneswaran et al. [42] describe a framework for building semantic signatures for client-side vulnerabilities packet traces collected from a honeypot. The work shares the similar observation with ours that correlating flows can help to reduce false positives; however, our work focuses on the specific problem of detecting server-side exploit kits using the structure of HTTP traffic. As such, our approach is different in that we model kits as trees, and take advantage of structural similarity properties to reduce FPs. We also use thresholding to control false positive and negative rates. More recently, Stringhini et al. [35] proposed a learning approach to detect malicious redirection chains using a proprietary dataset. The technique requires traffic from a large crowd of diverse users from different countries, using different browsers and OSes to visit the same malicious websites in order to train a classifier. Unfortunately, as shown in the work, the approach leads to relatively high false positives and negatives with modest data labels and can only detect chains whereby the last node is deemed malicious. By contrast, our work does not model client usage patterns and is not lim-

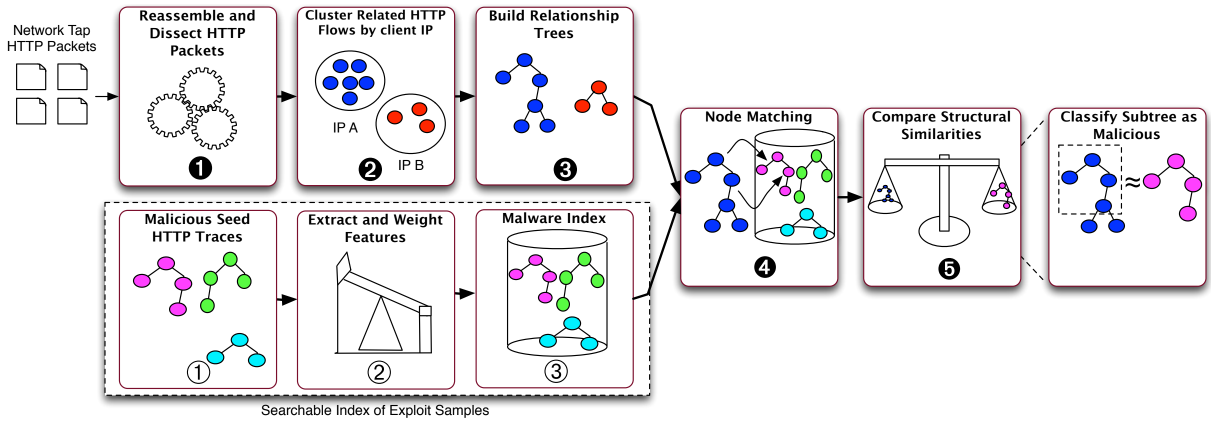
Network Tap HTTP Packets
Reassemble and Dissect HTTP Packets
❶
Cluster Related HTTP Flows by client IP
IP A
IP B
❷
Build Relationship Trees
❸
Node Matching
❹
Compare Structural Similarities
❺
Classify Subtree as Malicious
Malicious Seed HTTP Traces
①
Extract and Weight Features
②
Malware Index
③
Searchable Index of Exploit Samples

Figure 1: High level overview of the search-based malware system.

ited to the presence of redirection chains to identify exploit kits. Our technique is based on structural similarity; therefore, the last node in the structure does not need to be malicious. Finally, our approach is designed to specifically reduce false positives and negatives in light of a small amount of malicious training data.

Subtree Similarity Search Problem: Lastly, we note that the subtree similarity-search problem on large datasets remains an open research problem. Most proposals require scanning each tree in the dataset and then applying tree edit distance techniques to prune the search space. Recently, Cohen [4] combined the general structural commonalities of trees as well as the uncommon elements to reduce the number of trees checked in a similarity search. The drawback of that work is that the indices are 10x the size of the input data and only works with single-labeled nodes. Our work is based on similar ideas to Cohen [4] but works on trees where the nodes themselves have a large number of features. To make the approach practical, we leverage the sparsity of the feature space in network traffic.

3. APPROACH

For the most part, today's network-centric approaches for detecting HTTP-based malware use HTTP flows individually when performing analytics, but doing so can lead to high false positive rates. By contrast, we focus on the interactions *between* flows to identify malicious cases in network traffic in order to reduce false positives and identify exploit kits — hopefully before they have an opportunity to exploit a vulnerable client. Our key insight is that to infect a client, a web-based exploit kit will lead the client browser to download a malicious payload by making multiple web requests to one or more malicious servers. We use those multiple requests to build a tree-like structure and model the problem as a subtree similarity search problem.

A high-level overview of our approach is shown in Figure 1. There are two main components: an index of known exploit kits (Figure 1 (bottom)) and an online component that monitors HTTP traffic and performs comparisons with the index to identify and label potentially malicious traffic (Figure 1 (top)).

Indexing stage: In step ①, we collect HTTP traffic samples representing client browser interactions with various flavors of exploit kits (e.g., Fiesta) and convert them into tree-like representations. Flow-level and structure information are extracted from these trees (step ②) and then stored in a tree-based invertible index (step ③) called a malware index as described in more detail in Section 3.2.

Classification stage: HTTP traffic is monitored at the edge of an enterprise network, and packets are dissected and reassembled into bidirectional flows (see step ❶). The reassembled flows are grouped by client IP addresses (step ❷) and assembled into tree-like structures (step ❸, § 3.1) called web session trees. The nodes in the web session tree are then mapped to "similar" nodes of the trees in the malware index using content features (step ❹, § 3.3.1), and finally, the mapped nodes are structurally compared to the trees in the index to classify subtrees as malicious (step ❺, § 3.3.2). Given a web session tree and an index of malware trees, the goal is to find all malicious subtrees in the tree that are similar to a tree in the index.

3.1 On Building Trees

In both components of our system (indexing and classification), HTTP traffic is grouped and converted into tree-like structures called web session trees. We use a two-step process to build these session trees for analysis. The first step in the process is to assemble HTTP packets into bidirectional TCP flows and then group them based on their client IP addresses. Flows are ordered by time, and then associated by web session using a technique similar to that used by Ihm and Pai [14] and Provos et al. [27]. A web session tree is defined as all HTTP web requests originating from a single root request over a rolling time window of a tuneable parameter $\triangle t_w$ (empirically set to five seconds in our implementation). A node in the tree is an HTTP flow representing some web resource (e.g., webpage, picture, executable, and so on) with all related flow attributes including URL, IP, port, and HTTP header and payload information. An edge between nodes represents the causal relationship between the nodes (e.g., a webpage *loads* an image). For example, a client surfing to Facebook creates a single root request for the Facebook main page (i.e., the root node of the web session tree), which in turn *loads* images and JavaScript files (i.e., the child nodes). All related files form a client "web session" and the relationships between these resources form a tree-like structure as outlined below.

