# USING CONTEXT TO IMPROVE NETWORK-BASED EXPLOIT KIT DETECTION

Teryl Taylor

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2016

Approved by:

Fabian Monrose

John McHugh

Jay Aikat

Alex Berg

Ting Wang

## ABSTRACT

TERYL TAYLOR: Using Context to Improve Network-based Exploit Kit Detection
(Under the direction of Fabian Monrose)

Today, our computers are routinely compromised while performing seemingly innocuous activities like reading articles on trusted websites (e.g., the NY Times). These compromises are perpetrated via complex interactions involving the advertising networks that monetize these sites. Web-based compromises such as exploit kits are similar to any other scam — the attacker wants to lure an unsuspecting client into a trap to steal private information, or resources – generating 10s of millions of dollars annually. Exploit kits are web-based services specifically designed to capitalize on vulnerabilities in unsuspecting client computers in order to install malware without a user's knowledge. Sadly, it only takes a single successful infection to ruin a user's financial life, or lead to corporate breaches that result in millions of dollars of expense and loss of customer trust.

Exploit kits use a myriad of techniques to obfuscate each attack instance, making current network-based defenses such as signature-based network intrusion detection systems far less effective than in years past. Dynamic analysis or honeyclient analysis on these exploits plays a key role in identifying new attacks for signature generation, but provides no means of inspecting end-user traffic on the network to identify attacks in real time. As a result, defenses designed to stop such malfeasance often arrive too late or not at all resulting in high false positive and false negative (error) rates. In order to deal with these drawbacks, three new detection approaches are presented.

To deal with the issue of a high number of errors, a new technique for detecting exploit kit interactions on a network is proposed. The technique capitalizes on the fact that an exploit kit leads its potential victim through a process of exploitation by forcing the browser to download multiple web resources from malicious servers. This process has an inherent structure that can be captured in HTTP traffic and used to significantly reduce error rates. The approach organizes HTTP traffic into tree-like data structures, and, using a scalable index of exploit kit traces as samples, models the detection process as a subtree similarity search problem.

The technique is evaluated on 3,800 hours of web traffic on a large enterprise network, and results show that it reduces false positive rates by four orders of magnitude over current state-of-the-art approaches.

While utilizing structure can vastly improve detection rates over current approaches, it does not go far enough in helping defenders detect new, previously unseen attacks. As a result, a new framework that applies dynamic honeyclient analysis directly on network traffic at scale is proposed. The framework captures and stores a configurable window of reassembled HTTP objects network wide, uses lightweight content rendering to establish the chain of requests leading up to a suspicious event, then serves the initial response content back to the honeyclient in an isolated network. The framework is evaluated on a diverse collection of exploit kits as they evolve over a 1 year period. The empirical evaluation suggests that the approach offers significant operational value, and a single honeyclient can support a campus deployment of thousands of users.

While the above approaches attempt to detect exploit kits before they have a chance to infect the client, they cannot protect a client that has already been infected. The final technique detects signs of post infection behavior by intrusions that abuses the domain name system (DNS) to make contact with an attacker. Contemporary detection approaches utilize the structure of a domain name and require hundreds of DNS messages to detect such malware. As a result, these detection mechanisms cannot detect malware in a timely manner and are susceptible to high error rates. The final technique, based on sequential hypothesis testing, uses the DNS message patterns of a subset of DNS traffic to detect malware in as little as four DNS messages, and with orders of magnitude reduction in error rates.

The results of this work can make a significant operational impact on network security analysis, and open several exciting future directions for network security research.

To my friends and family.

## ACKNOWLEDGEMENTS

I was supported by so many different people during my time at the University of North Carolina at Chapel Hill. First, I would not be here if it was not for John McHugh who encouraged me to pursue a Ph.D, and was there for many fruitful discussions. I also must express my gratitude to my advisor, Fabian Monrose, for his mentorship through all steps of this program. He learned my weaknesses and used them to make me a better researcher. He was there every step of the way to ensure that I was successful.

I also want to acknowledge my committee members, Ting Wang, Alex Berg, and Jay Aikat, for providing excellent discussion, proof reading, and support. There were also many hours spent with my fellow doctoral students, Srinivas Krishnan, Kevin Snow, and Andrew White, and collaborators, Xin Hu, Marc Ph. Stoecklin, Jiyong Jang, Reiner Sailer, Douglas Schales, Jan Werner, Nathan Otterness, and Scott Coull. UNC Information Technology staff members Murray Anderegg and Bil Hayes were always there when I had a technical problem, and Stan Waddell, Alex Everett, Jim Gogan, and Danny Shue graciously provided data to make this dissertation possible.

I could not have done this research without the financial support of the National Science and Engineering Research Council of Canada (NSERC), the University of North Carolina at Chapel Hill, the National Science Foundation (NSF), the Department of Homeland Security, IBM Research and Cisco Systems.

Finally, my friends and family helped me through this program with their support. I want to thank Victor Heorhiadi, Christian F. Orellana, Edward Marlowe, Oluwafemi Alabi, Istvan Csapo, Kory Menke, Mike Seman, and Jocelyn Friedman for always being there. I also need to thank my mom and dad, and sister Marlo for their encouragement and patience, and Christina Rader for being my biggest cheerleader.

## PREFACE

This dissertation is original, unpublished, independent work by the author, Teryl Taylor, except where due reference is made in the text of the dissertation.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| AGD | Algorithmically Generated Domain name |
| API | Application Program Interface |
| CAS | Compare And Swap |
| CDF | Cumulative Distribution Function |
| CDN | Content Distribution Network |
| CPU | Central Processing Unit |
| DGA | Domain-name Generation Algorithm |
| DHCP | Dynamic Host Configuration Protocol |
| DNS | Domain Name System |
| DOM | Document Object Model |
| EXE | EXEcutable |
| FN | False Negative |
| FP | False Positive |
| FQDN | Fully Qualified Domain Name |
| GB | GigaByte |
| HTTP | HyperText Transfer Protocol |
| HTTPS | HyperText Transfer Protocol Secure |
| IDS | Intrusion Detection System |
| IP | Internet Protocol |
| JI | Jaccard Index |
| LRU | Least Recently Used |
| MX | Mail eXchange |
| NIDS | Network Intrusion Detection System |
| NX | Non-eXistent |
| P2P | Peer-to-Peer |
| PEB | Process Environment Block |
| PTR | PoinTeR |
| ROC | Receiver Operating Characteristic |

| | |
|---|---|
| SSL | Secure Sockets Layer |
| TCP | Transmission Control Protocol |
| TLD | Top Level Domain |
| TP | True Positive |
| TRW | Threshold Random Walk |
| TTL | Time To Live |
| UDP | User Datagram Protocol |
| URL | Uniform Resource Locator |
| VM | Virtual Machine |
| WJ | Weighted Jaccard index |

## LIST OF SYMBOLS

| | |
|---|---|
| $B_t$ | A blanket of $t$ — i.e., the set of all supertrees of $t$ that have one more node than $t$ |
| $D$ | A database of trees |
| $E$ | A set of edges nodes |
| $L$ | A labeling function mapping a tree node to its label |
| $\Sigma$ | An alphabet of tree labels |
| $S$ | A Subtree |
| $T$ | A labeled tree |
| $v_x$ | Tree node $x$ |
| $V$ | A set of tree nodes |
| $\uparrow$ | A backwards traversal from child to parent |

# CHAPTER 1: INTRODUCTION

Today, our computers are routinely compromised while performing seemingly innocuous activities like reading articles on trusted websites (Zarras et al., 2014) (e.g., the NY Times). 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 cyber criminals 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 (PWC, 2013), and its wide-spread reach makes it an excellent target for fraudsters. Furthermore, the 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 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 attackers posing as legitimate advertisers (Lemos, 2010). Such abuses are not isolated incidents and so-called *malvertising* has plagued many popular websites (Raywood, 2012), exploited mobile devices (Schwartz, 2013), and has even been utilized as vessels for botnet activity (Clark, 2013). 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.

According to Grier et al. (2012); De Maio et al. (2014), approximately 40% of these web-based attacks are launched by exploit kits, which are web services specifically designed to deliver malicious payloads (e.g., bots) to unsuspecting client machines. Exploit kits, such as Fiesta and Blackhole represent an entire software-as-a-service subindustry worth millions of dollars. For example, Cisco estimates that the popular exploit kit Angler is responsible for over 90,000 infection attempts a day and generates revenues of $60 million annually from ransomware alone (Biasini et al., 2015).

The exploitation of a user's system typically follows a four-step process as shown in Figure 1.1. In step ❶, 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 (step ❷) 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 (step ❸), which is then used to select a file such as Flash or JAVA, that can exploit the system configuration. The downloaded file exploits a vulnerability on the system that allows the attacker to install a malicious binary (step ❹) or otherwise control the victim's machine.

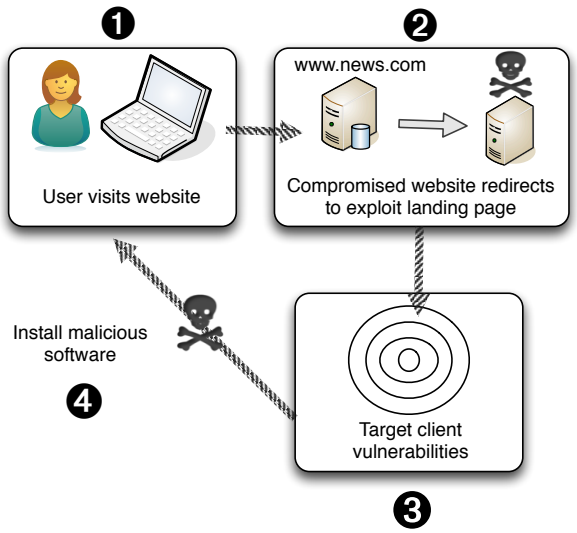



Figure 1.1: A typical process to exploit a victim's machine.

The malicious binary installed on the client's computer is typically designed to steal sensitive data or utilize system resources to perform some revenue generating tasks such as spamming, click fraud, ransoming data, or mining bitcoins. These binaries may enroll the host machine (called a bot) in a larger pool of infected machines to form a "botnet". A botnet is controlled by one or more command-and-control servers that relay commands and binary updates to members of the pool, allowing botnet owners to reap the financial rewards of the combined efforts of the bots. In order to maintain contact with the command-and-control server, the bot has some dynamic mechanism for learning the server's IP address. One of the key ways in which this is done is through abusing the domain name system (DNS) by using a domain name generation algorithm (DGA) to generate thousands of random domain names. The bot queries each domain name looking for a valid address for its command-and-control server. Only one of the domain names is registered and those queries that do not elicit an IP address return a non-existent or NX response.

Security analysts typically defend enterprise networks from these attacks using network monitoring devices (such as intrusion detection systems) that search network traffic as it passes through the network's

edge for signature matches or known malicious domain names. Network-based detection approaches are popular because network operators often do not have control over the devices that join their network, and malware can disable critical client-side protections such as anti-virus, making detection difficult. Attackers 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 look "normal" in the context of surrounding traffic. Furthermore, current network-based detection research has ignored the operational impact proposed solutions have on a security analyst's ability to effectively do their job. Current approaches have high false positive and false negative rates that place a heavy burden on the security analyst who must vet each false positive, and clean up the potential mess of a missed detection. Indeed, even the signatures heavily used by commercial products arrive too late to be effective.

This is not to say that current approaches are not useful. Signatures are a key cog in defending our networks and will not be replaced any time soon; however, more research needs to be done to address their limitations. In this work, I describe the characteristics of an ideal network-based exploit detector and explore the key operational drawbacks of current state-of-the-art approaches that cause them to fall short of the ideal case — i.e., high error rates, timeliness of discovery, lack of necessary and sufficient features, and inability to detect new previously unseen attacks. I then propose new novel detections techniques to help reduce the gap between the ideal case and current conditions. The techniques rely on utilizing structural interactions inherent in both HTTP and DNS traffic in order to detect both exploit kits and their corresponding bots.

The web structure of HTTP traffic is an important feature for detecting exploit kits because the structure encodes the inherent process outlined in Figure 1.1 that is not inherent in benign traffic, thus reducing misclassification rates. Exploit kits are also out-of-the-box solutions with infrastructures of hundreds of machines (Biasini et al., 2015). As a result, it is difficult for the attacker to change the structure for each exploit kit instance, and structure stays relatively constant across instances. Finally, attackers actually enforce a web structure by spreading an exploit across the multiple web files that are downloaded by the client during infection. This is done so that the defender cannot simply analyze an exploitable file in isolation for detection, but must analyze all web files at once.

*I hypothesize that by modelling the HTTP traffic associated with the exploitation process as a tree structure, one can provide enough context related to web-based attacks, in order to reduce the false positive rates by orders of magnitude over current state-of-the-art approaches. Furthermore, by utilizing such*

*context and utilizing virtualization, one can reduce false positive rates and detect previously unseen attacks as they occur. Finally, by leveraging the contextual patterns of NX DNS traffic, one can detect bots — that abuse the domain name service by using domain generated algorithms — before they contact their command-and-control server using a single computer on a large enterprise network.*

To support this assertion, the dissertation first motivates the problem by discussing the exploit kit attack model, and then proposes three novel network-based techniques to detect such exploits. In each case, I demonstrate the effectiveness of the technique empirically using large-scale real-world network datasets, and compare outcomes with existing approaches. The techniques are outlined below.

### 1.1 Detecting Exploit Kits via Subtree Similarity Search

One of the key problems with current network-based detection techniques is that they have high false positive and false negative rates (i.e, error rates). In Chapter 4, I propose a new approach for detecting exploit kits by leveraging the inherent structural patterns in HTTP traffic to classify exploit kit instances. The key insight is that an exploit kit leads the browser to download payloads using multiple requests from malicious servers. These interactions are captured in a "tree-like" form, and using a scalable index of malware samples, the detection process is modeled 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. I show that the 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 the analysis period.

### 1.2 Detecting Exploit Kits via Context and Virtualization on the Wire

While utilizing structure can vastly improve detection rates over current techniques, it does not go far enough in helping defenders detect new, previously unseen attacks. 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 Chapter 5, I introduce the design and implementation of a novel framework for adapting honeyclient-based systems to operate on-the-wire at scale. The framework captures and stores a configurable window of reassembled HTTP objects network-wide, uses lightweight content rendering to establish the chain of requests leading up to a suspicious event, then serves the initial response content back to the honeyclient system on an isolated network. The framework is evaluated by analyzing a diverse collection of web-based exploit kits as they evolve over a one year period. Case studies provide interesting insights into the behavior of these exploit kits. The empirical evaluations suggest that the approach offers significant operational value, and a single honeyclient server can readily support a campus deployment of thousands of users.

### 1.3 Detecting Network Malfeasance via Sequential Hypothesis Testing

Once the attacker has successfully dropped a malicious binary on a client machine, there are relatively few network-level clues available for the defender. However, the domain name system plays a vital role in the dependability and security of modern network and can act as a bellweather to detect such activities. Attackers have turned their attention to the use of algorithmically generated domain names (AGDs) in an effort to circumvent network defenses; however, because such domain names are increasingly being used in benign applications, this transition has significant implications for techniques that classify AGDs based solely on the format of a domain name. To highlight the challenges they face, Chapter 6 examines contemporary approaches and demonstrates their limitations. These shortcomings are addressed by proposing an online form of sequential hypothesis testing that classifies clients based solely on the non-existent (NX) responses they elicit. Evaluations on real-world data show that the technique outperforms existing approaches, and for the vast majority of cases, it can detect malware before they are able to successfully rendezvous with their command- and-control servers.

### 1.4 Innovations

In summary this dissertation makes the following contributions:

1. Chapter 3 presents the first large-scale comparison of subtree mining algorithms on a variety of real-life datasets. In this dissertation, HTTP traffic is modeled as a tree structure. Current literature does not provide insight into whether subtree mining algorithms are appropriate mining web traffic. As a result,

I run four state-of-the-art subtree mining algorithms on 10 real life datasets and provide insight into their characteristics and bottlenecks.

2. Chapter 4 presents a new online technique for detecting exploit kits that uses the tree structure of HTTP traffic to reduce (by orders of magnitude) the false positive rates over existing online approaches. The approach also provides a novel solution to the subtree similarity search problem, whereby, each tree node represents a high dimensional feature space. A version of this work appeared in:

    - Taylor, T., Hu, X., Wang, T., Jang, J., Stoecklin, M., Monrose, F., and Sailer, R. (2016a). Detecting malicious exploit kits using tree-based similarity searches. In *ACM Conference on Data and Application Security and Privacy*

3. Chapter 5 proposes a novel model for analyzing HTTP traffic scalably on a network using a honeyclient, as well as, a trigger and replay mechanism in a controlled environment. By utilizing a honeyclient, one can execute traffic from potentially malicious websites and monitor system-level changes to improve detection rates over the current state-of-the-art. A version of the work appeared in:

    - Taylor, T., Snow, K. Z., Otterness, N., and Monrose, F. (2016b). Cache, trigger, impersonate: Enabling context-sensitive honeyclient analysis on-the-wire. In *Symposium on Network and Distributed System Security*

4. Chapter 6 presents a new algorithm for detecting client computers infected by bots that abuse DNS to connect to their command-and-control servers. Unlike current approaches, the algorithm can detect bots in near-realtime and, in 80% of cases, detect them before the bot contacts a command-and-control server. A version of this work appeared in:

    - Krishnan, S., Taylor, T., Monrose, F., and McHugh, J. (2013). Crossing the Threshold: Detecting Network Malfeasance via Sequential Hypothesis Testing. In *IEEE/IFIP International Conference on Dependable Systems and Networks*

# CHAPTER 2: BACKGROUND

This chapter reviews the important concepts used throughout the remainder of this dissertation. It covers the exploit kit attack model, as well as a high-level overview of current techniques for detecting such web-based attacks both before and after infection. Finally, the chapter describes domain generation algorithms and how malware uses them to compromise defenses.

## 2.1 Exploit Kit Attack Model

An exploit kit is a web-based service (website) designed to infect machines that visit the site — usually without a client's knowledge. Like any other service, an exploit kit generates money for its owner by acting as a malware delivery system. Attackers pay a monthly subscription fee with a guaranteed number of infections, and supply the malware payload to be installed on the infected machines. The malware, in turn, generates revenue for the attacker through some activity such as spamming or click jacking (Grier et al., 2012).

A web-based exploit kit attack is similar to any other scam — the attacker wants to lure an unsuspecting client into a trap to steal private information, or resources. Such attacks typically involve a four-step process as shown in Figure 2.1.

In the *bait phase*, the attacker tries to lure the client to a malicious website without their knowledge using a multitude of techniques. The most common technique is for the attacker to inject an `iframe` linking to a malicious page into one of the advertising networks dotting the Internet. To do so, the attacker can compromise an existing advertising server, or even purchase legitimate ad space. Another popular technique is phishing whereby a criminal sends a legitimate looking email with a link to the malicious site. Finally, a technique sometimes called "watering hole" sees an attacker compromise a popular website and infect unsuspecting visitors. In most baiting scenarios, there is a small time window before a defender detects and removes the compromised links, as such, the attacker must continuously be searching for new servers to compromise.

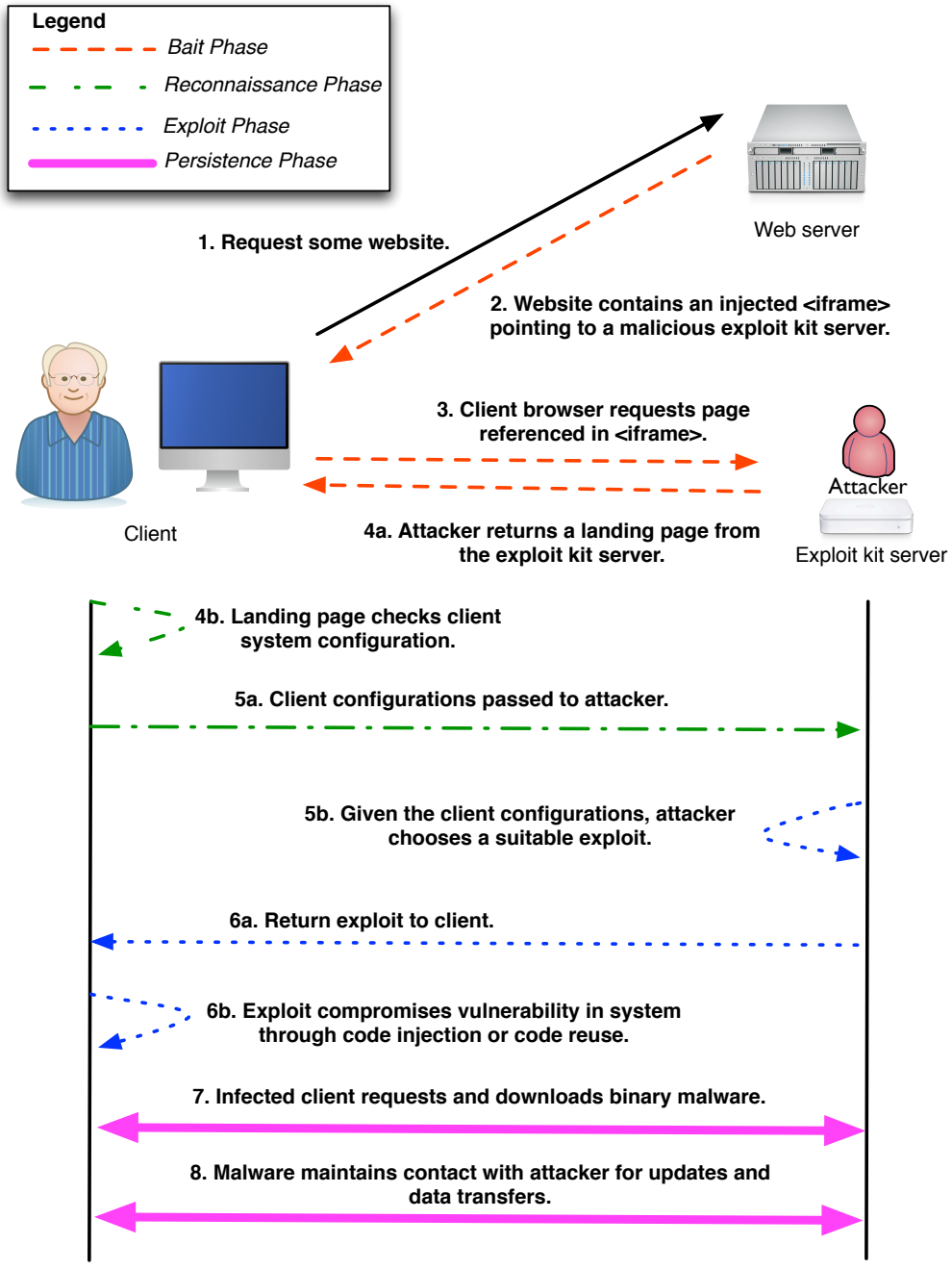



Figure 2.1: The four phases of an exploit kit infection.