Each HTTP flow is compared with flow groups that have been active in the last $\triangle t_w$ window for the associated client IP address. Flows are assigned to a particular group based on specific header and content-based attributes that are checked in a priority order. The highest priority attributes are the HTTP `Referer` and the `Location` fields. The `Referer` field identifies the URL of the webpage that linked the resource requested. Valid `Referer` fields are used in approximately 80% of all HTTP requests [14] making them a useful attribute in grouping. The `Location` field is present during a `302` server redirect to indicate where the client browser should query next. After a time window expires, a web session tree is built from the associated flows. Note that our approach can analyze HTTPS traffic in cases where there is a man-in-the-middle proxy that can decrypt SSL sessions.

We chose this tree building technique because our dataset lacked the full packet payloads required to use more complex and exact

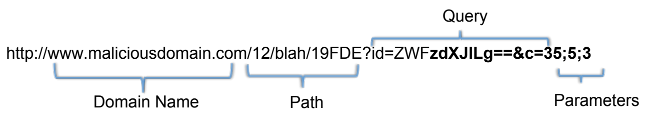
Query
http://www.maliciousdomain.com/12/blah/19FDE?id=ZWFzdXJILg==&c=35;5;3
Domain Name
Path
Parameters

Figure 2: The components of a URL for feature extraction.

approaches [23]. Even so, the tree building approach we used has been effectively applied in other studies [14, 27, 22] and aptly demonstrates the utility of our similarity algorithm. In Section 5.3, we discuss how our algorithm can be utilized to scalably build trees using more complex and time intensive techniques.

3.2 On Building the Malware Index

The malware index is built using HTTP traces from samples of well-known exploit kits (e.g., Fiesta). These samples are gathered by crawling malicious websites [15, 17, 18] using a honeyclient. A honeyclient is a computer with a browser designed to detect changes in the browser or operating system when visiting malicious sites. The first step in building the index is to compile a list of URLs of known malicious exploit kits from websites such as threatglass.com, and urlquery.net. Next, each page must be automatically accessed using the honeyclient and the corresponding HTTP traffic is recorded. Each trace is transformed into a tree using the process in Section 3.1, and then content-based (node-level) and structural features are extracted and indexed.

Content-based (Node-level) Indexing: An exploit kit tree is comprised of N nodes, where each node represents a bidirectional HTTP request/response flow with packet header, HTTP header, and payload information available for extraction and storage in a document style inverted index. Each bidirectional flow (or node in a tree) can be thought of as a document, and its features as the words of the document, which are indexed. Each node is given a unique integer ID and three types of features are extracted: token features, URL structural features, and content-based features.

Token features are mainly packet header and URL features. They are gathered from the URL by breaking it down into its constituent parts: domain names, top level domain, path, query strings, query key/value pairs, parameters, destination IP addresses, and destination subnets. All attributes are stored as bags of tokens. For example, the token features for the URL shown in Figure 2 would be: `www.maliciousdomain.com`, `com`, `12`, `blah`, `19FDE`, `id=ZWFzdXJILg==`, `c=35`, `5`, and `3`.

URL structural features abstract the components of the URL by categorizing them by their data types rather than their actual data values (as in the token features). We use 6 common data types in URLs: numeric, hexadecimal, base64 encoding, alphanumeric, and words. These datatype encodings are used in conjunction with the lengths or ranges of lengths of corresponding tokens to generate structural URL features. For example, the URL structural features for the URL shown in Figure 2 `12/blah/19FDE` would be broken into 3 features: path-num-2, path-word-4, path-hex-5.

Content-based features are extracted from the HTTP headers or payloads where possible. They include binned content lengths, content types, and redirect response codes.

Structural Indexing: Each malware tree in the index is assigned a unique tree identifier, while each node has a unique node identifier. The tree is stored as a string of node identifiers in a canonical form that encodes the tree's structure. The canonical string is built by visiting each node in the tree in a preorder traversal, and appending node identifiers to the end of the string. Note that each indexed node contains the identifier for its corresponding tree to allow for easy mapping from node to tree while each tree structure is labelled by exploit kit type (e.g., Flashpack, Fiesta).

3.3 On Subtree Similarity Searches

With a malware index at hand, we then monitor HTTP traffic at the edge of an enterprise network, and convert the traffic into web session trees. Our task is to determine whether any of the trees contain a subtree that is similar to a sample in the index. If so, the tree is flagged as malicious and labeled by its exploit flavor.

We approach the subtree similarity search problem using a two-step process: node level similarity search and structural similarity search. First, we determine whether any nodes in a web session tree T are "similar" to any nodes in the malware index. If there are multiple nodes in T that are similar to a tree E in the index, then we extract the subtree S containing those nodes, and compare S structurally with E using a tree edit distance technique. Subtrees with sufficient node overlap and structural similarity with E are flagged as malicious. Structural similarity is used because it significantly reduces false positives over grouping HTTP flow sequences (§ 5).

3.3.1 Node Level Similarity Search

To determine whether any nodes in T are sufficiently similar to nodes in the malware index, we extract the set of token, URL structure, and content-based features from each node x in T. These node features are then used to query the index and return any nodes i that have a feature in common with node x. Node similarity is measured by a score based on the overlapping features between nodes.

In this work, we compare two node similarity approaches: the Jaccard Index, and the weighted Jaccard Index to determine how weighting affects the accuracy of the algorithm. The Jaccard Index [10] is a similarity metric that measures the similarity of two sets $X = \{x_1, ..., x_n\}$ and $I = \{i_1, ..., i_n\}$ by calculating $J(X, I) = \frac{|X \cap I|}{|X \cup I|}$. This generates a score between 0 and 1, with higher scores meaning higher similarity. More precisely, we use a variant of the Jaccard Index, called relevance to determine how relevant the set of node features of x in T is to the set of node features of i in the index. To calculate the relevance of X to I, the Jaccard Index becomes: $J(X, I) = \frac{|X \cap I|}{|I|}$.

Two flows x and i are considered similar if $J(X, I) > \epsilon$, where X and I are feature sets of x and i respectively, and ϵ is a user defined threshold. If a node in tree T is similar to a node in the index, the node in T is assigned the ID from the node in the index. The node IDs are used to compare the structural similarities of the subtrees of T with the matching trees in the index (Section 3.3.2).

A weighted Jaccard Index [10] introduces weighting to the features of the set. A higher weight value on a feature emphasizes those features that are most distinctive to a malicious flow; thereby, increasing the similarity score of two nodes that are malicious. The weighted intersection of X and I is defined as

$$W(X, I) = \sum_{x \in X \cap I} w(x),$$

where w is the weight of each feature x.