Once the user is lead to a malicious landing page, the page analyzes the client machine to determine what operating system, software, and plugins are installed (i.e., web browser, Flash version, etc.) during the *reconnaissance phase*. An attacker can determine what software is installed on a client using JavaScript APIs that are already available in all commercial browsers.

With the list of installed software at hand, the exploit kit can choose from exploits that can capitalize on vulnerabilities (bugs) on the victim's machine (*exploit phase*). The attacker's goal is to leverage an exploit in order to download and run unrestricted arbitrary code on the client — effectively allowing the attacker free reign on the device. Exploits result from memory bugs in web browsers and plugins such as buffer overflows wherein application code allows for data to be written past the boundary of a memory buffer (Szekeres et al., 2013). Attackers leverage these bugs by either overwriting the browser's program stack with their own code (called *code injection* (Aleph One, 1996)), or to redirect the logical program flow to instructions already in memory in order to provide alternative program logic (called return oriented programming (Buchanan et al., 2008)).

The attacker now has full control over the computer and seeks to maintain his foothold by downloading a set of malicious binaries in the *persistence phase* that will persist across system reboots. The binaries serve two purposes. They perform some task to generate revenue for the attacker including: spamming, key logging, click fraud, browser hijacking, bitcoin mining, and ransomware (Grier et al., 2012). The binaries also maintain contact with the attacker by communicating with a command-and-control server over some protocol such as peer-to-peer, DNS, or IRC (Zeidanloo and Manaf, 2009). Such communication allows an attacker to upgrade a host and rent infected clients to customers. The collective combination of infected client machines and command-and-control servers is called a *botnet* with individual infected machines called *bots*. The following section investigates the prevalent techniques used by network security analysts to detect that a network client has interacted with an exploit kit.

## 2.2 Detecting an Exploit Kit

Security analysts have two vantage points at which they can defend their networks — directly on the host or at network boundaries. Security analysts will use a combination of both vantage points to protect their networks. This dissertation focuses on network-level defenses that result from monitoring traffic at the edge of a network boundary because security analysts cannot trust the devices on the network that they do

not control. As such, users or attackers alike can add new devices to the network that do not have the proper client-side defenses or disable defenses on existing machines. While the network vantage does not give the analyst all the rich information available at the client, it does provide a global view and historical record of all host communication enabling detection of malicious behaviors as they occur. Finding malicious behaviors, such as a client's interaction with an exploit kit, becomes a data mining problem of finding malicious patterns in a sea of data. The following subsection investigates the characteristics of an ideal network-based malware detector, and discusses at a high-level how current techniques fall short.

### 2.2.1 Characteristics of an Ideal Network-based Malfeasance Detector

There are four characteristics that are highly sought after in the design and implementation of network-based malware detectors. The first characteristic is that the detector can identify malicious network behavior as it is occurring in order to be able to stop or mitigate an infection before it can do more damage — called the timeliness requirement. The detector should also choose characteristics (features) that are necessary and sufficient to declare something as malicious. These types of features make it harder for the attacker to avoid detection. Next, we want a detector that detects everything that is malicious and nothing that is benign, and, finally, the detectors should be able to adapt to a changing adversary so that it can discover previously unseen attacks. Unfortunately, current detectors face tradeoffs sacrificing certain characteristics in order to achieve others.

The majority of network-based detector research focuses on satisfying the real-time characteristic. With tens to hundreds of thousands of data points to be analyzed per minute, real-time performance is of the utmost importance for any detector to be useful in an operational environment; however, maintaining detection speed has serious implications for detection accuracy — i.e., no data point within the network traffic is examined indepthly. This creates two problems. In order to meet the real-time requirement researchers will extract "cheap" features from the traffic. These features are extracted quickly, but they are not necessary nor sufficient for detecting malicious content. For example, Ma et al. (2011) found that URLs for malicious websites have differing characteristics (e.g., more characters, embedded IP addresses) than URLs to benign websites, Provos et al. (2007); Cova et al. (2010) found that malicious sites were more likely to use obfuscation to prevent detection, while Cova et al. (2010) and Mekky et al. (2014) note that malicious websites often utilize a number of redirections; however, because malicious websites are not required to have longer URL's or use obfuscation, and because some benign websites have these same characteristics, these approaches tend

to have high false negative and false positive rates (Taylor et al., 2016b,a). Similarly, Provos et al. (2008) reported a 10% false negative rate and Canali et al. (2011) reported a false positive rate of between 5% and 25% when classifying websites as malicious or benign, while Provos et al. (2007) only disclose that using obfuscated JavaScript as an indicator leads to a high number of false positives. Chapter 4 examines the limitations of these types of approaches and investigates their poor detection performances.

Second, in order to maintain high speed analysis, researchers build fast traffic classifiers based on either content-based signatures or machine learning. These approaches implicitly make the assumption that future attacks will look similar to past attacks leaving detectors unable to identify so-called "zero-day" attacks that are completely new. As a result, these approaches are unable to adapt to a changing adversary as will be described further in Chapter 5.

One could improve false negative and false positive rates by running potential exploits associated with websites in a virtual environment and extracting behavioral features based on changes to the web browser and operating system (Provos et al., 2008). These approaches have been successful in identifying zero-day attacks (Taylor et al., 2016a); however, virtualization is slow making it difficult to process the deluge of traffic using such a technique. Consequently, achieving the timeliness requirement of the ideal detector has been a research topic of great importance. Chapter 5 presents a new technique that adapts virtualization to the network enabling more thorough analysis while examining a fraction of the overall data.

### 2.2.2 Operational Challenges in Network Defense

As described in Sommer and Paxson (2010), there are four major operational research challenges in network defense. These operational challenges were described by Sommer and Paxson (2010) in the context of anomaly detection, but can be extended to network defenses in general. First, there is a high cost of errors. Errors, in this context, describe misclassifications of network traffic. When a network detector incorrectly classifies some traffic as malicious, when the traffic is actually benign, (i.e., a false positive) a security analyst must waste time examining the incident. As the number of false positives increase, the detector becomes unusable (Axelsson, 1999). On the other hand, classifying malicious traffic as benign can have catastrophic implications for an organization enabling an attacker free access to private corporate data.

Another operational challenge is transferring results from the network detector to actionable items for the security analyst (Sommer and Paxson, 2010). Academic research focuses on a system's capability to identify

potentially malicious behaviors, but tell us little about why the behavior is malicious. In order to assist the analyst better, we must investigate ways to provide the necessary context to make analysis faster.

Network traffic is extremely diversified. There are hundreds of network protocols, each with their own format, and some of those protocols such as HTTP and DNS have an infinite number of message content possibilities, and high variability in bandwidth, burstiness, and connection duration (Sommer and Paxson, 2010) over short intervals. Further, in the case of HTTP and DNS, if one goes looking for a particular behavior in the traffic, one can often find it. This makes defining "normal" traffic patterns incredibly difficult. As a result, researchers must use large datasets over time in an effort to measure the impact of an approaches effectiveness.

The latter challenge makes it particularly difficult to conduct sound evaluations on network detectors. For one, there are few publicly available datasets to conduct security research, and those that are available are out of date due to the constant evolution of network traffic. Even when private data sources are available, such as a traffic collector at the edge of a network, restrictions are imposed such that researchers cannot store full traffic traces to disk making dataset labeling challenging. Without large labeled datasets, it becomes almost impossible to train machine learning models that can reliably detect malfeasance on the network.

In this dissertation, I investigate the operational impact of some of the prominent network-based detection and then present new approaches to detecting exploit kits, and bots that help to deal with some of the key operation challenges discussed in this section and push the research community closer to the ideal network-based malfeasance detector. I propose techniques that significantly reduce false positives while improving or maintaining false negatives over existing approaches. Furthermore, I present techniques that are evaluated on large real-world datasets, and scale without the need for large cloud infrastructure. Each technique also detects exploits as they happen and in certain cases before any malicious activity can take place. However, before discussing these approaches, I first discuss some of the key ways in which exploit kits evade detection.

### 2.3 Exploit Kits Evasion Techniques

At each step in the exploit kit attack model, kit authors build in evasive mechanisms to go undetected by network detectors. Since these kits are effectively websites or web services hosted on a server with a full URL, they are constantly changing their locations with new domain names (called throw-away domains),

URLs, and IP addresses to avoid blacklisting. A blacklist is a list of domain names and IP addresses that are known to serve malicious content and as such, are blocked at the network-level.

During the *bait phase*, an attacker will redirect a client to a benign website if the client is from a known defender's IP address space or if the client tries to load the exploit kit multiple times. For example, Google has a large cloud infrastructure that loads thousands of websites to test if they are malicious by monitoring operating system changes (Provos et al., 2008). If a request comes from a known Google IP address, the attacker will redirect to a benign site.

In the *reconnaissance phase*, the kit checks for signs that the client is a virtualized environment, which indicates that a defender is likely testing whether the site is malicious. To do the check, a kit will capitalize on bugs in the browser or a plugin to check for the presence of registry keys, I/O ports, background process, function hooks, or IP addresses that are specific to known virtual machine software (Kirat et al., 2014). If a virtualized environment is found, the exploit kit will redirect the client to a benign site.

Often, the attacker tries to make things difficult for the defender during the *exploit phase* by spreading the components of the exploit across a number web resources (i.e., Flash file, JavaScript, HTML file, images, etc.). For example, a Flash exploit will only attempt to exploit a system if the proper parameters are passed in by the loading website. This prevents a defender from running the Flash file in a virtualized environment in isolation and observing any malicious behavior. Chapter 5 describes a new approach for network-based detection of such web-based exploits by providing context to exploitable files. The chapter also presents case studies with real life exploit kit examples.

Upon successfully infecting a host, the installed malware (bot) contacts its command-and-control server to receive instructions and updates (*persistence phase*). The bot needs some dynamic process to accomplish this goal without getting blacklisted by the defender. There are several dynamic approaches available to the bot; however, this dissertation focuses on bots that abuse the domain name system (DNS) to contact a command-and-control using a domain-name generation algorithms (DGAs) to bypass defenses which are described in more detail below.

### 2.3.1 DNS and DGAs

The domain name system is one of the backbones of the Internet allowing users to surf to their favorite websites by simply remembering descriptive domain names (e.g., www.facebook.com, www.cnn.com). More specifically, the domain name system is a hierarchical distributed database that maps domain names to their

corresponding IP addresses (a process called a name resolution). A domain name consists of one or more parts, called labels, that are concatenated and delimited by dots (e.g., www.facebook.com). The rightmost label (e.g., com) represents the top-level domain, and the hierarchy of domains descends from right to left. Labels on the left represent subdomains of the domain to the right (e.g., www.facebook.com is a subdomain of facebook.com, which is a subdomain of com).

There are two main types of servers in the domain name system. An authoritative nameserver is a server that stores the actual domain name-to-IP address mappings while a recursive resolver is a DNS server that queries an authoritative nameserver to resolve a domain address. Domains are delegated to authoritative servers hierarchically. At the top of the hierarchy is the root server, and below it is the authoritative servers for each top-level domain, and below them are the authoritative servers for the sub domains, etc.

Figure 2.2 shows the process for resolving a DNS request for www.facebook.com. In the Figure, the client queries a recursive resolver for the IP address associated with www.facebook.com. In turn, the resolver will query a root server to find the location of the "com" server. Next, it will query the "com" server for the address of the "facebook.com" server and so on until the IP address associated with www.facebook.com is found and returned to the client. To speed up future queries, the DNS resolver will cache DNS responses for a period of time (called a time to live or TTL) specified by the authoritative server.

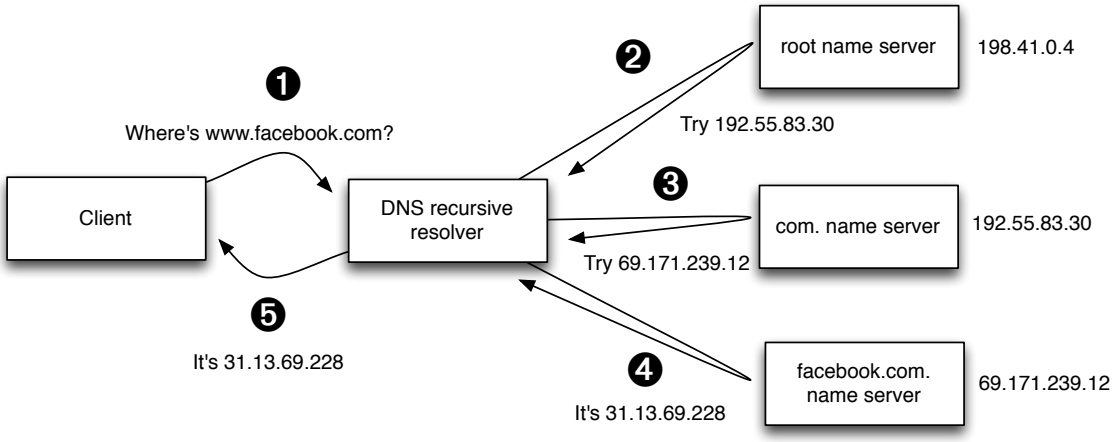



Figure 2.2: The DNS resolution process is recursive in nature. The client queries its local DNS resolver for the IP address associated with a domain name (e.g., www.facebook.com). In turn, the resolver recursively makes queries down the DNS hierarchy until it finds the requested IP address and returns it to the client.

DNS is a request-response UDP-based protocol that uses query and reply messages to pass around information. The query message contains the domain name being queried along with the type of response

requested. The most common type of response is an A record which designates that an IP version 4 address should be returned for the query domain name. Other response types include PTR (reverse lookup), which allows the client to lookup a domain name given an IP address, and MX, which contains the address of the email exchange server for the corresponding domain name. Responses also contain status codes to indicate whether a request was successfully fulfilled. If a client makes a request for a domain name that does not exist in the database, an NX response or non-existent status is returned. An in depth discussion of DNS is beyond the scope of this dissertation; however, Mockapetris and Dunlap (1988) provides an excellent overview of how the system works.

DNS was not designed with security in mind, and as a result, any misuse of this service can have a significant impact on a network's operational health. While some of the attacks attempt to exploit flaws in the resolution process (e.g., cache poisoning attacks (Kaminsky, 2008; Son and Shmatikov, 2010)), others are more subtle and leverage an enterprise's DNS infrastructure to facilitate their activities. In this dissertation, I focus on the latter problem, highlighting a growing abuse of enterprise name servers whereby infected clients use automated domain-name generation algorithms to bypass defenses.

The idea behind a DGA is that the attacker wants to discourage a defender from blacklisting his command-and-control servers by making it extremely costly for the defender to do so. A DGA is designed to generate thousands of random domain names using some global seed. The global seed could be the date and time, the 10 top trending words on twitter, or any publicly available piece of information. A bot (or malware) will then systematically query each domain name while only one of the domain names is actually registered in the DNS database as shown in Figure 2.3. The registered domain name corresponds to the command-and-control server. A bot contacts its command-and-control server everyday generating a new set of domain names each time. Examples of malware that exhibit such behavior are botnets such as `conficker` and `kraken` and web-based malware and trojans such as `RunForestRun` (Unmask Parasites, 2012). `Conficker` is a sophisticated computer worm that propagates while forming a botnet. Since its discovery in 2008, it has remained surprisingly difficult to counter because of its combined use of advance malware techniques. To date, it has infected millions of computers worldwide. The early variants would reach out to 250 pseudo-randomly generated domain per day from eight Top Level Domains (TLDs) in an attempt to update itself with new code or instructions.

A domain name generated by a DGA is called an algorithmically generated domain name (AGD) and has been used in differing contexts in the existing literature. Antonakakis et al. (2012), for example,

describe an AGD as an "automatically generated pseudo-random domain name" created by a botnet using a domain generation algorithm (DGA), whereas other authors (Bilge et al., 2011; Yadav et al., 2010; Born and Gustafson, 2010; Stone-Gross et al., 2009) refer to the process of generating domains as "domain fluxing." In this dissertation, an *algorithmically generated domain* is described as a domain that is generated by an automated process with the key objective of minimizing collisions within the DNS namespace. Consequently, algorithmically generated domains tend to be relatively long pseudo-random strings derived from some global seed. Google Chrome's domain generator, for example, creates three alpha-character strings (each of length ten) upon startup, and these strings are used to test whether the configured DNS server hijacks non-existent (NX) responses. If so, Chrome does not perform prefetching (ISC, 2011) of search terms that are entered into its location bar.

Even if the defender somehow gains access to the algorithm and the global key, she must register all possible domain names daily in order to be sure she has blocked the botnet. In Chapter 6, I investigate current approaches for detecting DGA related traffic and show that they do not satisfy the timeliness aspects of the ideal detector and can be easily evaded. I also propose a new technique for detecting bots and malware that use DGAs, which does satisfy the timeliness requirement of the ideal detector, and is much more difficult to evade than current detection techniques.

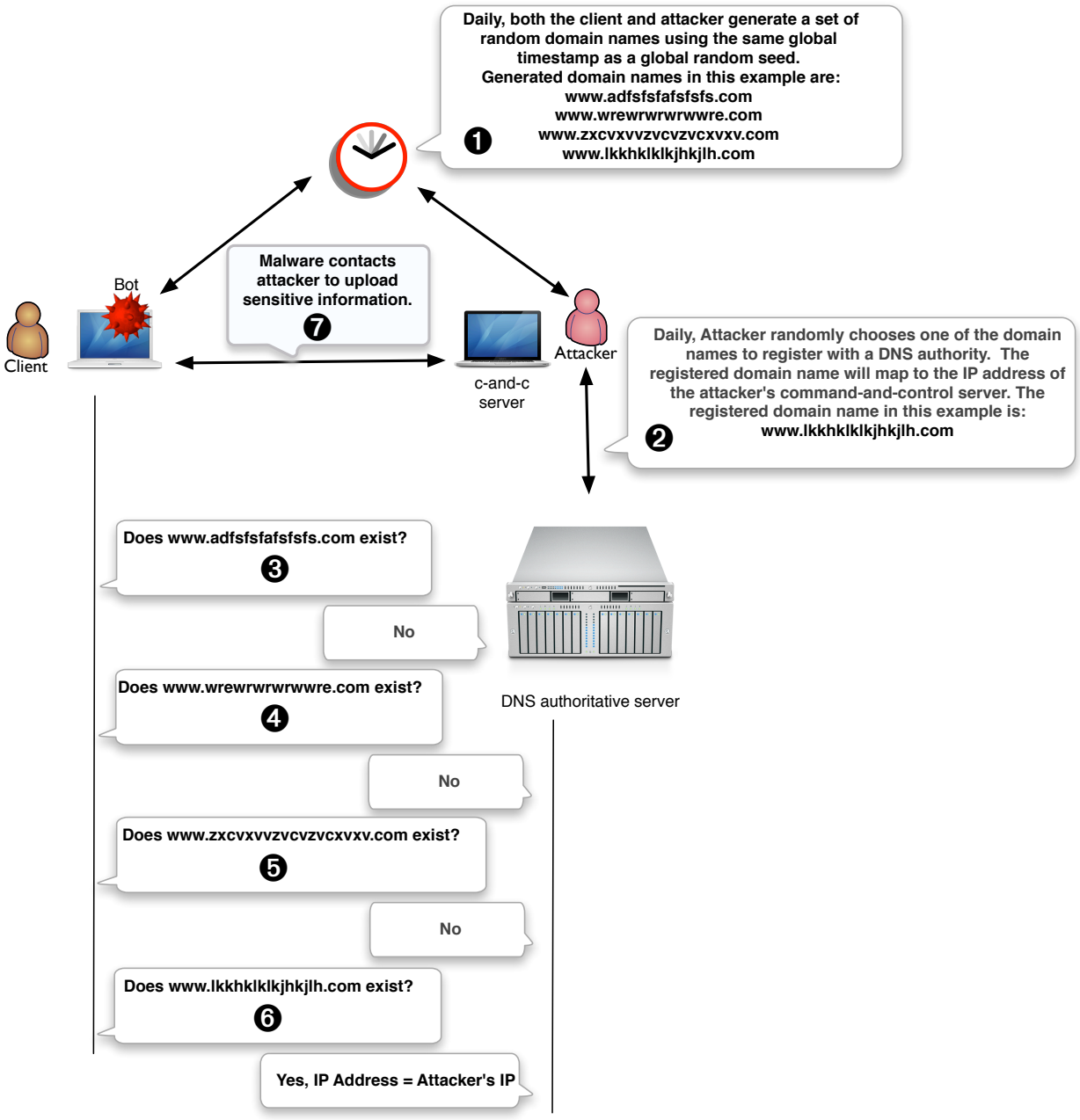



Figure 2.3: Example of how malware uses a DGA to contact the attacker's command-and-control server.

# CHAPTER 3: BACKGROUND

In Chapter 4, I investigate how the structural patterns in HTTP traffic are used to identify exploit kit related traffic while reducing false positives and false negatives over existing approaches. In this chapter, I describe a typical tree structure, and perform an indepth analysis of one of the most popular techniques for analyzing tree data — namely subtree mining. The analysis will help guide the approach chosen in Chapter 4 and inform the research community as to the limitations of subtree mining techniques on large real-life datasets.

## 3.1 Modelling HTTP Traffic as Trees

When a user surfs to a website (e.g., `http://www.cnn.com`), the web browser retrieves the site's main page from the server, parses it, and begins retrieving all the embedded web objects that comprise the page. These objects include other embedded pages, JavaScript objects, images, and videos. The embedded pages, and JavaScript objects may, in turn, load even more web objects. From a network perspective, web browsers and servers communicate in a text-based request/response protocol called HTTP whereby each loaded web object corresponds to a new HTTP web request and response by the browser to/from the server(s). Each HTTP request contains the URL (or location) of the file requested, and a list of headers that confer information about the web client to the web server. The most important request header for this dissertation is the `Referer` header which contains the URL of the web object that requested the current web object.

The web server responds with an HTTP response that contains the content requested as well as a set of headers describing the content (e.g., content type, content length, etc.), and status code indicating whether the request was successfully fulfilled. In some cases, the server may redirect the web browser to another server responding with a redirection status code, and a `Location` header with the new location of the web object.

The astute reader will note that the relationships between the HTTP request/responses of web resources for a single website form a tree-like structure, where each HTTP request/response pair becomes a node in the tree and the node x loads node y relationship between two sets of pairs represents the branches of

the tree. Figure 3.1 shows an example of this tree-like structure for a website that has a root page called `index.html`. In this dissertation, such trees are called web session trees.

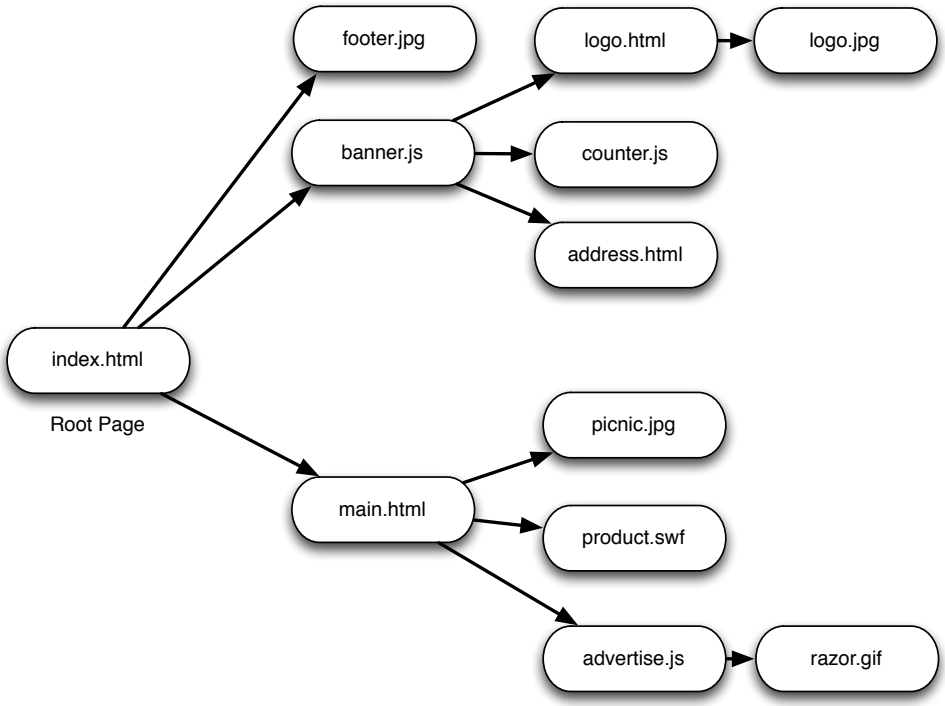



Figure 3.1: A root page (index.html) can load several different web resources (e.g., JavaScript, HTML, JPG), which, in turn, can load other web resources. This "loads" relationship forms a tree-like structure.

### 3.2 Subtree Mining: A Comparison of Algorithms on Real World Datasets

With the notion of a web session tree in place, we now need a way to extract the structural properties of the tree for analysis. There are a few options including tree edit distance (discussed in Chapter 4), but by far the most popular approach in the literature is that of subtree mining. Subtree mining is the practice of breaking large, complicated structures into more manageable substructures (i.e., subtrees) and to study their patterns. Among all substructures, frequent substructures, i.e., those occurring sufficiently often in the database, are of particular importance, as they open the door for advanced analyses, such as search (Cohen, 2013), indexing (Zhao et al., 2007), and classification (Nguyen and Shimazu, 2011). Subtree mining has been applied in areas such as phylogenetic analysis in biology (Deepak et al., 2013), text mining (Subercaze et al., 2015), natural language processing (Nguyen and Shimazu, 2011), malware detection (Narouei et al., 2015), and robot task recognition (Gemignani et al., 2015). Although subtree mining algorithm research is

relatively old, semi-structured datasets that represent trees and graphs are ubiquitous and new algorithms based on graph and subtree mining are still being proposed (Bui et al., 2014; Hadzic et al., 2015; Narouei et al., 2015). There are even approaches that reduce graph mining problems into subtree mining problems in order to reduce the runtime complexity of mining (Gemignani et al., 2015). Subtree mining research is still relevant because there are relatively few better alternatives to encode structure for solving problems on semi-structured data. Indeed, mining frequent substructures represents non-trivial challenges. The process often requires scanning the entire database over multiple iterations, which can be prohibitively expensive for large-scale settings. To address numerous practical challenges and limitations, several approaches have been recently proposed (Zaki, 2005; Jiménez et al., 2012; Zou et al., 2006b; Kutty et al., 2007; Xiao et al., 2003; Tatikonda et al., 2006; Chehreghani et al., 2011; Asai et al., 2002; Chi et al., 2003; Hido and Kawano, 2005; Wang et al., 2004) (see (da Jiménez et al., 2010) for an excellent survey); however, to date, most evaluations of subtree mining algorithms are on synthetic or small scale real datasets leaving one to wonder how they perform on a variety of real-world datasets. My interest in subtree mining is motivated by the problem of discovering malicious subtree patterns in network traffic, but existing literature does not provide insight as to whether subtree mining represents a viable solution for a real-world networking dataset.

In what follows, I examine a recent line of inquiry on the problem of mining frequent subtrees in a database of rooted and labeled trees. Existing methods can be broadly classified into two categories: *candidate generation* and *pattern growth*. A candidate generation algorithm enumerates all possible subtree combinations and incrementally calculates the frequency count for each subtree using an indexing structure that stores the occurrences of frequent nodes in the database. A pattern growth algorithm follows the divide-and-conquer methodology and generates candidate subtrees by growing subtrees from the data itself.

Despite the substantial body of work, there is still a significant lack of understanding of the strengths and limitations of these algorithms in realistic settings, resulting in a set of widely held, yet questionable, conclusions. For example, it is believed that pattern growth techniques are superior (Zou et al., 2006b; Kutty et al., 2007; Wang et al., 2004; Deepak et al., 2013); however, little evidence suggests a measurable advantage. Second, while the performance of subtree mining algorithms is influenced profoundly by multiple factors (e.g., tree size, degree, depth, label distribution), due to limitations of evaluation datasets, little is known about the intricate interplay between these factors.

Motivated by this, I conduct the first large-scale comparative study on frequent subtree mining algorithms using a variety of synthetic and real datasets. The goal is to assess the performance of existing algorithms in

realistic settings and ultimately inform better algorithm design by investigating their strengths and limitations. This chapter begins by studying the characteristics of synthetic datasets (Zaki, 2005) used in the majority of studies and demonstrate their shortcomings when compared to real datasets. The work then proposes novel synthetic tree generators that provide great flexibility in setting multiple factors (e.g., tree size, depth, fanout, label distribution) and produce trees closely mimicking the characteristics of real datasets. Leveraging the generated synthetic datasets and seven large real datasets, I investigate the runtime performance of four representative subtree mining algorithms from the two main categories (candidate generation and pattern growth) under varying setting of profounders. The algorithms were chosen because they contain the core concepts of the two categories, are popular in the literature, and represent the current state of the art. I provide insights into the strengths and weaknesses of these algorithms, many of which challenge conventionally held beliefs.

Besides regular frequent subtree mining, the chapter also considers *closed* frequent subtree mining. A frequent subtree is closed if none of its supertrees have the same support. The concept of closed subtree is attractive because special pruning techniques can be applied to speed up the mining performance and reduce the number of subtrees generated. The performance impact gained by leveraging closeness is measured.

The remainder of the Chapter begins with the four subtree mining algorithms compared in this study. Section 3.4 details the methodology, while sections 3.5 and 3.6 describe the real-world and synthetic datasets utilized. Experimental results are provided in section 6.6 before discussion and lessons learned.

## 3.3 Subtree Mining Algorithms

For pedagogical purposes, I first introduce fundamental concepts and notations used throughout the paper, then formalize the problem of frequent subtree mining. The four representative algorithms for mining subtrees compared in this work are also described.

### 3.3.1 Background

A *labeled tree* is a connected acyclic graph defined as $T = \{V, E, \Sigma, L\}$ where $V$ is the set of tree nodes; $E \subseteq V \times V$ is the set of edges; $\Sigma$ is the alphabet for node labels; and $L : V \rightarrow \Sigma \bigcup \varepsilon$ is the labeling function mapping each node to its label ($\varepsilon$ denotes empty label). If all edges of $T$ are directed and there exists a special

root node $v_0$ which does not have incoming edges, then $T$ is called a *rooted tree*. If there is an ordering relationship defined between sibling nodes in a tree, then $T$ is called an *ordered tree*.

Depending on whether a binary ordering relationship $\leq$ is defined over the tree nodes, rooted trees can be classified into: *ordered trees* where $\leq$ is defined for every pair of siblings for every tree node; *unordered trees* when there is no predefined order between sibling nodes; and *partially-ordered trees* where $\leq$ is defined only on some sets of siblings. Most subtree mining algorithms, including those compared in this dissertation, are applicable to *ordered label trees*.

All the tree mining algorithms in this study store trees in a preorder canonical string format. The pre-order string is built by adding the label of each tree node in the order of pre-order tree traversal. A special symbol $\uparrow$ is used when the traversal backtracks from child to parent. For instance, the pre-order string of tree in Figure 3.2 (a) is $ABE \uparrow F \uparrow G \uparrow\uparrow C \uparrow DH \uparrow I$. The same scheme can be adapted to the representation of unordered or partially ordered tree — see Chi et al. (2005) for more details.

A subtree is a subset of nodes and edges extracted from a larger tree. The three main types of subtrees are induced, bottom-up, and embedded.

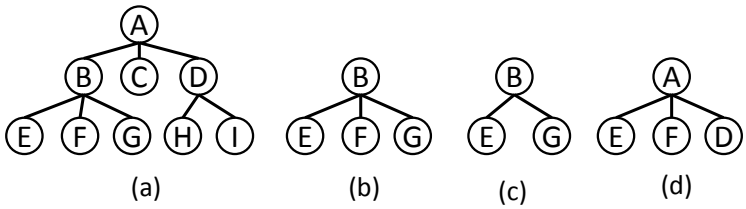



Figure 3.2: Types of subtrees. (a) original tree, (b) bottom-up subtree, (c) induced subtree, (d) embedded subtree

They are defined as follows:

- **Bottom-up subtrees**: For a rooted tree $T$ with vertex set $V$ and edge set $E$, a subtree $T'$ is a bottom-up subtree if and only if (1) $V' \subseteq V$; (2) $E' \subseteq E$; (3) all the labels of $V'$ and $E'$ are preserved; (4) if $T$ is ordered or partially ordered, any ordering among the siblings in $T$ must be preserved in $T'$; (5) for any $v \in V$, if $v \in V'$, then all the descendants of $v$ in $T$ must be preserved in $T'$. A bottom-up subtree $T'$ is obtained by taking one vertex $v$ from $T$ and all its descendant nodes and associated edges. Figure 3.2 (b) shows an example of bottom-up subtree.

- **Induced subtrees**: An induced subtree $T'$ can be defined as a bottom-up tree without the last constraint. For any vertex $v \in V'$, it can contains only a subset of all its descendant nodes in $V$; however, if $v_1$ is the parent of $v_2$ in $T$ and both of them are present in $T'$, then $v_1$ must also be the parent of $v_2$ in $T'$. Intuitively, an induced subtree $T'$ can be obtained by repeatedly removing leaf nodes from a bottom-up subtree of $T$. Figure 3.2 (c) shows an example of induced subtree.

- **Embedded subtrees**: An embedded subtree further relaxes the constraints of induced subtrees by allowing breaking parent-child relationships; however, the ancestor-descendant relationships among vertices of $T$ must be preserved. A $T'$ is an embedded subtree of $T$ if and only if: (1) $V' \subseteq V$; (2) the labels of all the nodes of $V'$ in $T$ is preserved in $T'$; (3) if $v_1$ is the parent of $v_2$ in $T'$, then $v_1$ must be an ancestor of $v2$ in $T$; (4) for all $v_1$ and $v_2$ in $V'$, preorder($v_1$) $<$ preorder($v_2$) in $T'$ if and only if preorder($v_1$) $<$ preorder($v_2$) in $T$. Figure 3.2 (d) shows an example of induced subtree. Notice that vertices $E$ and $F$ are not direct children of $A$ in the original tree.

Most frequent subtree mining algorithms attempt to discover frequent induced subtrees in a database of ordered, labeled trees, due to their wide-range applications. Therefore in the study I focus on the following problem:

*Given a database of ordered, labeled trees $D$, find all induced subtrees appearing in at least $\Delta$ trees in $D$, where $\Delta$ is a user-specified threshold (called the minimum support). The support threshold is typically presented as a fraction of the overall number of trees in the dataset.*

Existing subtree mining algorithms can be classified into two categories: candidate generation and pattern growth (da Jiménez et al., 2010), both built upon the *a-priori principle*: all subtrees of a frequent subtree must also be frequent. Thus both candidate generation and pattern growth form candidate subtrees using frequent nodes as basic building blocks, though different in their particular ways of constructing candidates. A plethora of algorithms based on these basic concepts have been proposed (da Jiménez et al., 2010), which provide a variety of enhancements to improve performance. To date a broad comparison of approaches has not been done.

Candidate generation has attracted more intensive research in this space (da Jiménez et al., 2010). It involves generating a set of subtrees based on frequent nodes and testing them against the datastore. Trees are

stored in a breadth-first or depth-first canonical form (Chi et al., 2005) and a special in-memory structure is built to map frequent nodes to their positions in trees in the database. New candidates are built by continuously adding frequent nodes on either the right-most or left-most path of current ones.

By comparison, pattern growth has been highly touted in the literature as much faster than candidate generation, but has received little attention by the majority of subtree mining studies (da Jiménez et al., 2010). Pattern growth grows subtree patterns from the data itself, by starting at frequent nodes in dataset and visiting surrounding nodes to build subtrees (Kutty et al., 2007); therefore, it only generates candidates that actually appear in the data eliminating the costly checking phase employed in traditional candidate generation algorithms. The tree dataset is partitioned by frequent nodes and each time a potential frequent subpattern is found, it is mined against all candidates in the partition.

Chi et al. (2003); Kutty et al. (2007) focus on closed and maximal subtree mining to reduce the number of subtrees generated. Closed and maximal subtrees are a subset of all subtrees. A frequent subtree is closed if none of its supertrees have the same support (count). It is maximal if none of its supertrees have a support count higher than the minimum threshold.

This chapter explores the performance benefits of both candidate generation and pattern growth-based induced subtree mining techniques as well as closed induced subtree mining techniques. The substantial body of work on subtree mining precludes the possibility of evaluating all existing algorithms; therefore, I choose four representative subtree mining algorithms. The first two algorithms are based on candidate generation techniques: FREQT (Asai et al., 2002) and CMTreeMiner (Chi et al., 2004) (mines closed subtrees); while the other two are based on pattern growth techniques: PrefixISpan (Zou et al., 2006b) and PCITMiner (Kutty et al., 2007) (mines closed subtrees). I chose PrefixISpan and PCITMiner because they are the only pattern growth algorithm designed for induced subtrees (that we know of) while others focus on either embedded (Wang et al., 2004; Zou et al., 2006a) or maximal (Paik et al., 2008) subtrees. FREQT and CMTreeMiner were chosen because they are by far the most popular candidate generation algorithms in the literature, and they still represent the state of the art in the research area. Most other methods follow paradigms similar to either FREQT or CMTreeMiner, but with limited extension and marginal performance improvement (Tan et al., 2008; Chehreghani et al., 2011; Hido and Kawano, 2005) or are built for specific datasets (Deepak et al., 2013; Termier et al., 2008). For example, Hido and Kawano (2005) extended the rightmost expansion model of FREQT to support both right and left expansion while Chehreghani et al. (2011) presents a new equivalent class extension. In understanding the strengths and

weaknesses of two major schools of subtree mining algorithms, I have no vested interest in any of these specific extensions.

The analysis that follows is by no means an easy feat, requiring (in some cases) an implementation of algorithms based off the description in there respective papers, a thorough understanding of each algorithm, and an assurance that the outputs were the same across each dataset. All implementations of the algorithms in our comparative study were written in C++, compiled with GCC 4.1.2, using optimization 3, with STL vectors, and the densehashmap[1] to ensure a fair comparison. The implementation of the algorithms and correspondence with authors (in some cases) took several months to complete.

### 3.3.2 Review of Selected Algorithms

■ **FREQT** (Asai et al., 2002) is a candidate generation subtree mining algorithm that incrementally constructs sets of subtree candidates of a particular size. As with all algorithms studied, candidates are generated by adding frequent nodes only on the rightmost branch of the tree.

In the first pass, a set of all single-node frequent candidates are mined and their occurrences (in the dataset) are marked using a data structure called an occurrence list. An occurrence list is simply an index which knows the location of all frequent subtree candidates in the dataset. In subsequent passes, the set of all candidates of size $k$ are computed by merging the candidates of size $k-1$ computed earlier. All patterns of a certain size and their occurrence lists are maintained in memory.

It is prudent to note that FREQT's definition of a frequent subtree is slightly different than the typical definition of frequency. That is, FREQT treats an entire dataset of trees as one big tree with a single root node and defines the frequency of a subtree as the number of occurrences of the subtree in the larger tree. The typical definition is to count the frequency as the number of trees in which a subtree appears. For example, if subtree $s$ appeared 5 times in a single tree, FREQT would have a frequency count of 5, while the other algorithms would count it only once. The subtle difference does not have a significant impact on our findings.

**Implementation** The JAVA-based implementation released by Asai et al. (2002)[2] was reimplemented in C++ for a more fair comparison with the other algorithms. It was modified to read canonical formatted trees (as first described in Zaki (2005)) instead of XML. The rewrite was done based off version 4 of the source code (dated March 24th, 2004). Additionally, Asai et al. (2002) describe two feature additions to

[1] `https://github.com/sparsehash/sparsehash`

[2] `http://research.nii.ac.jp/~uno/codes.htm`

the base FREQT algorithm: 1) duplicate node detection and 2) node skip. Duplicate node detection detects and minimizes the impact of duplicate nodes in the dataset while node skip prunes infrequent nodes when growing subtrees during the candidate generation phase of the algorithm. I enable the node skip feature and disable the duplicate detection feature so that there is an environment where FREQT and PREFIXISPAN are compared fairly. Next, the experiments are run with duplicate detection enabled to show how the feature improves performance.