Then, the weighted Jaccard Index becomes:

$$WJ(X, I) = \frac{|X \cap I|}{|X \cup I|} = \frac{W(X, I)}{C(X) + C(I) - W(X, I)},$$

where $C(X) = |X| = \sum_{x \in X} w(x)$. Again, we use a variant of the weighted Jaccard Index to calculate the relevance of X to I:

$$WJ(X, I) = \frac{|X \cap I|}{|I|} = \frac{W(X, I)}{C(I)},$$

We apply a probabilistic term weighting technique first introduced by Robertson and Jones [32] which gives an ideal weight to term t from query Q. The terms are used in a similarity-based scoring scheme to find a subset of the most relevant documents to query Q. Here, term t is a feature extracted from node x.

To calculate a feature weight $w(f)$, we first consider a dataset of N benign HTTP flows, and R tree instances from a particular

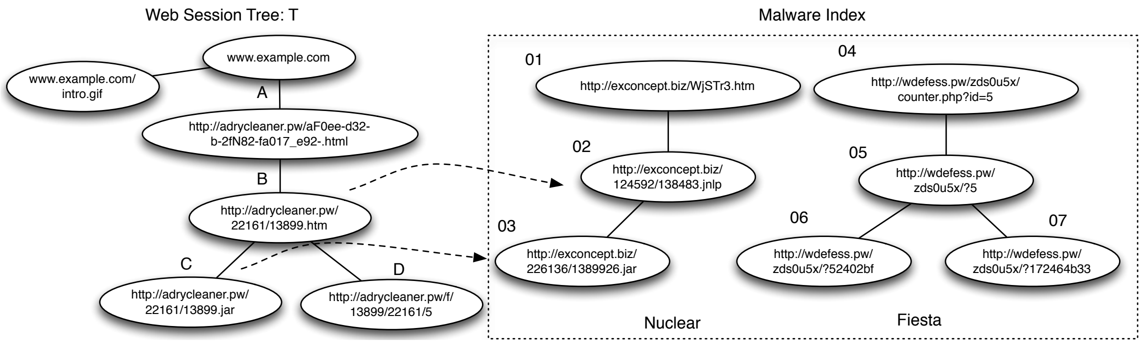

Figure 3: A simplified similarity search on the index. Web session tree T contains nodes that are similar to nodes of one of the Nuclear trees in the index. Those nodes in T are subsequently mapped to their corresponding nodes in the index to form subtrees.

exploit kit flavor (e.g., Nuclear, Fiesta, etc.). Let some feature f index r of the malicious trees in R and n of the benign flows in N. As such, $p = \frac{r}{R}$ is the probability that feature f indexes an exploit kit, while $q = \frac{(n-r)}{(N-R)}$ is the probability that f indexes a benign flow. Therefore, the weight of feature f becomes:

$$w(f) = log(\frac{p(1-q)}{(1-p)q}) = log(\frac{r(N-R-n+r)}{(R-r)(n-r)}).$$

When $r = 0$, i.e. feature f does not index any of malicious trees, the formulation is not stable; therefore, we apply the following modification as suggested by Robertson and Jones [32]:

$$w(f) = log(\frac{(r+1/2)(N-R-n+r+1/2)}{(R-r+1/2)(n-r+1/2)}).$$

Our technique requires a node-level similarity threshold for each exploit kit family stored in the malware index in order to determine that a node in T is similar to nodes in the index. To compute the thresholds, we compare the node similarities scores of each tree in the malware index, against all the other trees in the malware index that are in the same exploit kit family. An average node similarity score is calculated for each node in each tree in an exploit kit family. The node-level threshold for the kit is calculated by finding the node in the tree with the lowest average similarity score.

This process is presented in Algorithm 1. For pedagogical reasons, we use Fiesta tree samples from the malware index to illustrate the approach. For each tree t in the set of Fiesta trees, we first find all trees s that have a tree edit distance similarity score above zero (lines 3-5). For any node in t that has a similarity score above 0.1 with s, its score is recorded (lines 7-9). Finally, we store the minimum average score as the threshold for the kit. During the feature extraction stage, token and content-based features are ignored in order to provide a conservative lower bound on the threshold.

Algorithm 1 Finding the node level similarity threshold for the Fiesta exploit kit using the set of all Fiesta tree samples in the index

```
 1: $T_f \leftarrow$ set of all Fiesta Trees in Index
 2: $minval = 1.0$
 3: for all ( do $t \leftarrow T_f$)
 4:     for all ( do $s \leftarrow T_f$)
 5:         if $TreeSimScore(s,t) > 0.0$ then
 6:             for all ( do $n_s \leftarrow Node(s); n_t \leftarrow Node(t)$)
 7:                 if $score \leftarrow NodeSimScore(n_s, n_t) \geq 0.1$ then
 8:                     $n_t.totalScore += score$
 9:                     $n_t.numberScores += 1$
10:                 end if
11:             end for
12:         end if
13:     end for
14:     for all ( do $n_t \leftarrow Node(t)$ )
15:         $avg = n_t.totalScore/n_t.numberScores$
16:         if $avg \leq minval$ then
17:             $minval \leftarrow avg$
18:         end if
19:     end for
20: end for
21: $threshold = minval$
```

3.3.2 Structural Similarity Search

After a node level similarity search between a tree T (built from the network) and the trees in the malware index, there will be 0 or more nodes in T that are considered "similar" to nodes in the malware index. A node in tree T may be similar to multiple nodes in a single tree in the index or even in multiple trees. The next step is to extract the subtrees S within T that map to the corresponding trees in the index. Figure 3 shows a simplified example of a structural similarity search. Node B in tree T maps to node 02 based on node similarity for a Nuclear tree in the index. Similarly, node C in T maps to node 03. These node mappings are used to build subtrees of T that are compared to the corresponding trees in the index.

Subtrees from tree T are compared to the trees in the index using tree edit distance [11]. Tree edit distance uses the number of deletions, insertions, and label renamings to transform one tree into another. We enforce ancestor-descendant relationships in our setup. For example, if a node was an ancestor of another node in a tree in the index, the relationship must be maintained in the subtree S. As shown later, this restriction helps to reduce false detections. The result of the tree edit distance calculation is a structural similarity score between 0 and 1 that is then used to classify the subtree as either being benign or similar to a specific exploit kit.

4. DATASET AND TRAINING

The efficacy of our approach is evaluated using logs collected from a commercial HTTP proxy server (called BlueCoat) that monitors all web traffic for a large enterprise network. The proxy server records all client-based bidirectional HTTP/HTTPS flows from eight sensors at edge routers around the network and acts as a man-in-the-middle for HTTPS sessions providing a view into encrypted traffic. Each sensor saves flow records into its own set of hourly log files. Flows contain both TCP and HTTP headers.

For our first set of experiments, we analyzed 628 hours worth of labeled log data spanning different days during November 2013 and July 2014. The log files were chosen because they contained known instances of Nuclear, Fiesta, Fake, FlashPack, and Magnitude exploit kits along with several instances of a clickjacking [13] scheme that we refer to as ClickJack. Statistics for the dataset are summarized in Table 1 (labeled Dataset 1). We also utilized a separate three-week long dataset from January 2014 which was *unlabeled* to show the operational impact of our technique. Statistics for the dataset are also described in Table 1 (labeled Dataset 2) and are discussed in Section 6.