■ **CMTREEMINER** is a candidate generation algorithm suggested by Chi et al. (2004) and similar to FREQT, except it only reports closed and maximal subtrees, and generates candidates one by one, instead of all candidates for a fixed size $k$ at once. Since it generates only closed and maximal trees, it leverages a set of pruners to reduce the number of candidates generated. The core component of the CMTreeMining (Chi et al., 2004) algorithm is called the *blanket*. The blanket $B_t$ of a tree $t$ is the set of all supertrees of $t$ that have one more node than $t$. A frequent subtree is maximal iff for every $t' \in B_t$, $support(t') < \Delta$; $t$ is closed iff for every $t' \in B_t$, $support(t') < support(t)$. $t' \in B_t$ and $t$ are *occurrence matched* if for each occurrence of $t$ in the dataset, there is a corresponding occurrence of $t'$; $t' \in B_t$ and $t$ are *support matched* if for each tree in which $t$ appears, there is a corresponding appearance of $t'$. If two trees are occurrence matched, they are also support matched.

The approach of Chi et al. (2004) first finds all frequent nodes in the datastore and generates single node candidate trees as well as their corresponding occurrence lists. For each candidate subtree, an occurrence-matched "blanket" is created for all vertices that do *not* appear on the rightmost path of the candidate. For example, Figure 3.3 shows a subtree $s$ with nodes $A - G$ with a sampling of the possible ways in which the subtree could be extended by a node $H$. Each $H$ node is a blanket of $s$ by definition. However, only node growth along the rightmost path of $s$ is allowed (see the transparent $H$ nodes). This is done to improve efficiency and reduce duplicate subtree generation. If there are any nodes $H$, that are occurrence matched with $s$ (i.e., everywhere $s$ appears $H$ also appears), and do not appear along the rightmost path (shaded nodes in Figure 3.3), $s$ is pruned. To see why, take the shaded $H$ node on the root as an example. If that node appears everywhere $s$ appears, $s$ can be pruned, because $s$ will be included in the set of subtrees that are rooted by $H$. If $s$ is occurrence matched with $H$, then it must be support matched as well; therefore, $s$ cannot be closed.

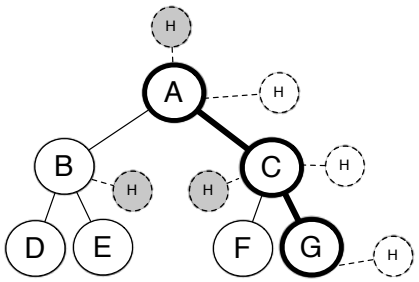



Figure 3.3: $s$ consists of nodes $A - G$. A blanket consists of adding a node $H$ either along the rightmost path of $s$ (e.g., the transparent $H$ nodes) or off the rightmost path (e.g., the shaded $H$ nodes).

If the candidate tree is not pruned, it is checked for closure by calculating a support match blanket. The same principle holds as with the occurrence matched blanket—if there is a support matched candidate in the blanket whose node does not appear on the rightmost path, then the candidate is not closed. Next, the candidate is extended by single nodes on the rightmost path and the algorithm is recursively called on the new candidates.

Each time a candidate is generated along the rightmost path, it is checked for closure and maximality. Finally, maximal trees are tested with a frequency blanket to verify they are maximal. If all trees $t'$ in the frequency blanket of $t$ have a support threshold less than $\Delta$, than $t$ is considered maximal. Note that the frequency blanket is only checked after closure is confirmed on all paths, and maximality is confirmed on the rightmost path. In doing so, the number of frequency blanket checks are substantially reduced.

**Implementation** I use the C++ implementation (dated April 4, 2008) for CMTREEMINER (Chi et al., 2003)[3]. The implementation was modified to support hashes rather than arrays to improve performance for high cardinality labels (the original implementation only supported $2^{16}$ labels).

■ **PREFIXISPAN** is a pattern growth technique suggested by Zou et al. (2006b). It generates subtree candidates by traversing the datastore instead of enumerating the space of candidate trees. It is based off the concept of a *growth element* which can be thought of as follows: Given a tree $t$ of $m$ nodes and another tree $t'$ of $(m + 1)$ nodes, $t$ is considered a prefix of $t'$ and the extra node in $t'$ is called the growth element of $t$. The growth element consists of both the node label and attaching position in $t$.

The algorithm is also built on the concept of a *projected instance*. The idea is as follows. Suppose we have a tree $t$ and a subtree candidate $s$, and some occurrences of $s$ in $t$. The postfix subtrees attached to $s$

[3] `http://www.yunchi.org/publication/software.html`

form a projected instance of $t$. A postfix subtree is simply the remaining portion of the tree following the occurrence of the subtree $s$. If we take the projected instances of all occurrences of $s$ across all the trees in $D$, we get a projection database of all potential growth elements of $s$. Given this projection database, we can grow the subtree along the rightmost path and calculate its support as it grows. This process effectively reduces the need to check a subtree with the database, resulting in improved performance. As $s$ is grown, the projection database is continually partitioned until there are no more entries at which point a new set of subtrees is mined.

■ **PCITMINER** incorporates two extensions to PREFIXISPAN in order to support closure(Kutty et al., 2007). Kutty et al. (2007) propose that the subtree space be reduced by using a backward scan based on the following observation: given two single-node frequent subtrees $s$ and $s'$ in a dataset $D$, if $s'$ is the parent of $s$ wherever $s$ appears in $D$, then the projection growth of $s$ can be halted because $s'$ will contain all subtrees using the prefix[4] of $s$. Note that the backward scanning approach is similar to an occurrence match blanket idea, but only prunes candidates with occurrence matched parents instead of all occurrence matched nodes not on the rightmost path.

Kutty et al. (2007) also propose the use of a forward and backward extension event checking mechanism. A backward extension event occurs when there is a node that is not on the rightmost path of the current subtree, but is support matched with the subtree. A backward extension is similar to a support match blanket and signals that a subtree is not closed.

A forward extension event occurs when the extra node $n$ in $s'_p$ is along the rightmost path and again $s_p$ would not be considered closed.

**Implementation** I was unable to procure an implementation of either PCITMINER or PREFIXISPAN from the original authors after repeated attempts to contact; therefore, I built a C++ implementation based off the papers[5]. I diverged from the approach of Kutty et al. (2007) in two ways. I found that the backward scanning feature alone did not provide an adequate enough performance comparison with CMTREEMINER because the feature was unable to prune the subtree space as well as an occurrence match blanket (see §3.7.3 for

[4] Kutty et al. (2007) describes backward scanning incorrectly in Lemma 1. The Lemma says that $s'$ must be a parent of $s$ in all trees of $D$. However, $s'$ and $s$ must be occurrence matched for backward scanning to work.

[5] I did not use the pseudo-projection optimization described in Zou et al. (2006b), because the details were omitted, and I was unable to devise a version that was significantly faster than the unoptimized case.

further discussion); therefore, I extended the backward scanning technique to include an occurrence matched blanket. This change improved the performance of the original algorithms on real-world datasets.

The closure check in the PCITMINER paper involves checking for closure by storing a database of "potentially" closed subtrees, upon which new subtrees are checked. I favored a simpler solution that checks for support matched growth elements and eliminates storing potentially closed subtrees.

### 3.4 Methodology

The four algorithms (CMTREEMINER, PCITMINER, FREQT, and PREFIXISPAN) were tested against a random sample of 8,000 trees from each of seven real and four synthetic datasets (described next). Eight thousand was the chosen number because the Weather and Wikipedia only contained 8,000 trees and the number was large enough to show bottlenecks during evaluation. Note that FREQT was run with both duplicate detection disabled and then enabled.

All algorithms were tested against the same samples and the minimum support threshold was varied between each run across four values: 0.05, 0.01, 0.001, and 0.0001. The thresholds were chosen because they represent a wide spectrum of workloads for the algorithms. At the lowest threshold (0.0001), every node is frequent while at the highest threshold (0.05), no dataset has more than 145 frequent nodes.

**Metrics** To study the performance of the aforementioned algorithms, I analyzed their behavior based on (*i*) their *candidate generation* phase where a set of candidate subtrees are created to be mined and (*ii*) their *dataset iteration* phase where subtrees are verified. I compute (1) the number of subtree candidates generated and checked against the database, (2) the total number of tree nodes visited, (3) the size of occurrence lists in terms of number of items, (4) wall time, and (5) the number and size (in terms of nodes) of each frequent and closed subtree.

**Machine and Operating Systems Specifications** All experiments were divided across two machines running Linux CentOS 6. The first machine had four Intel Xeon X7560 @ 2.27GHz CPUs each with eight cores; the second machine had eight Intel Xeon X7550 @ 2.00GHz CPUs each with eight cores. Each algorithm ran on a single core and was allocated 20 GBs of memory (from an available 528 GB). Experiments that did not complete after 24 hours were terminated. For consistency, the datasets were split across the two machines, ensuring that all algorithms were run on the same dataset using the same hardware.

Table 3.1: 8,000 trees sampled from real-world datasets.

| Datasets | # labels | Depth | | | Fanout | | | Size | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Min | Max | Avg | Min | Max | Avg | Min | Max | Avg |
| Webtraffic | 5,427 | 1 | 28 | 2.75 | 0 | 2,669 | 18.8 | 1 | 2,670 | 24 |
| Securities | 5,560 | 4 | 4 | 4 | 9 | 11 | 10.04 | 20 | 24 | 22.08 |
| CSLOGS | 7,408 | 2 | 86 | 4.79 | 1 | 148 | 4.63 | 2 | 428 | 14.56 |
| Weather | 34,964 | 4 | 5 | 4.49 | 10 | 40 | 21.52 | 32 | 86 | 52.95 |
| DBLP | 37,357 | 4 | 6 | 4.01 | 2 | 17 | 8.13 | 6 | 36 | 18.27 |
| Wikipedia | 67,234 | 5 | 6 | 6 | 8 | 10 | 9.34 | 26 | 33 | 30.2 |
| Uniprot | 87,489 | 6 | 7 | 6.02 | 13 | 981 | 71.92 | 49 | 4,451 | 251.22 |

### 3.5 Real World Datasets

For a comprehensive evaluation of the four algorithms, seven real-world tree datasets and four synthetic datasets were collected. Statistics for each dataset are shown in Table 3.1. These datasets were chosen because they represent a wide spectrum of characteristics including number of unique labels, depth, fanout, and size. A small subset of these datasets have been used in the literature (Cohen, 2013; Zou et al., 2006b; Kutty et al., 2007; Chehreghani et al., 2011).

Since ordered induced labeled trees are mined, each dataset is stored in an ordered depth-first canonical form using the algorithm described in Chi et al. (2005). String labels are converted to integers for convenience.

1. CSLOGS consists of web log files for the websites at Rensselaer Polytechnic Institute. The dataset has the widest variation in depth of any of the datasets tested.

2. DBLP consists of 632,000 bibliographic entries and is the largest in terms of the number of unique labels.

3. Securities consists of 17,000 money market funds from January to October 2013 and is the smallest in terms of unique labels and maximum tree depth.

4. Uniprot consists of annotated protein sequences and has the largest trees.

5. Weather contains the climate conditions for the US and has trees with consistent depth and fanout parameters.

6. Webtraffic consists of network traces of HTTP traffic from about 3,000 clients on an enterprise network. Trees are formed by grouping and building relationships between the HTTP connections. The dataset is the most diverse in terms of varying fanout and depth.

7. Wikipedia includes 8,000 static wikipedia pages. This dataset has the most similar trees in terms of depth, fanout, and size.

For the analysis that follows, label distribution is an important factor impacting the runtime performance of subtree mining algorithms; therefore, to highlight pertinent issues, I first briefly discuss key characteristics of the labels in the seven real-world datasets gathered for the experiments. Figure 3.4 shows the empirical cumulative distribution function of the number of unique labels seen in a dataset at a particular tree depth. The cardinality of the labels increases as the depth increases, with relatively few unique labels seen closer to the root, meaning that the internal portions of the trees are relatively similar to other trees in the datasets.

Node labels appear in relatively few positions within a tree. For example, in five datasets (DBLP, Securities, Weather, Uniprot, Wikipedia), approximately 99% of all labels appear at exactly one depth and have the same number of children (i.e., degree) in all locations where they appear. Over 90% of all labels appearing more than once have exactly one degree/depth combination — a characteristic that makes these datasets ideal for closed subtree mining.

The Webtraffic and CSLOGS datasets are different from the rest in that only 20% of the labels have a single depth and combination. That said, over 90% percent of the nodes appear in less than ten positions. Shortly, attention is turned to how these characteristics affect the effectiveness of the closed subtree mining algorithms.

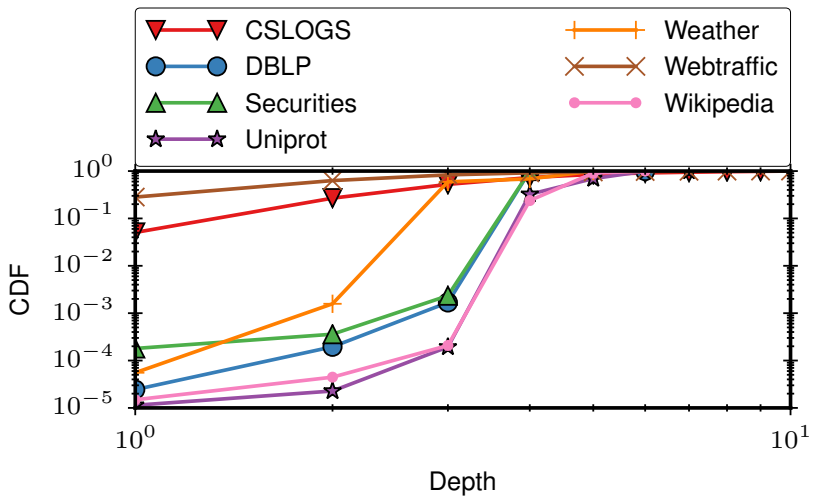



Figure 3.4: CDF of the number of unique labels per tree depth for all datasets.

Figure 3.5 shows the number of frequent nodes per support threshold for three of the datasets — Uniprot, CSLOGS, DBLP. The other datasets are omitted for clarity; however, the results are consistent. Note, that for the largest dataset (Uniprot) there are only 145 frequent nodes for the highest threshold we tested (0.05), and only 25 in DBLP. As seen later, even a small subset of frequent nodes can cause an algorithm to mine for hours or exhaust memory.

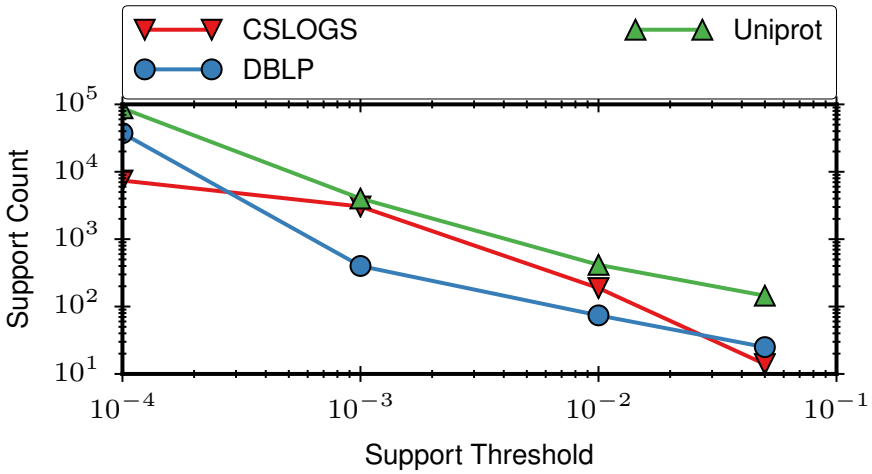



Figure 3.5: The number of frequent nodes per threshold (0.0001, 0.001, 0.01, 0.05).

## 3.6 Synthetic Datasets

The fact that one has no control over the attributes of realistic datasets means that one cannot solely rely on such datasets to gain a better understanding of the characteristics that affect the algorithms under study. To gain the desired level of control, I explore several synthetic datasets.

### 3.6.1 Synthetic Tree Generator in Literature

One of the most popular synthetic tree generators (used in no less than eleven subtree mining studies (Zaki, 2005; Jiménez et al., 2012; Zou et al., 2006b; Kutty et al., 2007; Xiao et al., 2003; Tatikonda et al., 2006; Chehreghani et al., 2011; Asai et al., 2002; Chi et al., 2003; Hido and Kawano, 2005; Wang et al., 2004)) was examined. That generator is the one originally proposed by Zaki (2005), which takes a set of parameters (e.g., tree depth (D), fanout (F), number of unique labels (L), total number of nodes (M), and number of subtrees (T)) as input. Based on the input parameters, the generator builds a single master tree with M nodes, and then creates smaller trees by traversing the nodes of the master tree. The generator can be set to always start from the root of the master tree, or to start from a randomly selected node when producing subtrees. Past work

Table 3.2: Common configs used in Zaki's tree generator (Zaki, 2005).

| **Parameter** | **F5** | **D10** | **T1M** |
|---|---|---|---|
| Master Tree Size | 10,000 | 10,000 | 10,000 |
| No. of Unique Labels | 100 | 100 | 100 |
| No. of Trees | 100,000 | 100,000 | 1,000,000 |
| Fanout | 5 | 10 | 10 |
| Depth | 10 | 10 | 10 |
| No. of Unique Trees | 2,042 | 3,099 | 3,046 |

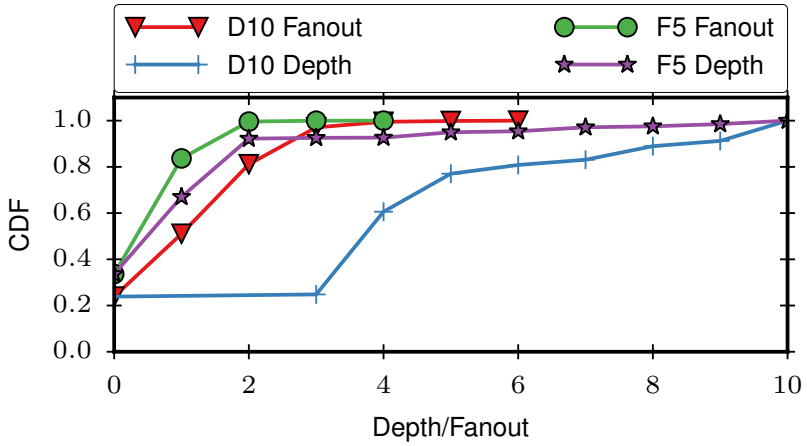
has used Zaki's generator to create three common datasets for evaluating subtree mining algorithms. These prototypical configurations (called F5, D10, and T1M) are listed in Table 3.2.

The goal was to generate tree datasets with variations in fanout, depth, and label distribution using Zaki's approach; however, upon close inspection it was observed that the trees generated in these datasets were noticeably different from what the input parameters suggested. For example, Figure 3.6a shows the cumulative distribution functions of the max depth and fanout of the trees generated in datasets F5, D10, and T1M. Notice that 98% of the trees are of fanout two or less, while over 75% of the trees are of depth of two or less. Furthermore, 30–50% of the trees contain only a single node. With such small tree sizes and a limited number of node labels, the generator only created approximately 3,000 unique structures (Table 3.2).

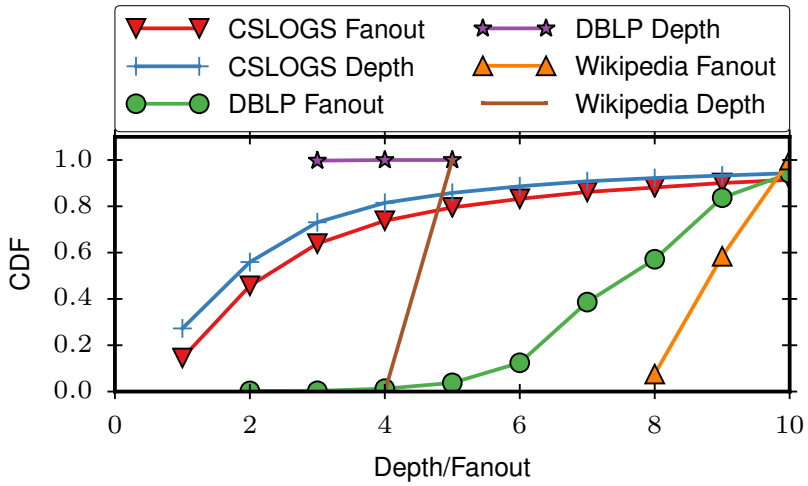
The problem is that Zaki's generator was designed to create synthetic datasets to model the tree-like structures inherent in client webpage accesses; yet it is being used as a generic tree generator to test the performance of tree mining algorithms under a wide variety of application domains. That is unfortunate, since many real-world datasets (see Figure 3.6b) have characteristics that are drastically different from webpage access patterns in terms of fanout and depth. More troubling is the fact that the trees output by Zaki's generator are in stark contrast to that for real world websites, and so analysis of mining algorithms using only this data can easily lead to incorrect conclusion. To that point, note that the CSLOGS dataset has no single node trees and more than 91% of its trees are unique.

### 3.6.2 Custom Synthetic Tree Generators

For the experiments, there were two requirements for the synthetic generator: (1) providing fine-grain control over tree characteristics such as fanout, depth, tree size, number of unique labels, label distributions (both within a tree and across a dataset), and number of trees, (2) modeling different real-world datasets (not just a single real-world dataset as in the case of Zaki's generator). Such a generator would maintain the structure of trees in real-world datasets, but also enable the alteration of characteristics such as labeling,




(a) CDFs of the max depth/fanout of the datasets generated by Zaki's tree generator (Zaki, 2005).




(b) CDFs of the max depth/fanout for real-world datasets.

Figure 3.6: Maximum fanout/depth characteristics of real and synthetic datasets.

number of trees, or create slight perturbations in depth, tree size and fanout. As a result, three tree generators were built.

■ **K-ary Generator**: The $k$-ary generator builds perfect $k$-ary trees. A *perfect k-ary* tree is a tree in which each node contains either 0 or $k$ children and where all leaf nodes are at the same level. The $k$-ary generator takes fanout, depth, and a label distribution as input, and outputs a user-specified number of trees using the label distribution. Labels are assigned to each node based on a transitional probability. The root node label is assigned using a label distribution of the root nodes from a real dataset, and then each child node label is assigned using a conditional probability based on its parent node label.

More formally, the transitional probability model $P_L$ for the label set is as follows: define an alphabet of labels $L = l_1, ..., l_n$ and a set of labels at a depth $d$ $L_d = l_{d1}, ..., l_{dm}$ for a dataset $D$, for $0 < d < q$, where $q$ is the maximum depth found in $D$. $|l_{di}|$ is defined as the number of occurrences of the label $l_i$ in $D$ at depth $d$, and $|L_d|$ as the total number of labels seen in the dataset at depth $d$. When building a $k$-ary tree, the probability of selecting $l_i$ as the root label is $P(l_i) = P(l_i|d = 0) = |l_{di}|/|L_d|$. The probability of selecting a label at any other node in the tree is conditional on its parent label — i.e., $P(l_i) = P(l_{di}|l_{(d-1)p})$ where $l_{(d-1)p}$ is the parent label.

It is prudent to note that $k$-ary trees are not found in real-world datasets (at least in those studied here); however, they are useful for creating datasets with consistent depth, tree size, and fanout parameters. These controlled settings make it easier to analyze the strengths and weaknesses of the subtree mining algorithms. To aid in the goal, this generator created the $\mathcal{S}_{\text{fanout}}$ and $\mathcal{S}_{\text{fanout:depth}}$ datasets.

**Varying fanout ($\mathcal{S}_{\text{fanout}}$)** Fanout was measured by generating 8 different $k$-ary datasets of depth two and varying $k$ = 2, 3, 5, 10, 15, 20, 25, 30 using the Securities dataset for labeling. Note that in the real-world datasets, the number of unique labels tends to increase as tree depth increases with the leaves of the tree having the highest number of unique labels. As a result, labels from depth three and four of the Securities dataset were used to model the labelling of depth one and two of these generated datasets. The characteristics are summarized in Table 3.3.

**Varying fanout and depth ($\mathcal{S}_{\text{fanout:depth}}$)** To measure the impact of fanout and depth, 12 $k$-ary datasets were generated that varied in depth from $2, \ldots, 4$ with $k = 2, \ldots, 5$.

■ **Trace-based Generator**: The trace-based generator builds synthetic datasets designed to closely mimic real datasets by utilizing probability distributions for fanout, depth, and labels. The generator takes a fanout, depth, transitional label distribution, and tree size as input. Once the overall tree depth is determined, the total fanout for each level of the tree is calculated based on depth and tree size, while individual node fanouts are determined by a conditional probability based on the node's depth. This maintains the structure and size of a real dataset, all while exploring the full spectrum of the parameter space by fixing certain parameters and altering others. The trace-based generator created the $\mathcal{S}_{\text{trees}}$ dataset.

**Varying number of trees ($\mathcal{S}_{\text{trees}}$)** To measure the impact of the number of trees, the DBLP labeling, fanout and depth distributions were applied and trees of depth 4, fanout 10, and size of 20 were created. Four datasets were generated with 8,000, 16,000, 32,000 and 64,000 trees.

Table 3.3: Characteristics of synthetic datasets.

| | Vary fanout ($\mathcal{S}_{\text{fanout}}$) | Vary fanout Vary depth ($\mathcal{S}_{\text{fanout:depth}}$) | Vary labels ($\mathcal{S}_{\text{labels}}$) | Vary # of trees ($\mathcal{S}_{\text{trees}}$) |
|---|---|---|---|---|
| **Depth** | 2 | $d = 2, \ldots, 4$ | $avg$=4 $max$=5 | 4 |
| **Fanout** | $k$=2,3,5,10, 15,20,25,30 | $k = 2, \ldots, 5$ | $avg$=10 $max$=28 | 10 |
| **Tree Size** | $3 - 31$ | $3 - 781$ | $avg$=22 $max$=58 | 20 |
| **Label** | Securities | DBLP | DBLP | DBLP |
| **Generator** | $k$-ary | $k$-ary | label | trace |
| **# Trees** | 8K | 8K | 8K | 8,16,32,64K |
| **Description** | Eight datasets; each with fanout $k$ | Twelve datasets; each with depth $d$ and fanout $k$ combination. | Seven datasets; each with a no. of distinct labels — 100,1K,2K,3K,4K,5K,10K. | Four datasets; each with a different no. of trees. |

■ **Label Generator**: The label generator artificially reduces the number of distinct labels in a dataset to a user-specified level. Such a task is challenging, as one must artificially change the number of distinct labels in a dataset but still maintain the relative frequencies of the labels as well as their relative positions in the tree. My approach takes a real dataset of trees $D$, and a desired number of labels $m$, and outputs $n$ trees with $m$ labels and a label distribution of $D$. The generator first reads enough trees from $D$ to build a transitional probabilistic label model $P$ with $m$ labels, and then relabels $n$ trees from $D$ using $P$. Details are shown in Algorithm 1. The label generator created the $\mathcal{S}_{\text{labels}}$ dataset.

**Varying number of unique labels ($\mathcal{S}_{\text{labels}}$)** In order to measure the impact of label cardinality on the algorithms, the DBLP dataset as the basis for the artificially created datasets. We then generated datasets that contained 100, 1,000, 2,000, 3,000, 5,000, and 10,000 labels.

### 3.7 Evaluation

Now it is time to focus on the runtime performance of the conventional and closed subtree mining algorithms as tested on the real and synthetic datasets.

#### 3.7.1 Output Verification

Although the primary goal of this study is to compare the runtime performance of subtree mining algorithms, to make fair comparison, it is necessary to ensure that these algorithms provide similar results.

**Algorithm 1** Pseudo code for the label generator (§3.6.2)

**Require:** $D$: dataset of trees. $n$: desired number of trees. $m$: desired number of unique labels. $\mathcal{L}$: total label set for D. $\mathcal{L}_t$: label set of tree $t$.

**Ensure:** A tree dataset $D_n$ of N trees, containing $m$ labels, and having the label distribution of D. Note: $|D| > |n|$, $|m| < |\mathcal{L}|$

$\mathcal{L}_m = \emptyset$;

$D_n = \emptyset$;

$i = 0$;

**while** $|\mathcal{L}_m| < m$ **do**
    $t = D[i]$;

    $\mathcal{L}_m = \mathcal{L}_m \cup \mathcal{L}_t$;

    calculate transitional probability model $P$;

    $i++$;

**end while**

$j = 0$;

$i = 0$;

**while** $j < n$ **do**
    $t = D[i]$;

    **for** $l$ in preorder traversal of $t$ **do**
        **if** $l$ not in $\mathcal{L}_m$ **then**
            $l = P(parent(l))$;

        **end if**

    **end for**

    $D_n = D_n \cup t_i$;

    $j++$;

    $i++$;

**end while**

Across all the real datasets, the outputs of FREQT and CMTREEMINER were contrasted against that of PREFIXISPAN and PCITMINER in terms of size and count statistics of (closed) frequent subtrees. While both Asai et al. (2002) and Chi et al. (2004) have theoretically proved the correctness of their algorithms, there are small discrepancies in the outputs of FREQT and PREFIXISPAN over certain datasets. The discrepancies are within 100 subtrees and diminish as the support threshold increases. For example, at threshold 0.01, FREQT generates 2,550 subtrees on the CSLOGS dataset while PREFIXISPAN generates 2,531; on the Securities dataset FREQT generates 247,382 subtrees while PREFIXISPAN generates 247,380. Output was not affected on the Wikipedia or DBLP datasets. Upon closer inspection, the difference can be attributed to their different definitions of frequency (see Section 3.3.2) which affects subtrees of size three or less. Since the results were so close, they did not impact the performance described in the following section.

### 3.7.2 Conventional Subtree Mining Algorithms

FREQT's runtime is 1.5 to 2.0 times higher than PREFIXISPAN over the real datasets; however, neither algorithm shows any runtime complexity advantage over the other. FREQT and PREFIXISPAN both generate approximately the same number of subtree candidates to be checked for frequency; however, FREQT iterates 1.5 to 1.8 times more over the dataset than PREFIXISPAN primarily due to how infrequent trees are pruned. That is, FREQT builds all possible subtrees of a particular size $k$ in a single pass through the dataset, and then prunes all infrequent trees in a secondary pass. By contrast, PREFIXISPAN grows a single rooted subtree at a time, and prunes the infrequent nodes as it grows.

FREQT and PREFIXISPAN are only able to output results (within 24 hours) for the two smallest datasets DBLP (Figure 3.8a) and Securities while they fail to finish for any threshold on the three largest datasets—Weather (Figure 3.9b), Uniprot, and Webtraffic. Note as well that FREQT fails to complete at the lowest threshold of any dataset. Due to the exponential growth in the total length of occurrence lists, it routinely exhausts memory. This occurs because FREQT simultaneously constructs all subtree candidates of a given $k$ along with their occurrence lists and stores everything in memory. Initially, as $k$ grows, there is typically an exponential increase in the number of subtrees generated of size $k$ along with a similar exponential growth in the total length of all occurrence lists.

PREFIXISPAN utilizes one to two orders of magnitude less memory than FREQT due to its recursive nature. The algorithm first finds all labels of frequent nodes in the dataset, denoted by $l_1, \ldots, l_i, \ldots$ and constructs occurrence lists accordingly; at each iteration, an arbitrary frequent label $l_i$ is picked and only

subtrees with root label as $l_i$ using the occurrences of $l_i$ are considered. In this way, PREFIXISPAN only needs to load subtrees of size $k$ and root label $l_i$, thereby making it more memory efficient at lower thresholds and larger tree datasets (see for example Figure 3.8a). In the larger tree datasets, PREFIXISPAN does not finish because of the exponential growth in the number of subtrees. For instance, for the Weather dataset at a threshold of 0.05, PREFIXISPAN checks over 700 million growth elements and finds 350 million frequent subtrees of size 10 to 30 at cutoff time, while for the CSLOGS dataset, PREFIXISPAN mines over $10^9$ subtrees of size 50 or less before the process is terminated. Neither algorithm finishes mining the Uniprot, Weather, and Webtraffic datasets because they are simply too large; however, the algorithms do finish at the highest two thresholds for the Wikipedia and CSLOGS datasets (see Figures 3.8b and 3.9a). Shortly, we will explore how the use of a closed filter approach can address this issue.

The duplicate detection feature of FREQT only affects its performance in the DBLP dataset (Figure 3.8a) because this dataset is the only one that has duplicate labels (with the same parent node) that are frequent across all thresholds. There are two labels in the DBLP dataset that are duplicated multiple times (some as many as 11 times in a tree) across a thousand trees. Other datasets have duplicate labels across multiple trees as well. For example, Wikipedia and CSLOGS, but these labels are only considered frequent at the two lowest thresholds where FREQT runs out of memory.

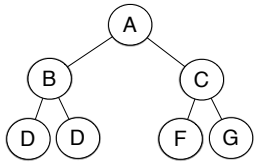
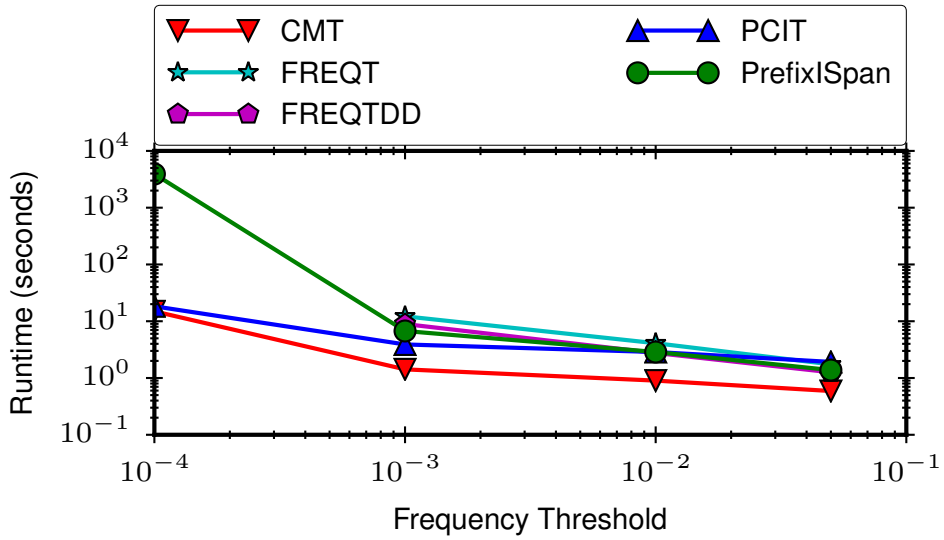
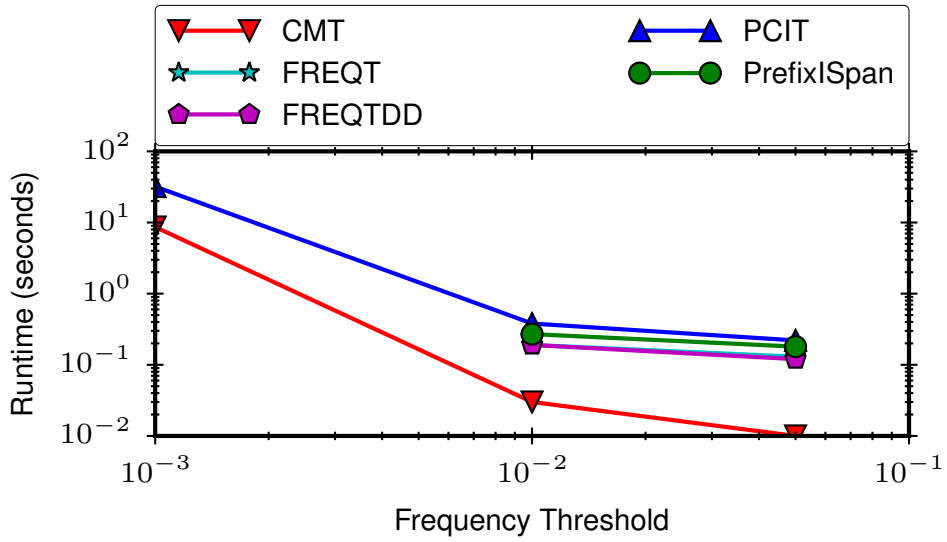



Figure 3.7: A tree with duplicate labels. The subtree A-B-D, will be mined twice in this tree without duplicate detection.

To see how duplicate labels can be a problem, take the example in Figure 3.7. This tree contains a subtree A-B-D, which has two occurrences. As mentioned above, during the mining process, subtree A-B-D will be grown along its rightmost path by a single node. In this case, two new subtrees will be formed, one with another node D attached to node B and another with node C attached to node A (call this subtree $s'$). Subtree $s'$ will actually be mined twice in the tree, once for each occurrence of subtree A-B-D, while conceptually it only needs to be mined once. The number of extra iterations over the tree is determined by the number of duplicate labels in the tree and the number of possible rightmost expansions of current subtree. The number of possible growth elements increases exponentially with tree size.

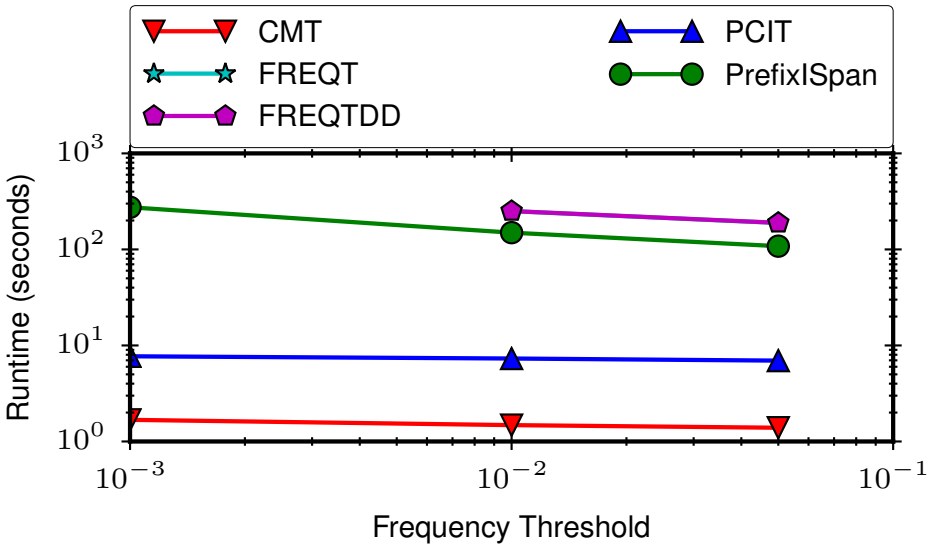



(a) DBLP dataset.

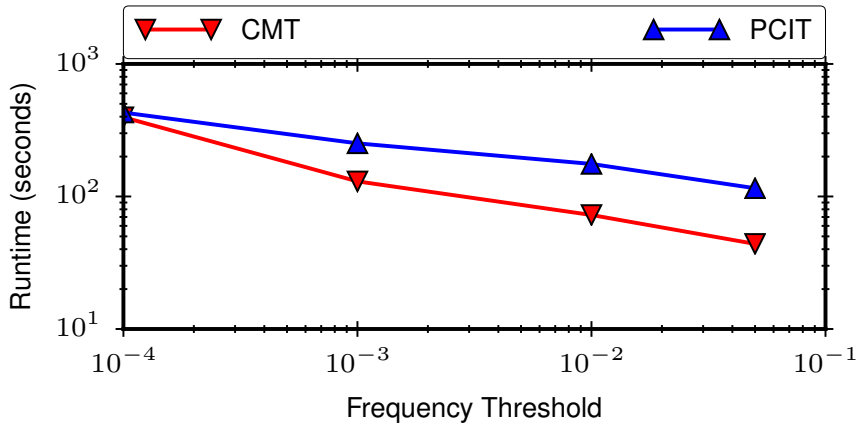



(b) CSLOGS dataset.

Figure 3.8: Graphs for real datasets.

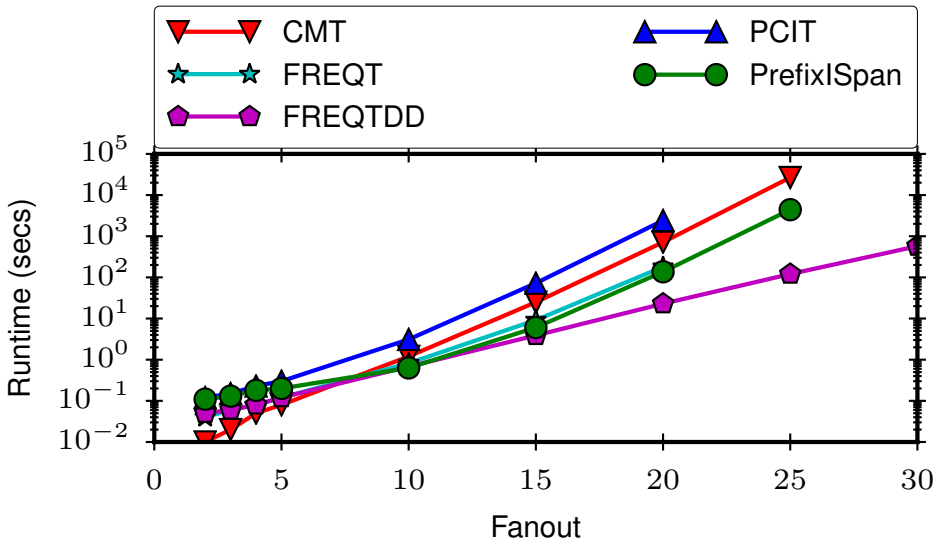



(a) Wikipedia dataset.