Table 1: Summary of datasets.

	Dataset 1	Dataset 2
Network sensors	8	8
Hours analyzed	628	3264
Client IP addresses	345K	> 300K
Bidirectional flows processed	800M	4B
HTTP tree structures processed	116M	572M

4.1 Implementation

The implementation is a multi-threaded application written in approximately 10,000 lines of `Python` and `C++` code. It processes archived bidirectional HTTP flows that are read and converted into web session trees on the fly while node and tree features are stored in the `Xapian` search engine. The prototype uses separate threads to read and parse each flow, to build HTTP web session trees, and to compare the most recently built tree to the malware index.

System Environment: All experiments were conducted on a multi-core Intel Xeon 2.27 GHz CPU with 500 GBs of memory and a 1 TB local disk. Notice that the platform is chosen because it facilitates our large-scale experiments by enabling multiple instances of the prototype to be run in parallel. The actual memory allocated for each prototype instance is 20G.

4.2 Building the Malware Index

As mentioned in Section 3.2, the malware index is essentially the "training data" used to detect malicious subtrees in the dataset. As such, the index is populated with exploit kit samples from a completely disjoint data source. We populated the malware index with exploit kit samples downloaded from a malware analysis website [7]. The operator collected HTTP traces of exploit kits using a honeyclient and stored them in a pcap format. We built a tool that transforms these traces into HTTP trees that are in turn indexed. The 3rd column of Table 2 provides a count of how many instances of each kit were downloaded and indexed. Note that none of the instances installed in the index appear in the proxy data logs. The clickjacking sample was downloaded from another website [25].

Table 2: Testing and training sets. Exploit kits collected from www.malware-traffic-analysis.net used to build the malware index.

Exploit Kit	Instances in Dataset 1	Instances in Malware Index
Fiesta	29	26
Nuclear	7	10
Magnitude	47	12
ClickJack	130	1
FlashPack	2	7
Fake	575	12

The second aspect of building the malware index is to calculate feature weights for all node features in the index when using the weighted Jaccard Index for node similarity. This requires malicious samples from the malware index as well as samples of normal traffic in order to determine how prevalent a feature is in both the malicious and benign dataset. In our experiment, we used 10 days worth of benign data from a single sensor in the BlueCoat logs to calculate feature weights. The benign data included over 4.4 million bidirectional flows.

Finally, we calculate the node similarity thresholds for each exploit using Algorithm 1 (§3.3.1). The thresholds for the weighted and non-weighted node similarity scores ranged between 0.2 to 0.25 depending on the exploit kit as shown in Table 3.

4.3 Establishing Ground Truth

In order to establish a ground truth as a test set for our experiments, we compiled a list of regular expressions from various sources in order to identify exploit kit instances in Dataset 1. First, we ran the Snort Sourcefire exploit kit regular expression rules from the Vulnerability Report Team [37] over the entire dataset. The ruleset included signatures for detecting exploit kits, such as Nuclear, Styx, Redkit, Blackhole, Magnitude, FlashPack, and Fiesta. We augmented these signatures with regular expressions gathered from a malware signatures website (www.malwaresigs.com) that included regular expressions for Fiesta, Angler, FlashPack, Styx, and Redkit. Through manual inspection of flows in Dataset 1 that match these signatures/regexes, we were able to identify several instances of the Fiesta, Nuclear, ClickJack, FlashPack, Fake, and Magnitude exploit kits (see the middle column of Table 2). False positives were painstakingly removed by grouping URLs by domain names, and by comparing them against publicly available blacklists and whitelists, including online searches against various API's engines (e.g., VirusTotal, GoogleSafe Browsing, URLQuery.net, Alexa, malwaredomainlist.com, and Google).

Table 3: Node-level thresholds computed by Algorithm 1.

Exploit Kit	Threshold (Weighted JI)	Threshold (JI)
Fiesta	0.25	0.25
Nuclear	0.23	0.25
Magnitude	0.25	0.25
ClickJack	0.25	0.25
FlashPack	0.23	0.2
Fake	0.23	0.25

Unfortunately, our analysis was conducted shortly after the author of the Blackhole and Cool exploit kits was arrested in Russia [36]. Hence, these exploit kits, which were once credited with over 90% of new infections [36], collapsed leaving attackers scrambling to find an alternative. Although, we were unable to obtain traces of the Blackhole or Cool exploit kits, we procured many instances of the Fiesta and Magnitude kits, which became prevalent after Blackhole's demise [30]. Recent studies [30, 9] show that there are approximately 6-8 exploit kit types dominating the Internet at any one time, accounting for the relatively small number of different but popular kits found on the analyzed network.

5. FINDING THE NEEDLE IN A HAYSTACK

In this section, we evaluate and compare our approach on Dataset 1 against the Snort Intrusion Detection System as well as two recent machine-learning approaches to detect exploit kit instances.

5.1 Comparison with Snort

In all cases, but FlashPack, the weighted and non-weighted node similarity approaches yielded the same results; therefore we leave indepth discussion of these approaches for Section 5.3.

■ *Fiesta*: In evaluating Fiesta, we compared our approach against the Snort rule 29443, which detects Fiesta outbound connections attempts. The rule focuses on the single flow related to the exploit payload and detects Fiesta instance by searching a particular alpha numeric pattern in the URL. As a result, it also flags 597 benign flows that match the regex pattern. On the contrary, our technique focuses on the structural path of flows taken to arrive at the exploit payload. As such, in our technique, not only are we able to eliminate these accidental matches that are unlikely to share similar structures with Fiesta instances, but also identify the exploit before the payload is reached, and even cases where no payload was downloaded at all. The results are summarized in Table 4.

Table 4 shows that using structure eliminated all 597 false positives flagged by the Snort rule and also identified cases that Snort missed. In most cases, our approach detected a Fiesta instance in as little as two or three nodes. Furthermore, it detected three instances that were not originally flagged in the ground truth, because our approach was able to detect the path of requests to the payload. In six cases, the exploit kit never reached a payload, and in another two, the payload string did not match Snort's regex. We missed two instances of Fiesta that accessed the same landing page but at different times. These instances were missed because there were no structures in the index similar enough to the instance to attain a structural score. There was no overlap between the false negatives missed by both techniques.

■ *Nuclear*: To track Nuclear, we used three Snort rules 28594, 28595, and 28596, which search for numeric `jar` and `tpl` file name of malicious payloads as well as specific directory structures in URLs. As noted in Table 4, the Snort signatures performed reasonably well for detecting all five Nuclear instances; because in all these cases, Nuclear was able to proceed to the payload-download stage. However, by looking for specific file types, these regexes missed an instance of Nuclear that was downloading a malicious `pdf` (which we detected). Furthermore, the generality of the signatures (e.g., matching numeric jar or tpl names) leads to 24 false alarms on legitimate websites that download benign jar files with numeric names. Our approach, on the other hand, strikes a better balance between specificity and generality. By leveraging structural properties of multi-stage exploit kits, it eliminates all false positive cases (which do not share similar tree structures with Nuclear exploit kits) and is able to generalize to new variation of exploit kits with previously unseen payloads. Although our approach failed to detect two instances of Nuclear that were structurally the same, that failure arose because our index did not have a similar example in the datastore.

The most interesting instance of Nuclear found in the data was downloaded through an advertisement on a popular foreign news site. That exploit successfully downloaded both a `Java` exploit and a malicious binary to the unsuspecting client machine.

■ *Magnitude*: In order to evaluate Magnitude, we utilized Snort rules 29188 and 28108, which search for hex encoded `eot` and `swf` files, respectively. Results for all techniques are shown in Table 4. The Snort rules generated over 60,000 false positives and missed an instance that did not download a payload while the classifier detected all exploit kit instances but with a high FP rate. By contrast, using the structure of correlated flows, we had zero false positives and zero false negatives.

■ *FlashPack*: Our empirical analysis shows that FlashPack is one of the more difficult exploit kits to detect because of its use of common `php` file names such as `index.php`, `flash.php`, and `allow.php`. Snort uses rule 29163 to identify a subset of these files (i.e., those which have a specific query string to reduce false positives). However, the query string can be easily manipulated by attackers to evade detection and it often varies across different FlashPack variations. As a result, the Snort rule was unable to detect the two instances of FlashPack variations in the data. We experimented with a much looser regular expression to identify all instances; however, it generated over 43,000 false positives.

Using our approach, we were able to identify both instances in the dataset, with only 68 false positives (weighted node similarity) and 109 false positives (non-weighted) — four orders of magnitude reduction over the loose regular expression. The added false positives in the non-weighted case are due to the increased number of node-level false matches in the non-weighted Jaccard Index calculation. FlashPack was the only exploit kit analyzed where setting a minimum *structural threshold* had a significant impact on the false positive rate (We return to that later in §5.3). The two true instances had similarity scores of 0.75 and 0.85 respectively. With a conservative structural similarity score of 0.5, the number of FPs is reduced to three (weighted) and 19 (non-weighted) (Table 4). Forensic analysis revealed that both instances of FlashPack were loaded through banner ads when two separate clients visited entertainment websites. In one of these cases, the exploit successfully downloaded both a malicious Flash file as well as a `Java` archive to the vulnerable client.

■ *ClickJack*: Clickjacking is a technique in which an attacker tries to fool a web user into clicking on a malicious link by injecting code or script into a button on a webpage [13]. To detect instances of the ClickJack kit, we loaded a single instance of its structure into the index and then performed searches on the entire dataset. There was no equivalent Snort rule for finding such an exploit and so a comparison to Snort was not possible. Our approach identified 130 instances of the clickjacking scheme with zero false positives and zero false negatives. Interestingly, our analysis found that the ClickJack subtree was the initial entry point into various exploits including an instance of the Magnitude exploit kit, and several trojans. With an online version of our approach, we would have been able to detect the exploit before it was downloaded.

■ *Fake - Installer*: Our final case study focuses on the Fake Installer exploit kit, which is an exploit that attempts to install a fake Adobe update for an unsuspecting client. This kit is identifiable by the `checker.php` file it uses to check the system and attempt a download of a malicious payload. This common file name can trigger an excessive number of false positives, so because of this, there was no corresponding Snort rule. We conducted our own analysis on our dataset specifically looking for the checker.php file and found 1,200 cases of this file in a three month period. Of those 1,200 cases, we were able to confirm 575 to be the Fake Installer. Utilizing our approach, we successfully identify all such cases with zero false positives and zero false negatives.

Summary: Table 4 summarizes the detection results of our approach and Snort. Regarding exploit kits for which Snort rules are available (i.e., Fiesta, Nuclear, Magnitude, and FlashPack), our structure similarity-based approach achieved a 95% detection accuracy while outperforming Snort (at 84%). Considering that false positives place undue burden on analysts to perform a deeper investigation on each reported incident, reducing false positives by over three orders of magnitude is a non-trivial improvement. In addition, our approach identified all instances of two exploit kits for which Snort rules were not available (i.e., Clickjacking and Fake). The approach reduces false positives by utilizing both content and structure, effectively creating a larger feature space.

Table 4: Comparison (weighted) to Snort signatures.

Exploit kits	**#**	**Structural Sim**			**Snort**		
		TPs	FPs	FNs	TPs	FPs	FNs
Fiesta	29	25	0	4	19	597	10
Nuclear	7	5	0	2	5	24	2
Magnitude	47	47	0	0	46	60000+	1
FlashPack	2	2	3	0	0	9	2
Total	85	79 (95%)	3	4	70 (84%)	60630+	13
ClickJack	130	130	0	0	-	-	-
Fake	575	575	0	0	-	-	-
Total	705	705 (100%)	0	0	-	-	-

5.2 Comparison with State of the Art

Next we compare our approach with a statistical classifier proposed by Ma et al. [21]. The classifier is based on Logistic Regression with Stochastic Gradient Descent using features similar to those described in Section 3.2. The classifier labels URLs as either malicious or benign and is trained with all 1,000 URLs used to build the malware index, as well as 10,000 benign URLs collected from BlueCoat logs with a $10x$ class weight applied to the "malicious" class. Parameters for the algorithm are tuned using a grid search and five fold cross validation on the would be training set. Results are shown in Table 5 indicating that the classifier performed well at detecting exploit kit instances. The classifier was able to detect two more instances of Fiesta than our approach because both clients visited a landing page for an exploit kit, but did not reach a payload, exposing no web structure for our technique to detect. In the case of Nuclear, the classifier was unable to identify the instances that only used `.tpl` and `.pdf` file types.

Unfortunately, the technique flagged over 500,000 URLs as malicious in Dataset 1. Through a painstaking analysis of the URLs using malware reports, blacklists, and google searches, we were able to confirm 4,000 of the URLs to be malicious — 2,500 of the URLs were associated with the exploits kits found as ground truth in Dataset 1 (Table 2), which were also detected by our approach. Note that Table 2 represents numbers of trees, with each tree containing multiple URLs. The other 1,500 URLS were comprised of web requests to algorithmically generated domain names used by botnets [41], phishing sites, and malware download sites and were unrelated to exploit kit traffic. False positives were attributed to many different websites including content distribution networks, URL shorteners, and advertising networks. Clearly, due to the high false positive rate, the approach of Ma et al. [21] is infeasible in an operational environment.

Table 5: Comparison (weighted) to binary URL classifier.