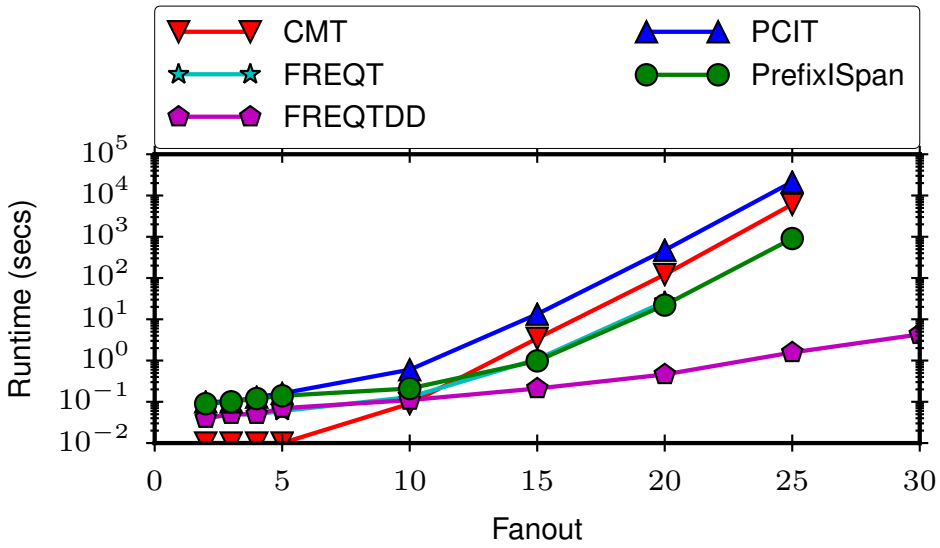



(b) Weather dataset.

Figure 3.9: Graphs for real datasets continued.




(a) $\mathcal{S}_{\text{fanout}}$ (threshold 0.001).




(b) $\mathcal{S}_{\text{fanout}}$ (threshold 0.05).

Figure 3.10: Graphs for synthetic datasets.

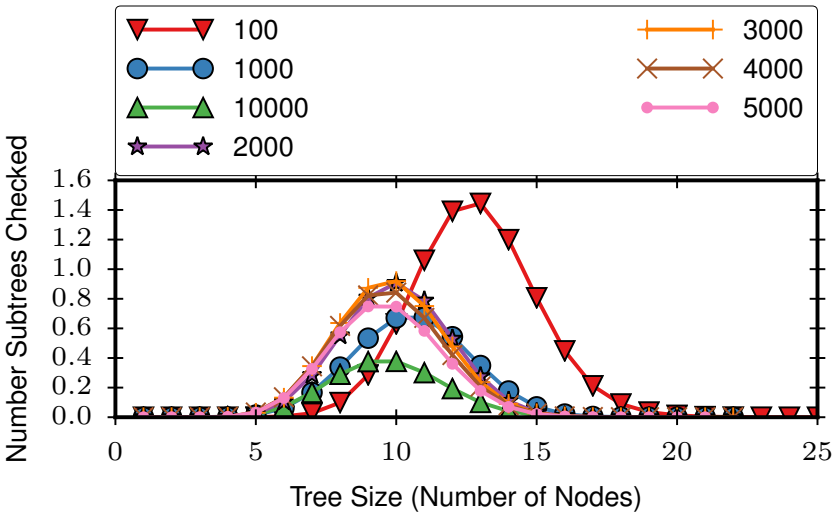
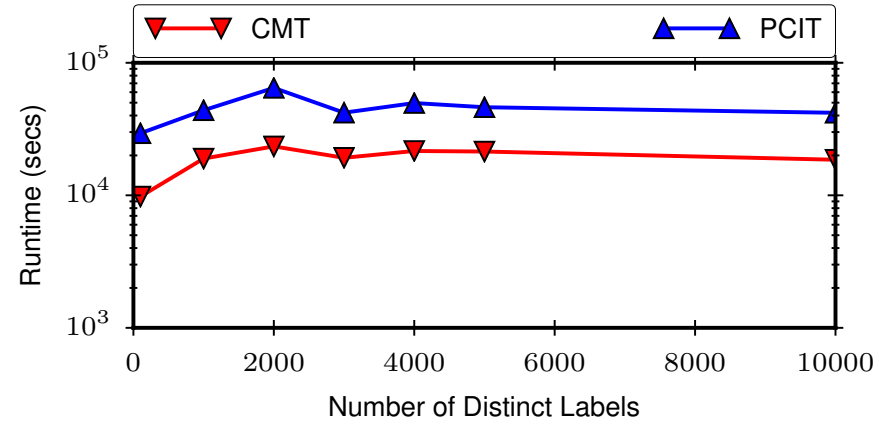



Figure 3.11: Number of subtrees checked per tree size for PREFIXISPAN on $\mathcal{S}_{\text{labels}}$ (threshold 0.001) (y-axis $x10^7$).

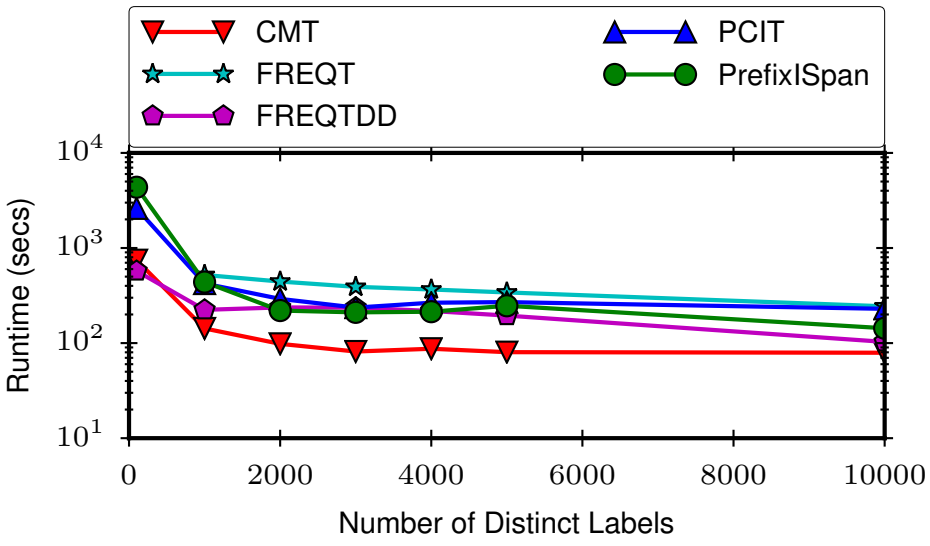
**On varying fanout – $\mathcal{S}_{\text{fanout}}$** As the fanout increases there is an exponential increase in the runtime of algorithms (Figure 3.10a). FREQT outperforms PREFIXISPAN when the size of generated subtrees is less than 10, while PREFIXISPAN performs better as the size of subtrees grows beyond 10. As the threshold increases, the size of subtrees in the larger fanout datasets begin to shrink, and the runtime of FREQT and PREFIXISPAN begins to converge (see Figure 3.10b). The performance improvement of PREFIXISPAN over FREQT is attributed to memory usage as both algorithms have similar iteration and subtree checking metrics. Indeed, at threshold 0.001 and fanout 25, FREQT accesses memory cache 10 times more than PREFIXISPAN and 78% of all cache accesses are misses compared to only 1% for PREFIXISPAN. For smaller subtrees, FREQT outperforms PREFIXISPAN because of its iterative nature.

Increasing the threshold does not significantly reduce the runtime as was seen in the real world datasets (compare Figures 3.10a and 3.10b). For example, PREFIXISPAN's runtime decreases from 4,400 seconds to 3,300 seconds as the threshold increases from 0.001 to 0.01. Digging deeper showed that duplicate labels were again an issue in this dataset, as some nodes were being repeated as many as 11 times in a tree. Once FREQT's duplicate detection feature was activated, it was able to finish at higher fanout, and complete 100 times faster than PREFIXISPAN.

**On varying fanout and depth – $\mathcal{S}_{\text{fanout:depth}}$** PREFIXISPAN and FREQT runtime performances are remarkably similar. As with the varying fanout case, FREQT has a slight performance advantage for datasets where




(a) $\mathcal{S}_{\text{labels}}$ (threshold 0.0001).




(b) $\mathcal{S}_{\text{labels}}$ (threshold 0.001).

Figure 3.12: Graphs for synthetic datasets.

the subtrees generated are of size less than 10, while PREFIXISPAN has a performance advantage for larger subtrees. FREQT with duplicate detection provides little, if any, advantage over the base algorithm, when varying fanout and depth because there are few duplicate labels.

**On varying the number of labels – $\mathcal{S}_{\text{labels}}$** The analysis shows that datasets with trees with few labels (i.e., a shorter tail on the label distribution) tend to have worse runtimes than datasets with larger label sets (Figure 3.12b). This happens because with a short tail on the label distribution, more labels are considered frequent at lower thresholds. This has three negative effects on the algorithms. Since there are more frequent nodes, subtrees are bigger and second, there are more of them (Figure 3.11). Finally, as the number of labels is reduced, there are more duplicate labels in the dataset. FREQT with duplicate detection enabled (cf. Figure 3.12b), performs better than PREFIXISPAN in almost every scenario because it iterates over the dataset 3 to 10 times less than PREFIXISPAN depending on the threshold. Furthermore, as the threshold increases, FREQT runtime advantage over PREFIXISPAN increases by more than a factor of 3.

At the lowest threshold, neither algorithm completes because the maximal subtree size grew to 55. PREFIXISPAN generates over $10^{10}$ subtrees before termination. At threshold 0.001, PREFIXISPAN outperforms FREQT because subtree sizes are larger than 15, but as the threshold continues to increase, FREQT begins to outperform PREFIXISPAN because subtree sizes reduce to 10.

**On varying number of trees – $\mathcal{S}_{\text{trees}}$** Doubling the number of trees—while keeping all other parameters constant—doubles runtime performance, on average. This is not particularly surprising, and so for brevity the results are omitted.

**DISCUSSION OF FINDINGS/LESSONS LEARNED**

The pattern growth technique was originally developed as an alternative solution to the much slower candidate generation approaches for the sequential pattern mining domain (Pei et al., 2001). Pattern growth reduced the expensive multi-pass scanning required by previous algorithms by partitioning the data into smaller manageable subsets. **That said, the analysis reveals that the advantages in applying pattern growth in the subtree mining domain are not as significant as they may appear on first blush:**

- While works (e.g., (Zou et al., 2006b; Kutty et al., 2007; Wang et al., 2004; Deepak et al., 2013)) tout pattern growth as superior to candidate generation, independent analysis found little advantage in terms of performance. *No major differences were found in how pattern growth techniques generated*

*subtrees nor how they iterate over the trees in the induced case*. The projection database (as described by Zou et al. (2006b)) and an occurrence list (as described by Asai et al. (2002)) are similar even though they use different terminology and different implementations. Both are indexes that allow the algorithms to quickly find subtree instances in the dataset without having to scan and reduce the number of trees analyzed as a subtree is grown. Furthermore, both approaches use these indexes to continuously grow their subtrees by using nodes that occur in the dataset along the rightmost path of the subtree occurrence.

- Regarding runtime performance, neither algorithm completes in the presence of subtrees larger than approximately 30 nodes. FREQT is a memory-bound algorithm, and performs best when generating subtrees of less than 10 nodes. For larger subtrees, PREFIXISPAN is the better option because it is more memory efficient. *Maximal subtree size, label distribution, number of trees, and thresholds are the biggest drivers of performance*.

- One of the big concepts noted in subtree mining studies is that the number of subtrees explodes with tree size (Chi et al., 2004). Although this is correct, it disregards the impact that labeling and threshold play on the number of subtrees. Label distribution is all too often the forgotten variable in many studies when it is likely the most important one. One can get different results by simply changing the label of the dataset. In general, I find that decreasing the number of unique labels actually degrades performance because it increases subtree size and the number of subtree occurrences in the dataset. Furthermore, less labels can result in duplicate nodes which adversely affects the runtime performance of all four algorithms. *In response, some duplicate detection mechanisms should be used, or better yet, duplicate nodes should be merged to reduce tree sizes in the dataset*.

### 3.7.3 Closed Subtree Mining Algorithms

The idea behind a closed subtree mining algorithm is to mine a subset of all subtrees in order to improve mining performance. In general introducing "closedness" into subtree mining indeed improves runtime performance; however, in certain cases, the overhead of closedness filter can outweigh its benefits.

The main difference between conventional and closed algorithms is that a closed algorithm incorporates a pruning mechanism that eliminates a subtree $s$ if there is a node $g$ that is not along the rightmost path and is occurrence matched with $s$. Subtree $s$ can be pruned because there is some subtree $s'$ that will be grown in

the candidate generation phase that will contain all the nodes of $s$ and include $g$ along the rightmost path (see Section 3.3.2). The biggest drawback of the approach is that one must keep track of the occurrences of all nodes in $s$ throughout the dataset, rather than only the rightmost node of $s$ as in conventional approaches, meaning an increased memory footprint.

The original PCITMINER algorithm only utilizes a backward scanning mechanism for pruning. It is similar to the mechanism described above, but it only searches for occurrence matched nodes that are parents of $s$ rather than all $g$ not on rightmost path. Using backward scanning alone is an inadequate mechanism for pruning subtrees in real-world datasets compared to the full pruning approach using the occurrence matched blanket. For example, even with the backscan optimization PCITMINER was unable to return results on the Weather dataset using a threshold setting of 0.05. For a more apples-to-apples comparison with CMTREEMINER, both the backward scan, and the occurrence matched blanket optimizations are used.

Figures 3.8 and 3.9 show that both PCITMINER and CMTREEMINER finish for almost all datasets except for the two largest (in terms of tree size): Uniprot and Webtraffic. For relatively small datasets such as Securities and DBLP, the closed mining algorithms outperform the conventional techniques across all thresholds (Figure 3.8a). The distinct elbow occurs because at the lowest threshold there is a long tail of labels that appear only once, which are all considered frequent. The dramatic increase in labels causes an exponential increase in subtrees generated, requiring more iterations over the data. The pruner (i.e., occurrence match blanket) employed by the closed algorithms reduces the number of candidates generated by three orders of magnitude at the lowest threshold, thereby smoothing the run time curve.

Figure 3.13 shows the number of pruned and checked subtrees generated per tree size (i.e., number of nodes) by the two closed-tree algorithms as compared to PREFIXISPAN. The close subtree algorithms mine approximate $10^4$ less subtrees than PREFIXISPAN, primarily because for every subtree checked a subtree is pruned. More importantly, the majority of pruning occurs on subtrees of less than three nodes, stopping these trees from unnecessarily growing larger, and generating an exponential number of subtrees as they grow. Notice that the number of pruned and checked subtrees dips at a subtree size of three (Figure 3.13). Trees in DBLP are of depth four, and the first two levels have few labels, while the bottom two levels have many labels that appear relatively few times. Any subtree rooted by a node in the second level is automatically occurrence matched with the root label while any node that appears once is occurrence matched by its parent. As a result, only subtrees rooted with the root label grow beyond size two. As the threshold increases, the

pruner's impact diminishes (Figure 3.8a). At threshold 0.0001, PREFIXISPAN produces $10^3$ times more subtrees than PCITMINER, but only 15 times more at 0.05.

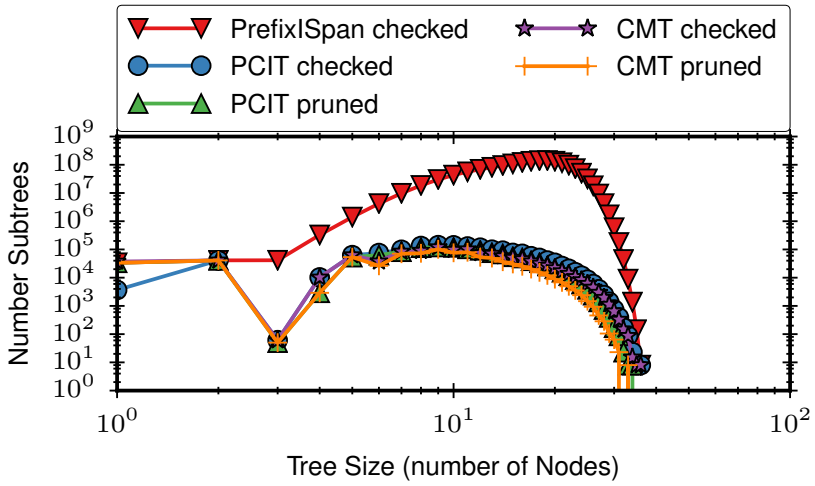



Figure 3.13: The number of pruned and checked subtrees generated per size by CMTREEMINER and PCITMINER on the DBLP dataset (threshold = 0.0001) compared to the number of subtrees generated by PREFIXISPAN.

For datasets with larger tree sizes, the subtree space increases exponentially and only the closed subtree mining algorithms are able to complete the task at hand. For example, in the Weather dataset and the highest threshold, PREFIXISPAN is hindered mining hundreds of millions of subtrees (of size 10–30) (see Figure 3.14), while CMTREEMINER checks only 64,000 subtrees, pruning 51,000 and generating only 3,100 subtrees—of sizes 1 to 50. At the lowest threshold, CMTREEMINER checks 16 million subtrees, prunes 14 million, and flags 154,000 as closed.

The results on the CSLOGS dataset are completely different from those on the Weather dataset. The reason for this difference can again be attributed to the presence of duplicate labels. To shed light on this issue, a separate experiment was conducted where the nodes in the CSLOGS dataset were relabeled by pre-pending a node's pre-order position in the tree to the label, thereby ensuring there are no duplicate labels. At the lowest threshold, both algorithms now mine 18,000 closed subtrees in just under six seconds.

The Uniprot and Webtraffic datasets are so large that none of the closed subtree mining algorithms finish for any of the support thresholds[6] tested because they all exhaust memory. Recall that PREFIXISPAN, CMTREEMINER and PCITMINER recursively grow a subtree $s$ to a subtree $s'$ by adding a single node. The state information (including occurrence lists) for $s$ is maintained in memory while $s'$ is mined. As a subtree

[6] The Webtraffic dataset completes for threshold 0.05 because there are only 14 frequent nodes.

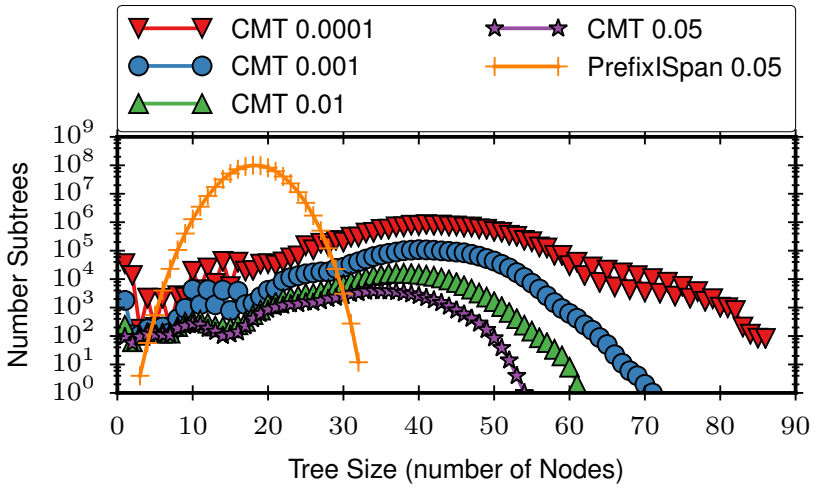



Figure 3.14: Number of trees checked by size for PREFIXISPAN at threshold 0.05 and for CMTREEMINER across all thresholds on Weather dataset. Note that PREFIXISPAN did not finish.

is recursively grown, this memory usage accumulates, and the algorithm eventually utilizes all of its memory. Note that the occurrence list for $s'$ can actually be larger than the occurrence list for $s$ in datasets where there are duplicate node labels in the same tree—as is the case in both the Webtraffic and Uniprot datasets.

Evaluations also show that memory exhaustion occurs much faster at higher thresholds primarily due to the presence of a core set of frequent nodes. For example, at the highest threshold in the Uniprot dataset there are 146 frequent nodes, 20 of which appear in 7000 or more of the trees. Furthermore, there are three node labels that appear over 100,000 times and 22 that appear over 10,000 times. The massive duplication of these nodes causes $s'$ to be multiplicatively larger than $s$.

Contrasting the performance of CMTREEMINER and PCITMINER, note that both algorithms have similar candidate generation phases. The major difference lies at the dataset iteration phase. CMTREEMINER is a basic candidate generation algorithm and given a subtree $s$, CMTREEMINER will grow $s$ by a single node $n$ at a time along the rightmost path, and create the occurrence list for the new subtree $s'$ and then begin recursively mining $s'$. The advantage of this approach is that it is extremely memory efficient for most tree datasets because only one subtree is grown at a time. Indeed, memory remains constant across all thresholds for DBLP, Securities, and Weather.

By contrast, PCITMINER grows $s$ by all its possible growth elements (nodes) along the rightmost path, creating build projections (akin to occurrence lists) for each. The technique maintains a list of projections and selects a single growth element to form $s'$ and then recursively mines that subtree before moving on to the other growth elements. Maintaining all the projections pushes PCITMINER's memory usage upwards

with lower thresholds, but allows the algorithm to iterate over the dataset less because it only has to mine all growth elements of $s$ once.

**On varying fanout – $\mathcal{S}_{\text{fanout}}$** As with the conventional mining algorithms, increasing the fanout causes a sharp increase in runtime (Figures 3.10a and 3.10b). The closed algorithms only outperform the conventional algorithms at the lowest threshold. For all other thresholds, the conventional algorithms have a runtime advantage. At the lowest threshold, the closed algorithms generate $10^3$ times less subtrees at fanout 20 than the conventional algorithms, but only $10$ times less at threshold $0.001$. The overhead associated with the pruning mechanism negates any benefit it provides due primarily to the label distribution of the fanout datasets. The leaf labels are drawn from a pool of 5,600 creating a situation where sibling nodes are less likely to be occurrence matched, and thereby reducing pruning opportunities. As in the real datasets, CMTREEMINER outperforms PCITMINER under all thresholds.