Exploit kits	**Ins-tances**	**Structural Sim**			**Classifier**		
		TPs	FPs	FNs	TPs	FPs	FNs
Fiesta	29	25	0	4	27	-	2
Nuclear	7	5	0	2	5	-	2
Magnitude	47	47	0	0	47	-	0
FlashPack	2	2	3	0	2	-	0
ClickJack	130	130	0	0	130	-	0
Fake	575	575	0	0	575	-	0
Total	790	784 (99%)	3	6	786 (99%)	500,000+ (URLs)	4

Recently, Stringhini et al. [35], Mekky et al. [22], Cova et al. [5], Eshete and Venkatakrishnan [8] proposed detecting malicious websites by counting the number of HTTP redirects (i.e., 302, javascript, or HTML) to hop from a compromised website to the malicious exploit. The key insight is that attackers utilize statistically more intermediate HTTP redirects than benign traffic in order to avoid detection. Our intention was to provide a comparative analysis to Stringhini et al. [35], but unfortunately, the approach of Stringhini et al. [35] requires modeling a diverse set of redirect chains of users visiting the same malicious websites with different environments (e.g. OSes and browsers) at geographically dispersed locations. Given that such widely heterogenous environments are not available in most enterprises, we evaluate the utility of using redirects as a main feature to detect exploit kits in traffic by exploring the full packet payload HTTP traces associated with 110 exploit kit instances. The instances included 14 distinct exploit kits: Angler, Blackhole, Dotka Chef, Fake, Fiesta, Flashpack, Goon, Hello, Magnitude, Neutrino, Nuclear, Styx, Sweet Orange, and Zuponic.

Redirection chains were built from each trace by extracting server and HTML (meta tag) redirects. Additionally, we manually analyzed a subset of 50 traces using an instrumented HTML parser, javascript engine(Rhino) and DOM (envjs) in order to build chains that included javascript redirections. We found that the traces had relatively short redirection chains, and the length the chain was dictated by the type of exploit kit. Exploit kits such as Blackhole, Nuclear, Fiesta, Goon DotkaChef, Fake, and Sweet Orange consistently had a single indirection to the exploit kit server. Indeed, server and meta redirections were rare with the main form of redirection being an `iframe` injection into the compromised site, or a `javascript` injection that built an `iframe`. Magnitude, Angler, Flashpack, Zuponic and Neutrino saw anywhere from 1 to 3 redirects with a combination of server, meta and javascript redirects. In fact, Styx was the only instance that had more than 4 redirects. These results are in stark contrast to the results of Mekky et al. [22] that show that over 80% of all malicious chains have 4 or more server redirects or that the average number of exploit kit server redirects are five [8].

Not only are we not seeing large redirect chains for exploit kits, but we are also seeing comparable length redirect chains in benign traffic due primarily to advertising networks. We built server and meta redirection chains on 24 hours worth of data from a large enterprise network consisting of 12 million bidirectional HTTP flows. In that time period, 400,000 redirection chains were generated including 35,000 chains of length 2 to 5, making the redirection feature prone to false positives. By contrast, our approach can utilize redirection chains, but focuses on the process by which an exploit kit attempts to compromise a host and models that into a tree-like structure in order to reduce false positives.

5.3 Findings and Discussion

We now take a closer look at why the use of structural information (especially, the ancestor-descendant relationship) is important in reducing false positives. We begin our analysis by focusing on the node-level similarity scores using the weighted and non-weighted Jaccard Index calculated between the HTTP flows in the archival logs (i.e., Dataset 1 in Table 2) and those in the malware index. The results are shown as a cumulative distribution function in Figure 4. Notice that over 98% of the flows in the dataset had a similarity score below the conservative lower bound thresholds (of 0.23/0.2) derived from Algorithm 1 while all nodes associated with malicious trees had a node similarity score of 0.22 or higher.

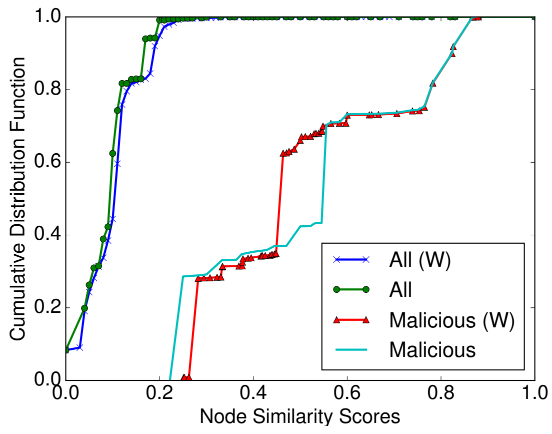

Figure 4: The CDF of node similarity scores between our test dataset and the malware index. "All" represents similarity scores between all nodes in the dataset and the malware index, while "Malicious" represents the node scores for trees in the dataset that were flagged as malicious(W = weighted).

The similarity scores for both node similarity metrics followed the same distribution; however, the non-weighted Jaccard Index generated on average lower similarity scores than the weighted approach (weighted mean = 0.10, non-weighted = 0.09) with similar standard deviations. As can be seen from Figure 4, the similarity gap between the malicious and benign nodes is smaller in the non-weighted case than in the weighted case. This leads to more node-level false positives, and, as a result, structural false positives as seen in the case of FlashPack. Intuitively, because the weighted Jaccard Index is weighted according to the importance of the feature, an unweighted version will be more likely to have false positives due to common features that are prevalent both in benign and malicious nodes. Even though the weighted version provides marginally better accuracy, the non-weighted Jaccard Index may be more desirable from an operational perspective because it does not require any feature training.

As shown in Table 6, there were a large number of false positives if we considered only node-level similarity (like Snort signatures that focus only on individual flows) for both weighted and non-weighted similarities. The false positive rate started to decrease when considering *multiple nodes* in a tree (without considering structure), as the probability of a benign website having two or more nodes in the same tree that match malicious patterns was an order of magnitude smaller. The false positive rate decreased further by another order of magnitude once a *structural score* was established using tree edit distance. After imposing the *ancestor-descendent requirement* on the tree structure, the false positives were reduced to 68 for the total of over 800 million flows. The results show that tree structure is the primary determining factor in reducing false positives.

Table 6: FPs for single node matching, multi-node matching without considering structure, structural similarity, and structural similarity with ancestor-descendant requirement.

	Threshold (Alg 1) non-weighted	**Threshold (Alg 1) weighted**	**Tight Threshold weighted**
Single Node	2,141,493	360,150	141,130
Multi-node (no structure)	79,321	32,130	5,878
Structural	5,967	3,800	420
Structural (w/ restriction)	109	68	68

As shown in Table 6 there is a several orders of magnitude reduction in the number of nodes (flows) that are similar to nodes in the index, w.r.t the total number of nodes (flows) in a given dataset (Table 1). We can leverage this result, by only building trees for flow clusters that have multiple similar nodes in common with a tree in the malware index, thus enabling us to scalably apply much more computationally expensive (and correct) tree building techniques to the wire (i.e., [23]) when full payloads are available. Table 6 also shows the detection rates under the optimal tight node-level similarity thresholds using weighted similarity. This bound is the maximum node similarity threshold allowed to still detect all true positives, and was calculated using the ground truth dataset. Even with the optimal bound, structural information was still needed to reduce the false positives.