**On varying fanout and depth – $\mathcal{S}_{\text{fanout:depth}}$** The closed algorithms are best suited for the lowest threshold, and the largest fanout/depth combinations where they are able to reduce the number of subtrees checked by 2 $\sim$ 3 orders of magnitude over conventional approaches. As the threshold increased, runtime converges across all algorithms.

**On varying labels – $\mathcal{S}_{\text{labels}}$** In contrast to the conventional mining algorithms, the closed algorithms perform better when there were less labels at the lowest threshold (Figure 3.12a), because there are more opportunities for pruning through occurrence matching. As the threshold increases (Figure 3.12b) a different phenomenon drives performance: with more labels, subtree sizes shrink because less nodes are considered frequent, and the pruning effect of the closed algorithms is minimized. Notice as well that as the threshold increases, FREQT with duplicate detection begins to outperform all other algorithms.

**DISCUSSION OF FINDINGS/LESSONS LEARNED**

The findings are consistent with common wisdom (Chi et al., 2004; Kutty et al., 2007) on the relative improvement in runtime performance of closed subtree mining over conventional subtree mining. **That said, a better understanding of when conventional approaches are more suitable has been lacking. This independent analysis shows that:**

- Closed subtree algorithms have the most impact at low thresholds, where maximal subtree sizes are larger. As the threshold increases, the overhead of the closed checking begins to outweigh its benefits

and conventional approaches become more appealing. *Based on analysis, label distribution is the most important factor driving when this crossover occurs*.

- As with conventional approaches, closed algorithms are not immune to subtree size. Closed algorithms are able to comfortably mine subtrees of approximately size 100, but succumb to the same issues as conventional approaches as subtrees grow beyond much more. In experiments, for subtrees of size 1000, even the most memory efficient algorithms exhaust memory. *One could help to avert issues with large subtrees by merging duplicate nodes, partitioning trees based on some logical cut point, or removing common nodes or those deemed irrelevant to the analysis at hand*.

- Some of the findings in Kutty et al. (2007) have lead to widely held misconceptions. In their study, Kutty et al. (2007) did not compare CMTREEMINER and PCITMINER implying that since pattern growth techniques are faster than candidate generation algorithms, a comparison was unnecessary. *However, findings suggest that the underlying mining algorithm do not have as much impact on closed mining performance as the pruning mechanism*. In experiments, the backward scanning feature (used in PCITMINER) does not provide comparable performance to CMTREEMINER's occurrence matched blanket for large subtrees. Furthermore, Kutty et al. (2007) suggests that PCITMINER performs better than CMTREEMINER for datasets with high fanout; however, there is no evidence that this is the case.

### 3.8 Tree Edit Distance: An Alternative to Subtree Mining

As seen in the evaluation, subtree mining techniques are largely ineffective for tree datasets that have large variations in both labelling, and tree size. Specifically, analysis on the Webtraffic dataset indicates that subtree mining is not a good fit for the task of analyzing the structure of HTTP trees.

One of the main drawbacks of subtree mining is that it is a pre-processing step that must be done on every tree in the dataset, which is unnecessary for the task in Chapter 4. An alternative approach that enables one to encode structure into a tree analysis is to use tree edit distance. Tree edit distance is the minimum cost-sequence of node adds, deletes and renames to change one tree into another (Pawlik and Augsten, 2011). The lower the cost, the closer the two trees are in terms of structure.

Unlike subtree mining, tree edit distance does not require processing of every tree in the dataset; however, the current state-of-the-art tree edit distance algorithm called RTED (Pawlik and Augsten, 2011) has at best a $O(n^2)$ complexity with a $O(n^3)$ complexity in the worst case. The algorithm works using a dynamic

decomposition strategy that recursively decomposes the input trees into subforests by removing either the left- or rightmost nodes at each recursive step based on some optimal choice (Pawlik and Augsten, 2011). Due to the algorithm complexity, approaches that utilize tree edit distance must minimize the number of distance calculations performed on a tree dataset.

Tree edit distance is often applied to the top $k$ subtree similarity search problem, which finds the $k$ best matches of a small query tree within a large document tree using tree edit distance as a similarity measure between subtrees (Augsten et al., 2011). Augsten et al. (2011) reduces the complexity of the subtree search by estimating the upper bound on a subtree size required to perform the tree edit distance. This reduces the number of tree edit distance calculations on a dataset, but the query time still increases linearly with the number of trees in the dataset. (Cohen, 2013) 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. In Chapter 4, I propose a new approach to detecting exploit kits in network traffic, by modelling the problem as a subtree similarity search problem. The approach utilizes indices in order to reduce subtree checks.

### 3.9 Final Thoughts

This chapter presented a systematic study over the runtime performance of four representative subtree mining algorithms: pattern growth (PREFIXISPAN, PCITMINER) and candidate generation (FREQT, CMTREEMINER). In particular, PCITMINER and CMTREEMINER focus on mining closed subtrees, which reduce the runtime and number of subtrees generated. These algorithms were evaluated on seven real-world datasets and four synthetic datasets. Furthermore, limitations in an existing synthetic tree generator were identified and, as a result, three more generators were developed that produce datasets that vary attributes such as fanout, depth, label cardinality and size of dataset.

**Key Take-Aways:**

1. This chapter highlights the need for the community to be more open about datasets. Zaki (2005)'s synthetic datasets have become the defacto benchmark for tree mining experiments, yet, these datasets are treated as blackboxes by the research community. I hope that by showing the differences between the parameters used to generate the trees, and the characteristics of the output, researchers will be more conscience about describing the datasets they use.

2. In terms of the subtree mining algorithms analyzed and contrary to widely held beliefs (Kutty et al., 2007; Zou et al., 2006b), there are no significant performance advantages of pattern growth over generate and test techniques. Further, while introducing "closedness" into subtree mining improves performance over a majority of real datasets under low support thresholds, the overhead associated with closedness checking easily dwarfs this performance gain as the support threshold modestly increases. Finally, the most impactful factors for subtree mining performance as label distribution, maximal subtree size, and number of trees.

3. In this chapter, we studied two kinds of tree datasets. First, we studied trees that are structurally similar, with low label cardinality near the root of the tree. These datasets are well suited for subtree mining algorithms as long as subtree sizes remain small — 10's of nodes for contemporary mining, and 100's of nodes for closed mining. Such approaches would work well on large datasets in a cloud using a MapReduce style architecture; however, given that over 90% of the labels in these datasets only appear in one depth/degree combination, the structural information gained from subtree mining may not be that useful for many applications, and less costly approaches that disregard structure (such as frequent itemset mining (Pei et al., 2001)) may be more suitable.

4. The other kind of dataset had large variations in both labeling and tree size, making subtree mining largely ineffective. For these datasets, alternative approaches that take into consideration structural variability may be better. The subtree mining algorithms performed poorly on the Webtraffic dataset indicating that such algorithms cannot be used to model structure in HTTP traffic. Chapter 4 investigates how modelling the structure of HTTP traffic can be used to significantly reduce the misclassification rates of exploit kit instances. Given the deficiencies in subtree mining algorithms, structure must be modelled using a different approach. I investigate an approach based on the subtree similarity search problem providing a new technique for the problem where the nodes of the tree are not single labels, but where each node is represented by a large number of features. Such an approach removes the preprocessing step of subtree mining allowing it to scale to large streaming datasets.

# CHAPTER 4: DETECTING EXPLOIT KIT TRAFFIC USING SUBTREE SIMILARITY SEARCH

Network-based intrusion detectors have been plagued by several operational challenges since their inception. Two of these challenges are 1.) dealing with the sheer number of errors (misclassifications) in network traffic, and 2.) reducing the semantic gap between a reported infection, what it means, and how it occurred (Sommer and Paxson, 2010).

This chapter proposes a novel detection technique to help deal with these two operational challenges for the specific problem of detecting exploit kit instances in network traffic. The chapter investigates how to leverage the structural patterns inherent in HTTP traffic to classify specific exploit kit instances. The 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 malicious servers. This creates an inherent exploitation process which is captured 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 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.

Identifying the malicious subtree within a sea of network traffic can be modeled as a subtree similarity problem. The approach can quickly identify the presence of similar subtrees given only a handful of examples generated by an exploit kit. In order to do so, an index of malicious tree samples is built 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. As shown in Chapter 3, subtree mining is not appropriate for this problem given the variability in size and shape of HTTP trees; therefore, I propose a new solution to the subtree similarity problem based on tree edit distance.

This chapter presents several contributions including a network-centric approach based on subtree similarity searching for detecting HTTP traffic related to malicious exploit kits on enterprise networks. The work shows that using the structural patterns of HTTP traffic can significantly reduce false positives with comparable false negative rates to current approaches. Furthermore, 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, is presented. Finally, the technique is evaluated on a large network deployment.

## 4.1 Literature Review

Over the past decade, the web has become a dominant communication channel, and its popularity has fueled the rise of malicious websites (Xu et al., 2013a) and malvertising as a vector for infecting vulnerable hosts. Provos et al. (2007) 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 (Provos et al., 2008) 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. (2014) 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. (2012) 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.

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 (Invernizzi et al., 2012; Li et al., 2012, 2013; Eshete and Venkatakrishnan, 2014). The crawled websites are verified using statistical analysis techniques (Li et al., 2012) or by deploying honeyclients in virtual machines to monitor OS and browser changes (Provos et al., 2008). Other approaches include the use of a PageRank algorithm to rank the

maliciousness of crawled sites (Li et al., 2013) and the use of mutual information to detect similarities among content-based features derived from malicious websites (Wang et al., 2013). Eshete and Venkatakrishnan (2014) 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. (2014) 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. (2015) 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. (2010), for example, designed a system to instrument JavaScript run-time environments to detect malicious code execution while Rieck et al. (2010) 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 (Rieck et al., 2010; Canali et al., 2011; Blum et al., 2010; Ma et al., 2009, 2011; Mekky et al., 2014; Nelms et al., 2015). More comprehensive techniques focus on extracting javascript elements that are heavily obfuscated or `iframes` that link to known malicious sites (Provos et al., 2007; Cova et al., 2010). Cova et al. (2010) and Mekky et al. (2014) note that malicious websites often require a number of redirections, and build a set of features around that fact. Canali et al. (2011) describes a static prefilter based on HTML, javascript, URL and host features while Ma et al. (2009, 2011) 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 (Provos et al., 2008, 2007; Canali et al., 2011). 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, this chapter proposes a framework for detecting flavors of exploit kits, and utilizing the interactions between HTTP flows to reduce false positives from a small seed of examples.

Yegneswaran et al. (2005) describe a framework for building semantic signatures for client-side vulnerabilities packet traces collected from a honeypot. The work shares the similar observation that correlating

flows can help to reduce false positives; however, the work described in this chapter focuses on the specific problem of detecting server-side exploit kits using the structure of HTTP traffic by modeling kits as trees, and take advantage of structural similarity properties to reduce FPs. More recently, Stringhini et al. (2013) 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 high false positives and negatives with modest data labels and can only detect chains whereby the last node is deemed malicious. By contrast, the work in this chapter does not model client usage patterns and is not limited to the presence of redirection chains to identify exploit kits. The technique is based on structural similarity; therefore, the last node in the structure does not need to be malicious. Finally, the approach is designed to specifically reduce false positives and negatives in light of a small amount of malicious training data.

**Subtree Similarity Search Problem:** 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. Cohen (2013) 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. The work in this chapter is based on similar ideas to Cohen (2013) but works on trees where the nodes themselves have a large number of features. To make the approach practical, the work leverages the sparsity of the feature space in network traffic.

## 4.2 Approach

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. This chapter focuses 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. The 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. Those multiple requests are used to build a tree-like structure and model the problem as a subtree similarity search problem.

A high-level overview of the approach is shown in Figure 4.1. There are two main components: an index of known exploit kits (Figure 4.1 (bottom)) and an online component that monitors HTTP traffic and performs comparisons with the index to identify and label potentially malicious traffic (Figure 4.1 (top)).

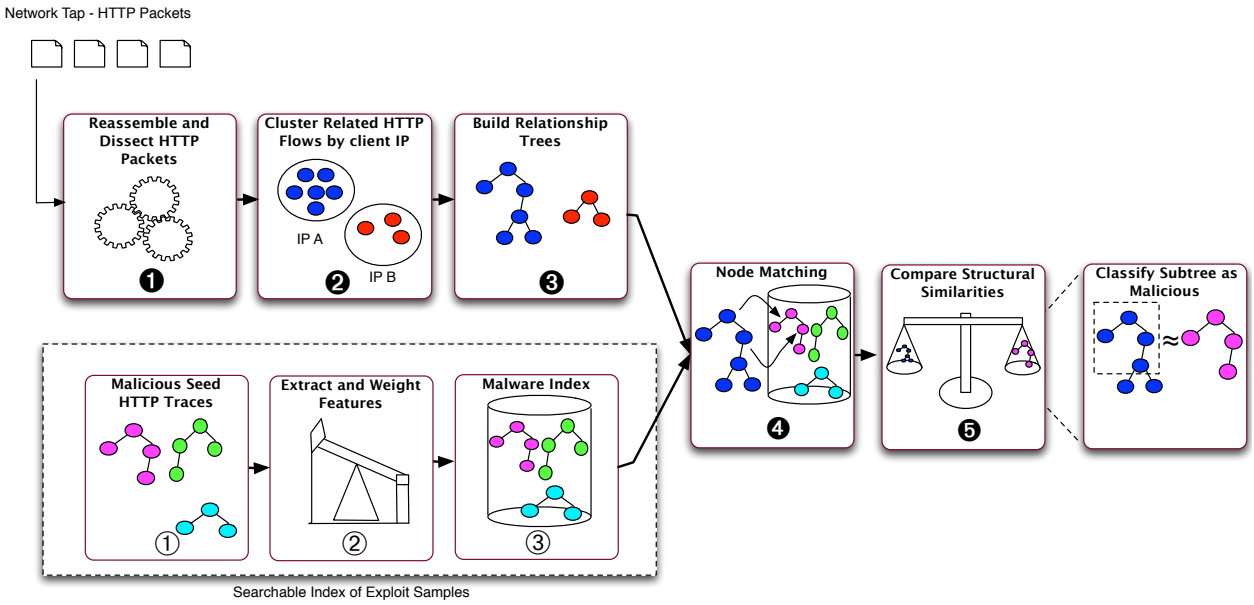



Figure 4.1: High level overview of the search-based malware system.

**Indexing stage:** In step ①, HTTP traffic samples, which represent client browser interactions with flavors of exploit kits (e.g., Fiesta) are collected and converted 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 4.2.2.

**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, § 4.2.1) 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, § 4.2.3.1), and finally, the mapped nodes are structurally compared to the trees in the index to classify subtrees as malicious (step ❺, § 4.2.3.2). 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.

### 4.2.1 On Building Trees

In both components of the system (indexing and classification), HTTP traffic is grouped and converted into tree-like structures called web session trees. A two-step process is used 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 as shown in Figure 4.2 using a technique similar to that used by Ihm and Pai (2011) and Provos et al. (2008). A web session 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). For example, a client surfing to Facebook creates a single root request for the Facebook main page, which would in turn make further requests for images, videos, and JavaScript files. All related files form a client "web session" and the relationships between these resources can form a tree-like structure as outlined below.

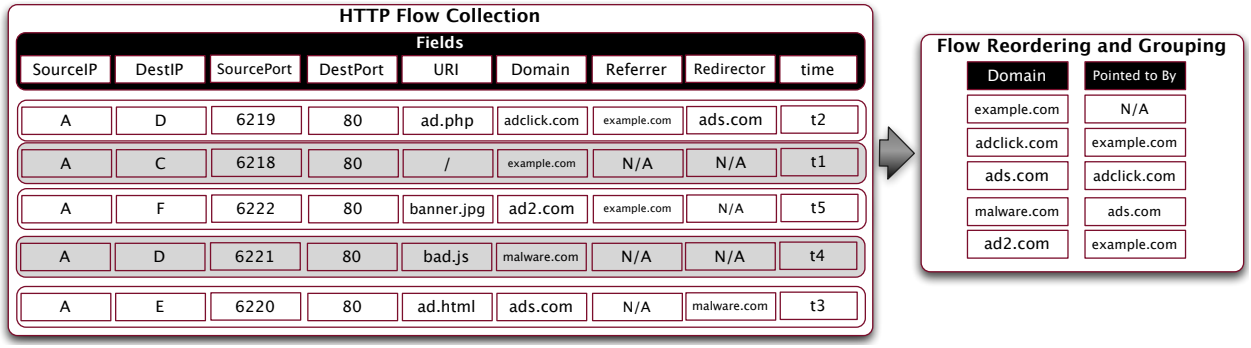



Figure 4.2: HTTP flows are reordered based on start time, and then grouped by IP and web session. This figure shows a set of HTTP flows where a client visited a fictional website `example.com` that in turn loaded ads from `adclick.com` and `ad2.com`. `adclick.com` redirects the browser to `ads.com` which redirects to `malware.com`.

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 (Ihm and Pai, 2011) 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. 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. Figure 4.3 shows the tree generated from Figure 4.2.

This tree building technique was chosen because the dataset lacked the full packet payloads required to use more complex and exact approaches (Neasbitt et al., 2014). Even so, the tree building approach used has been effectively applied in other studies (Ihm and Pai, 2011; Provos et al., 2008; Mekky et al., 2014) and aptly demonstrates the utility of the similarity algorithm. Section 5.3 discusses how the algorithm can be utilized to scalably build trees using more complex and time intensive techniques.

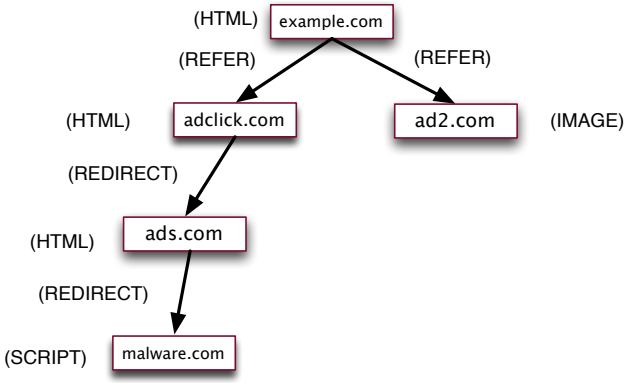



Figure 4.3: The labeled tree generated from Figure 4.2.

### 4.2.2 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 (Invernizzi et al., 2012; Li et al., 2012, 2013) 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 4.2.1, 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

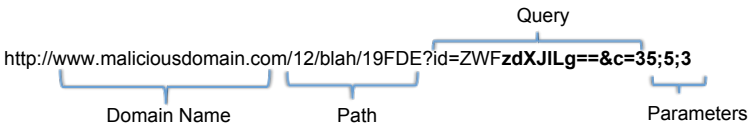



Figure 4.4: The components of a URL for feature extraction.

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 4.4 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). Six common data types are used 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 4.4 `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).

### 4.2.3 On Subtree Similarity Searches

With a malware index at hand, HTTP traffic is monitored at the edge of an enterprise network, and converted into web session trees. The next 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.

The subtree similarity search problem is approached using a two-step process: node level similarity search and structural similarity search. First, it is determined 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 the subtree $S$ containing those nodes is extracted, and compared structurally with $E$ using a tree edit distance technique. Subtrees with sufficient node overlap and structural similarity with $E$ are flagged as malicious.

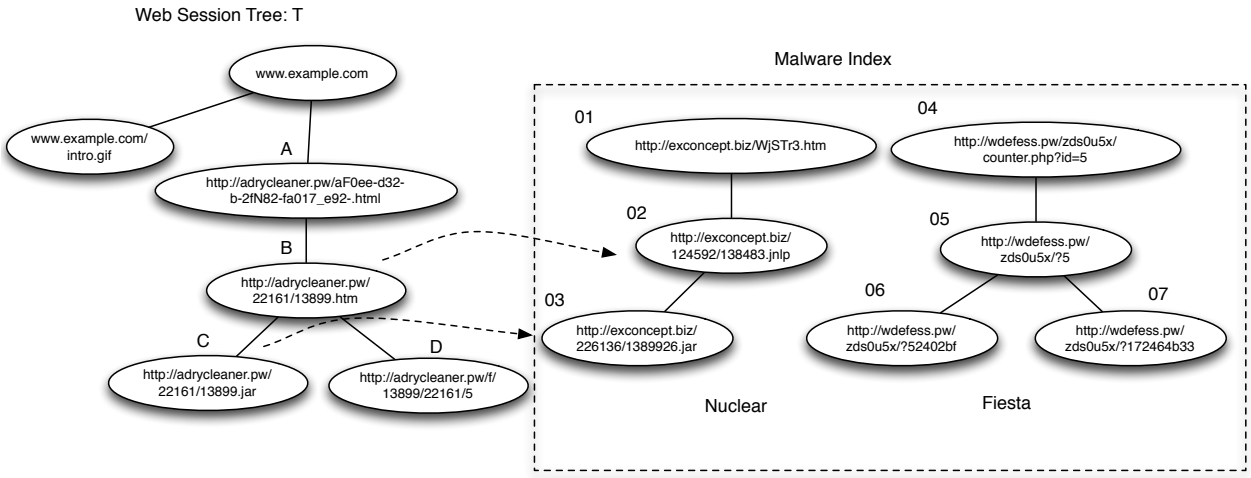



Figure 4.5: A simplified similarity search on the index. Web session tree $T$ contains nodes that are similar to nodes of one of the Nuclear trees in the index. Those nodes in $T$ are subsequently mapped to their corresponding nodes in the index to form subtrees. For example, node $B$ in $T$ maps to node $02$ and node $C$ maps to node $03$ based on node similarity scores. These node mappings form subtrees that are then structurally compared.