Our empirical analysis also showed that in the majority of cases, a relatively low minimum structural threshold (less than 0.05) for the tree-similarity score was sufficient because the flagged tree was indeed malicious in almost every case. The structural similarity threshold is specific to the similarity metric chosen and was set conservatively low to maximize true positives with few false positives, creating a clear separation between benign and malicious cases. Figure 5 shows the cumulative distribution of the tree edit distance scores for the malicious subtrees analyzed. The scores ranged anywhere from 0.2 to a perfect 1.0 due to a few factors. First, in some cases there may be multiple nodes added or missing from the subtree as compared to the malware index, causing an imperfect score. The second reason was that, especially in the case of ClickJack, the exploit may lead into other exploits or websites causing the subtree in the dataset to look different from any of the ones in the malware index. Taken together, these findings underscore the power of using structural information and subtree mining, particularly when there may be subtrees that are incomplete or contain previously unseen nodes compared to those encoded in the index. The combination allows us to attain maximum flexibility and reduce both false negatives and false positives over contemporary approaches.

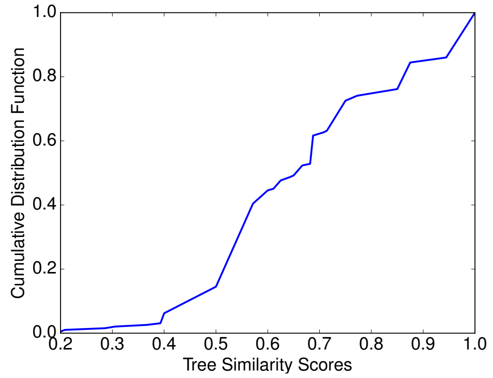

Figure 5: The CDF of tree similarity scores for malicious subtrees.

6. OPERATIONAL DEPLOYMENT

To further demonstrate the utility of our approach in a large enterprise environment, we analyzed three consecutive weeks of BlueCoat logs from January 6-31, 2014 (Dataset 2 in Table 1) using the weighted version of our approach. During the time period, over 4 billion bidirectional flows and 572 million HTTP trees were generated and analyzed using a malware index consisting of the Fiesta, Nuclear, Fake, ClickJack, and Magnitude exploit kits.

During the deployment we were able to identify 28 exploit kit instances with no false positives, compared with Snort signatures that generated over 22K false positives and missed most of the Fiesta instances, as shown in Table 7. Two of the Fiesta instances downloaded malicious `Java` files, while two others downloaded spyware. The Nuclear instance successfully downloaded a malicious `PDF` file followed by a malicious binary. We also discovered that two of the Clickjacking instances downloaded Popup Trojans. By contrast, the URL classifier of Ma et al. [21] generated an average of 143,000 alerts per day for a total of 3.6 million alerts in the month. Unfortunately, the sheer volume of alerts made it infeasible to vet each flagged URL.

Table 7: Live comparison to Snort signatures.

Exploit kits	**Structural Similarity**		**Snort**		
	TPs	FPs	TPs	FPs	FNs
Fiesta	20	0	2	340	≥ 18
Nuclear	1	0	1	0	-
Magnitude	1	0	1	22,224	-
Clickjacking	6	0	N/A	N/A	-
Fake	0	0	N/A	N/A	-

The fact that we were able to successfully detect these abuses on a large enterprise network underscores the operational utility of our technique. Indeed, one of the main motivating factors for pursuing this line of research and subsequently building our prototype was the fact that the high false positives induced by existing approaches made them impractical to network operators at our enterprise — who inevitably disabled the corresponding signatures or ignored the flood of false alerts altogether.

From an operational perspective, speed is as equally important as accuracy in order to keep up with the live traffic in a large enterprise network. Therefore, to assess our runtime performance, we evaluated the processing speed for the various components when processing one days worth of traffic across all eight sensors. Note that eight prototype instances were run — one for each sensor. The experiment shows that a single instance of our current prototype is able to process an entire day of traffic in 8 hours. Figure 6 illustrates the performance breakdown of different components of our prototype, indicating that on average, the prototype can parse 3.5K flows per second (302M flows per day), build trees at a rate of approximately 350 per second and conduct the similarity search at a rate of 170 trees per second. Profiling the similarity search module showed that over half the runtime was spent performing feature extraction and memory allocation, while only 5% of the time was spent searching the index. Sensors 5, 6, and 8 were slower than the other sensors because they received a larger portion of the traffic.

Lastly, we note that although our prototype was able to keep up with the average volume of traffic in the target enterprise, the same was not true at peak load. Statistics collected from one day of traffic across all eight sensors showed that at its peak, the network generated 6,250 flows and 550 trees per second. While our prototype falls short of processing at that speed, we note that by design, all the components (e.g., flow parsing, tree building and feature extraction) are parallelizable; as such, with modest hardware provisions we believe our prototype could efficiently handle the peak loads and operate in real-time. We leave this for future work.

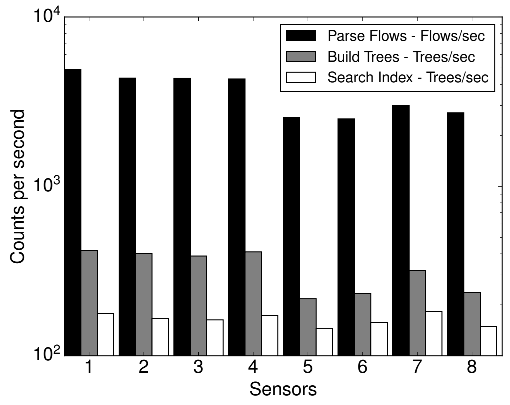

Figure 6: The performance of bidirectional flow parsing, tree building, and malware searching for one day of data across 8 sensors.

7. LIMITATIONS

As with any security solution, attackers will inevitably seek ways to bypass it. An obvious evasive strategy would be to hinder our ability to build subtrees from HTTP flows by using JavaScript and other obfuscation techniques that hide the relationship (e.g., redirection, reference) between HTTP flows. As mentioned previously, we believe our two step similarity algorithm allows us to significantly reduce the overall number of trees that need to built, allowing more computationally expensive and correct techniques to be used such as dynamic analysis [23], JavaScript de-obfuscation [19, 40], and statistical means [12, 24, 44] — all of which could be easily adopted in our setting to thwart evasive techniques. Moreover, in many enterprise environments, there is strict control over the software configuration of client devices in the network, and as such, a mandatory browser plugin could be enforced to make building web session trees easier than our current approach. Nevertheless, we reiterate that the focus of this work is not to build better HTTP trees, but to demonstrate the benefits of a tree structure-based detection approach in reducing false negatives and false positives.

In addition, because our approach relies on *node-level* and *structure-level* similarity to detect exploit kits, a skilled adversary might attempt to circumvent similarity matching by obfuscating flow features and dramatically modifying tree structures. Although the approach suggested herein is no silver bullet, we believe it raises the bar for attackers and makes evasion more difficult. For instance, by using an edit-distance based subtree mining algorithm to compare observed session trees, our approach offers resilience to common obfuscation and variation techniques (e.g., adding redirection nodes or changing malicious payloads). More importantly, a structural similarity based approach provides security analysts with flexibilities in tuning the thresholds such that changes to a few nodes in the web session trees are unlikely to significantly influence the matching results. On the other hand, generating functionally equivalent but structurally distinct exploit paths would be a non-trivial task for attackers. As future work, we plan to quantify our resilience against such obfuscation strategies.

From an operational perspective, the fact that our approach involves some manual effort on the part of the analyst (e.g., to find and install representative examples of exploits kits into the malware index) might appear as a limitation. Indeed, like most tasks in network security, performing this particular step requires some expertise and domain knowledge. That said, the burden on the operator could be lessened with automated techniques for building these indices, for example, from data made available through websites like threatglass.com. Furthermore, techniques applied in automated signature generation [42] may be useful.

Finally, like all network-based detection techniques that require packet inspection, the approach herein cannot operate on encrypted traffic. For many enterprises, however, the ability to inspect encrypted traffic is enforced at the border by using proxy servers specifically designed to decrypt and monitor encrypted traffic. This was precisely the case for the enterprise evaluated in this paper.

8. CONCLUSION

In this paper, we present a network-centric approach that uses structural similarity to accurately and scalably detect web-based exploit kits in enterprise network environments. By exploiting both the content and the structural interactions among HTTP flows, our approach allows us to not only reason about the likelihood of a sequence of HTTP flows being malicious, but to also pinpoint the exact subset of flows relevant to malvertising. By modelling HTTP traffic as trees, we can also determine from which root sites, or advertising networks, an exploit kit was launched. Our prototype implementation, which was evaluated on real world data collected from a large-scale enterprise network, worked remarkably well. In particular, our empirical results show significant improvement over the state-of-the-art methods in terms of false positive and false negative rates across a variety of exploit kits. Lastly, a preliminary analysis in an operational deployment demonstrates that our techniques can easily scale to handle massive HTTP traffic volumes with only modest hardware requirements.

9. ACKNOWLEDGMENTS

We express our gratitude to the Computer Science networking staff (especially, Murray Anderegg and Bil Hayes) for their efforts in deploying some of the infrastructure used in this study. We also thank Douglas L. Schales for help with data procurement as well as Jan Werner, Kevin Snow, Andrew White, and the anonymous reviewers for their insightful comments. This work is supported in part by the National Science Foundation (with a supplement from the Department of Homeland Security) under award numbers 1127361 and 1421703. 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.

10. REFERENCES

[1] 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.

[2] 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.

[3] J. Clark. Malicious javascript flips ad network into rentable botnet, July 2013. URL http://goo.gl/8mFLvQ.

[4] S. Cohen. Indexing for subtree similarity-search using edit distance. In *ACM SIGMOD Conference on Management of Data*, 2013.

[5] 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.

[6] 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.

[7] B. Duncan. Malware-traffic-analysis.net blog, July 2014. URL http://goo.gl/fXdSZz.

[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] 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.

[10] M. Hadjieleftheriou and D. Srivastava. Weighted set-based string similarity. *IEEE Data Engineering.*, 33(1), 2010.

[11] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *ACM Conference on Computer and Communications Security*, 2009.

[12] X. Hu, M. Knysz, and K. G. Shin. Rb-seeker: Auto-detection of redirection botnets. In *Symposium on Network and Distributed System Security*, 2009.

[13] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: attacks and defenses. In *USENIX Security Symposium*, 2012.

[14] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *ACM Internet Measurement Conference*, 2011.

[15] 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.

[16] R. Lemos. The doubleclick attack and the rise of malvertising, Dec. 2010. URL http://goo.gl/1HzmLF.

[17] 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.

[18] 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.

[19] G. Lu and S. Debray. Automatic simplification of obfuscated javascript code: A semantics-based approach. In *Conference on Software Security and Reliability*, 2012.

[20] 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.

[21] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Learning to detect malicious urls. *ACM Transactions on Intelligent Systems Technology*, 2(3), May 2011.

[22] 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.

[23] C. Neasbitt, R. Perdisci, K. Li, and T. Nelms. Clickminer: Towards forensic reconstruction of user-browser interactions from network traces. In *ACM Conference on Computer and Communications Security*, 2014.

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

[25] J. Nieto. Zeroaccess trojan - network analysis part ii, July 2013. URL http://goo.gl/LYssOV.

[26] 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.

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

[28] PWC. IAB internet advertising revenue report: 2012 full year results. Technical report, PricewaterhouseCoopers, Interactive Advertising Bureau, Apr. 2013.

[29] D. Raywood. Major league baseball website hit by malvertising that may potentially impact 300,000 users, June 2012. URL http://goo.gl/upKVXe.

[30] S. M. G. T. Response. Six months after blackhole: Passing the exploit kit torch, Apr. 2014. URL http://goo.gl/nAsxj0.

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

[32] S. E. Robertson and K. S. Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27(3), 1976.

[33] M. J. Schwartz. Android malware being delivered via ad networks, Aug. 2013. URL http://goo.gl/CrfKzo.

[34] B. Stock, B. Livshits, and B. Zorn. Kizzle: A signature compiler for exploit kits. Technical Report MSR-TR-2015-12, Microsoft Research, February 2015.

[35] 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.

[36] Trend Micro. The aftermath of the blackhole exploit kit's demise, Jan. 2014. URL http://goo.gl/DsjYUp.

[37] S. VRT. Snort vrt signatures, July 2014. URL http://www.snort.org/vrt.

[38] 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.

[39] L. Xu, Z. Zhan, S. Xu, and K. Ye. Cross-layer detection of malicious websites. In *ACM Conference on Data and Application Security and Privacy*, 2013.

[40] W. Xu, F. Zhang, and S. Zhu. Jstill: Mostly static detection of obfuscated malicious javascript code. In *ACM Conference on Data and Application Security and Privacy*, 2013.

[41] S. Yadav, A. K. K. Reddy, A. N. Reddy, and S. Ranjan. Detecting algorithmically generated malicious domain names. In *ACM Internet Measurement Conference*, 2010.

[42] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *USENIX Security Symposium*, 2005.

[43] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna. The dark alleys of madison avenue: Understanding malicious advertisements. In *ACM Internet Measurement Conference*, 2014.

[44] H. Zhang, D. D. Yao, and N. Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *ACM Symposium on Information, Computer and Communications Security*, 2014.