#### 4.2.3.1 Node Level Similarity Search

To determine whether any nodes in $T$ are sufficiently similar to nodes in the malware index, the set of token, URL structure, and content-based features are extracted 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, two node similarity approaches are compared: the Jaccard Index, and the weighted Jaccard Index to determine how weighting affects the accuracy of the algorithm. The Jaccard Index (Hadjieleftheriou and Srivastava, 2010) 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, a variant of the Jaccard Index, called relevance is used 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 $\epsilon$ 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 4.2.3.2).

A weighted Jaccard Index (Hadjieleftheriou and Srivastava, 2010) 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, a variant of the weighted Jaccard Index is used to calculate the relevance of $X$ to $I$:

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

A probabilistic term weighting technique first introduced by Robertson and Jones (1976) is applied, 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 $w(f)$, a dataset of $N$ benign HTTP flows, and $R$ tree instances from a particular exploit kit flavor (e.g., Nuclear, Fiesta, etc.) are considered. 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, the following modification is applied as suggested by Robertson and Jones (1976):

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

The 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, the node similarities scores of each tree in the malware index are compared 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 2. For pedagogical reasons, Fiesta tree samples from the malware index are used to illustrate the approach. For each tree $t$ in the set of Fiesta trees, all trees $s$ that have a tree edit distance similarity score above zero are found (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, the minimum average score is stored 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.

#### 4.2.3.2 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

**Algorithm 2** 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 ← set of all Fiesta Trees in Index
2: minval = 1.0
3: for all ( do t ← T_f)
4:     for all ( do s ← T_f)
5:         if TreeSimScore(s, t) > 0.0 then
6:             for all ( do n_s ← Node(s); n_t ← Node(t))
7:                 if score ← NodeSimScore(n_s, n_t) ≥ 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 ← Node(t) )
15:        avg = n_t.totalScore/n_t.numberScores
16:        if avg ≤ minval then
17:            minval ← avg
18:        end if
19:    end for
20: end for
21: threshold = minval
```

in tree $T$ may in fact 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 4.5 shows a simplified example of a structural similarity search. Node $B$ in tree $T$ maps to node 02 in 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 can be compared to the corresponding trees in the malware index.

Subtrees from tree $T$ are compared to the trees in the index using tree edit distance (Hu et al., 2009a). Tree edit distance uses the number of deletions, insertions, and label renamings to transform one tree into another. The ancestor-descendant relationships are enforced. 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.

### 4.3 Dataset and Training

The efficacy of the approach is evaluated using logs collected from a commercial HTTP proxy server (called BlueCoat[1]) that monitors all web traffic for a large enterprise network. The proxy server records all

[1] See `www.bluecoat.com` for more information.

client-based bidirectional HTTP/HTTPS flows from eight sensors at edge routers around the network and stores flow information in eight separate log files per hour. Furthermore, the proxy acts as a man-in-the-middle for HTTPS sessions providing a view into encrypted traffic. Each flow contains both TCP and HTTP headers.

For the first set of experiments, 628 hours worth of labeled log data spanning different days during November 2013 and July 2014 were analyzed. The log files were chosen because they contained known instances of Nuclear, Fiesta, Fake, FlashPack, and Magnitude exploit kits along with instances of a clickjacking (Huang et al., 2012) scheme refered to as ClickJack. Statistics for the dataset are summarized in Table 4.1 (labeled Dataset 1). A separate three-week long dataset from January 2014 which was *unlabeled* was used to show the operational impact of our technique. Statistics for the dataset are also described in Table 4.1 (labeled Dataset 2) and are discussed in Section 4.5.

| | **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 |

Table 4.1: Summary of datasets.

### 4.3.1 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`[2] 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.

[2] See `www.xapian.org` for more information.

### 4.3.2 Building the Malware Index

As mentioned in Section 4.2.2, 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. The malware index was populated with exploit kit samples downloaded from a malware analysis website (Duncan, 2014). The operator collected HTTP traces of exploit kits using a honeyclient and stored them in a pcap format. A tool was built that transforms these traces into HTTP trees that are in turn indexed. The 3rd column of Table 4.2 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 (Nieto, 2013).

| 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 |

Table 4.2: Testing and training sets. Exploit kits collected from www.malware-traffic-analysis.net used to build the malware index.

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 the experiment, 10 days worth of benign data from a single sensor in the BlueCoat logs were used to calculate feature weights. The benign data included over 4.4 million bidirectional flows.

Finally, the node similarity thresholds was calculated for each exploit using Algorithm 2 (§4.2.3.1). The thresholds for the weighted and non-weighted node similarity scores are shown in Table 4.3.

| 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 |

Table 4.3: Node-level thresholds computed by Algorithm 2 for weighted and non-weighted Jaccard Index.

### 4.3.3 Establishing Ground Truth

In order to establish a ground truth as a test set for the experiments, a list of regular expressions from various sources were compiled in order to identify exploit kit instances in Dataset 1. First, the Snort Sourcefire exploit kit regular expression rules from the Vulnerability Report Team[3] were run over the entire dataset. The ruleset included signatures for detecting exploit kits, such as Nuclear, Styx, Redkit, Blackhole, Magnitude, FlashPack, and Fiesta. These signatures were augmented 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, instances of the Fiesta, Nuclear, ClickJack, FlashPack, Fake, and Magnitude exploit kits(see the middle column of Table 4.2) were identified. 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 API's engines (e.g., VirusTotal, GoogleSafe Browsing, URLQuery.net, Alexa, malwaredomainlist.com, and Google).

The analysis was conducted shortly after the author of the Blackhole and Cool exploit kits was arrested in Russia (Trend Micro, 2014). Hence, these exploit kits, which were once credited with over 90% of new infections (Trend Micro, 2014), collapsed leaving attackers scrambling to find an alternative. Although there were no traces of the Blackhole or Cool exploit kits, there were many instances of the Fiesta and Magnitude kits, which became prevalent after Blackhole's demise (Symantec MSS Global Threat Response, 2014). Recent studies (Symantec MSS Global Threat Response, 2014; Grier et al., 2012) 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.

## 4.4 Finding the Needle in a Haystack

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

[3] See `www.snort.org/vrt` for more details; Date accessed: July 2014.

### 4.4.1 Comparison with Snort

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

■ *Fiesta*: In evaluating Fiesta, the approach was compared 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. The structural similarity technique focuses on the structural path of flows taken to arrive at the exploit payload. As such, in the structural similarity technique, not only were these accidental matches that are unlikely to share similar structures with Fiesta instances eliminated, but also the exploit was identified before the payload is reached. The results are summarized in Table 4.4.

Table 4.4 shows that using structure eliminated all 597 false positives flagged by the Snort rule and also identified cases that Snort missed. In most cases, the structural similarity 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. The approach 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, three Snort rules 28594, 28595, and 28596 were used. These rules search for numeric `jar` and `tpl` file name of malicious payloads as well as specific directory structures in URLs. As noted in Table 4.4, 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 the similarity approach 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. The structural similarity 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 the 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*: Magnitude was evaluated using Snort rules 29188 and 28108, which search for hex encoded `eot` and `swf` files, respectively. Results for all techniques are shown in Table 4.4. 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, there were zero false positives and zero false negatives.

■ *FlashPack*: The 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. Experimenting with a much looser regular expression identified all instances; however, it generated over 43,000 false positives.

The structural similarity approach was 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 (as discussed 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 4.4). 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 (Huang et al., 2012). To detect instances of the ClickJack kit, a single instance of its structure was loaded 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. The structural similarity approach identified 130 instances of the clickjacking scheme with zero false positives and zero false negatives. 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. An online version of the approach would have been able to detect the exploit before it was downloaded.

■ *Fake - Installer*: The 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. An analysis was conducted on the dataset specifically looking for the checker.php file and 1,200 cases of this file were found in a three month period. Of those 1,200 cases were confirm 575 to be the Fake Installer. The structural similarity approach successfully identify all such cases with zero false positives and zero false negatives.

**Summary:** Table 4.4 summarizes the detection results of the structural similarity approach and Snort. Regarding exploit kits for which Snort rules are available (i.e., Fiesta, Nuclear, Magnitude, and FlashPack), the structure similarity 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, the 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.

### 4.4.2 Comparison with State of the Art

The structural similarity approach is compared with a statistical classifier proposed by Ma et al. (2011). The classifier is based on logistic regression with stochastic gradient descent using features similar to those described in Section 4.2.2. The classifier labels URLs as either malicious or benign and is trained with all

| 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 | - | - | - |

Table 4.4: Comparison (weighted) to Snort signatures.

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 4.5 indicating that the classifier performed well at detecting exploit kit instances. The classifier was able to detect two more instances of Fiesta than the structural similarity approach because both clients visited a landing page for an exploit kit, but did not reach a payload, exposing no web structure for the structural technique to detect. In the case of Nuclear, the classifier was unable to identify the instances that only used `.tpl` and `.pdf` file types.

The classifier flagged over 500,000 URLs as malicious in Dataset 1. Through a painstaking analysis of the URLs using malware reports, blacklists, and google searches, 4,000 of the URLs were confirmed to be malicious — 2,500 of the URLs were associated with the exploits kits found as ground truth in Dataset 1 (Table 4.2), which were also detected by the structural similarity approach. Note that Table 4.2 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 (Yadav et al., 2010), 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. Due to the high false positive rate, the approach of Ma et al. (2011) is infeasible in an operational environment.

| 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 |

Table 4.5: Comparison (weighted) to binary URL classifier.

Stringhini et al. (2013); Mekky et al. (2014); Cova et al. (2010); Eshete and Venkatakrishnan (2014) 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. In this chapter, the intention was to provide a comparative analysis to Stringhini et al. (2013), but, the approach of Stringhini et al. (2013) 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, this chapter evaluates 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, a subset of 50 traces were manually analyzed using an instrumented HTML parser, javascript engine(Rhino) and DOM (envjs) in order to build chains that included javascript redirections. An evaluation 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. (2014) 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 (Eshete and Venkatakrishnan, 2014).

Not only were there not large redirect chains for exploit kits, but there are comparable length redirect chains in benign traffic due primarily to advertising networks. Server and meta redirection chains were built using 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, the structural similarity 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.

### 4.4.3 Findings and Discussion

Let us 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 4.2) and those in the malware index. The results are shown as a cumulative distribution function in Figure 4.6. 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 2 while all nodes associated with malicious trees had a node similarity score of 0.22 or higher.

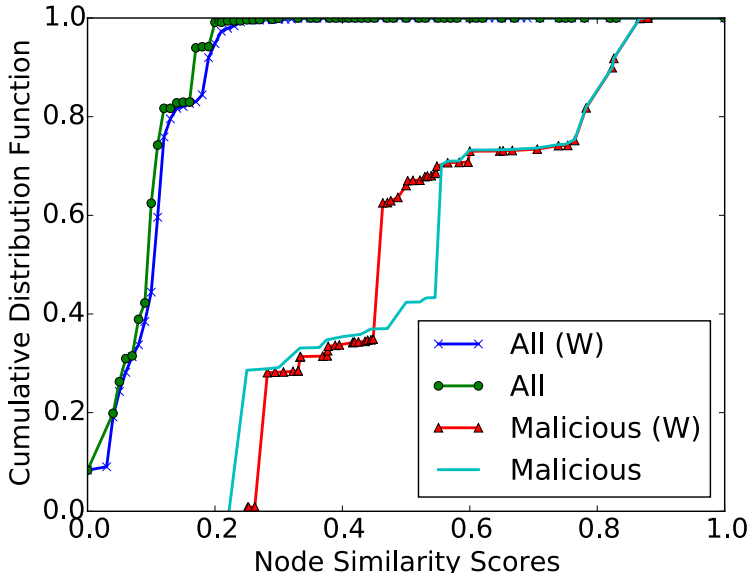



Figure 4.6: The CDF of node similarity scores between our test dataset and the malware index. "All" represents similarity scores between all nodes in the dataset and the malware index, while "Malicious" represents the node scores for trees in the dataset that were flagged as malicious(W = weighted).

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 4.6, 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. 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 4.6, there were a large number of false positives if only node-level similarity (like Snort signatures that focus only on individual flows) was considered 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-descendant 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.

| | **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 |

Table 4.6: FPs from the approach when considering single node matching, multi-node matching without considering structure, structure similarity, and structural similarity with ancestor-descendant requirement for weighted and non-weighted similarity.

As shown in Table 4.6 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 4.1). This result can be leveraged, by only building trees for flow clusters that have multiple similar nodes in common with a tree in the malware index, allowing the scalable application of much more computationally expensive (and correct) tree building techniques to the wire (i.e., (Neasbitt et al., 2014)) when full payloads are available. Table 4.6 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.

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 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 4.7 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. 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 for maximum flexibility and reduced false negatives and false positives over contemporary approaches.

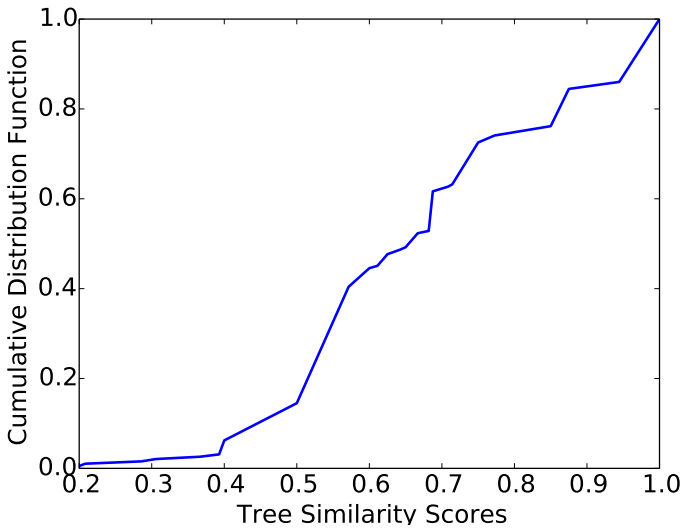



Figure 4.7: The CDF of tree similarity scores for malicious subtrees.

## 4.5 Operational Deployment

To further demonstrate the utility of the structural similarity approach in a large enterprise environment, three consecutive weeks of BlueCoat logs were analyzed from January 6-31, 2014 (Dataset 2 in Table 4.1) using the weighted version of the 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 28 exploit kit instances were identified with no false positives, compared with Snort signatures that generated over 22,000 false positives and missed most of the Fiesta instances, as shown in Table 4.7. 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. Furthermore, two of the Clickjacking instances downloaded Popup Trojans. The URL classifier of Ma et al. (2011) generated an average of 143,000 alerts per day for a total of 3.6 million alerts in the month and the sheer volume of alerts made it infeasible to vet each flagged URL.

| **Exploit kits** | **Structural Similarity** | | **Snort** | | |
|---|---|---|---|---|---|
| | TPs | FPs | TPs | FPs | FNs |
| Fiesta | 20 | 0 | 2 | 340 | $\geq 18$ |
| 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 | - |

Table 4.7: Live comparison to Snort signatures.

The fact that the structural similarity approach successfully detect these abuses on a large enterprise network underscores its operational utility. 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 runtime performance, the processing speed was evaluated 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 the current prototype is able to process an entire day of traffic in 8 hours. Figure 4.8 illustrates the performance breakdown of different components of the 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.

Note that although the 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 the current prototype falls short of processing at that speed, note that by design, all the components (e.g., flow parsing, tree building and feature extraction) are parallelizable; as such, with modest hardware provisions the prototype could efficiently handle the peak loads.

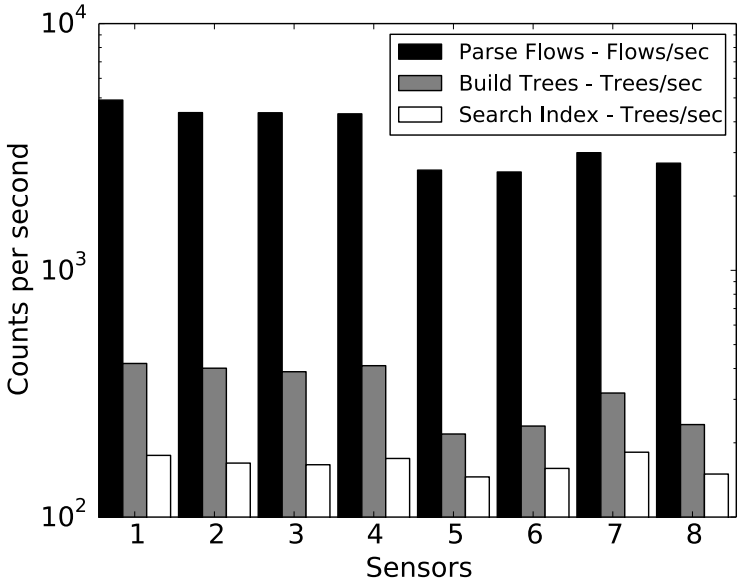



Figure 4.8: The performance of bidirectional flow parsing, tree building, and malware searching for one day of data across 8 sensors.

## 4.6 Limitations

As with any security solution, attackers will inevitably seek ways to bypass it. An obvious evasive strategy would be to hinder the 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, the two step similarity algorithm can 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 (Neasbitt et al., 2014), JavaScript de-obfuscation (Lu and Debray, 2012; Xu et al., 2013b), and statistical means (Hu et al., 2009b; Nelms et al., 2015) — all of which could be easily adopted in this

setting to thwart evasive techniques. 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 the current network-centric approach. Nevertheless, the main focus of this work is not in building better HTTP trees, but rather to demonstrate the benefits of a tree structure-based detection approach in reducing false negatives and false positives.

In addition, because the structural similarity 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, 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, the 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.

From an operational perspective, the fact that the structural similarity 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. 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 (Yegneswaran et al., 2005) 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 chapter.

## 4.7 Discussion and Lessons Learned

In closing, this chapter presents a novel network-centric approach that uses structural similarity to accurately and scalably detect web-based exploit kits in enterprise network environments. The prototype implementation, which was evaluated on real world data collected from a large-scale enterprise network, worked remarkably well. In particular, 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. A preliminary analysis in an operational deployment demonstrates that the technique can easily scale to handle massive HTTP traffic volumes with only modest hardware requirements.

**Key Take-Aways:**

1. While the attacker could make drastic changes to the exploitation process, it is unlikely because the multi-step process actually helps to prevent kit detection by the defender. The attacker first checks the versions of the software installed on the client, and tests whether the client is a honeyclient so that he does not reveal his exploit unless he is absolutely sure the client is legitimate and vulnerable. The attacker also spreads components of the exploit across multiple web files in order to prevent the defender from analyzing single files in isolation. By utilizing the entire exploitation process for detection, the defender forces the attacker to make a choice between leaving the inherent structure in the exploitation process or reveal the exploit sooner — a win for the defender because it makes it easier to adapt virtualization approaches to network defense. As a result, utilizing kit structure should be a valid detection approach for years to come.

2. The proposed subtree similarity search approach provides a fast and scalable technique for classifying structural data with high dimensional feature spaces. The approach does not require subtree mining, which can be a processing bottleneck, making it attractive for solving classification problems in other research areas including biology, text mining, and natural language processing.

3. Modeling the structure of web-based traffic not only drastically reduces false negative rates over current approaches, but also, the structure provides context for the security analyst. As a result, the analyst can determine the origin of the exploit and take appropriate action to mitigate future attacks. One of the important challenges for the research community moving forward will be to provide these contextual

linkages over much larger time windows in order to more efficiently retrace the steps of an exploited computer pre and post infection.

4. As demonstrated in the evaluation, one of the most expensive procedures in network-based intrusion detection is feature extraction. This forces current detection techniques to choose "cheap" features in order to keep up with the fire hose of traffic on large networks. Unfortunately, this leads to feature selections that are not necessary and sufficient for detection. Chapter 5 addresses the issue by providing a different model for network-based detection with a behavioral feature set. Behavioral features are better because there is a clear distinction between benign and malicious behaviors.

5. Establishing ground truth is one of the most difficult and time consuming tasks related to network security research. The process is counterintuitive as ground truth is established using some of the very tools a researcher is comparing against. One must analyze the dataset with as many automated tools available, and then manually investigate the results in order to fine tune the process. Chapters 5 and 6 will discuss other strategies for establishing ground truth.

# CHAPTER 5: DETECTING EXPLOIT KIT TRAFFIC USING REPLAY

Chapter 4 demonstrated that utilizing context in network-based malfeasance detectors can have real operational benefits for the security analyst. Unfortunately, the presented approach has two key limitations that needs to be addressed. Like any other signature or statistical approach, the technique must make assumptions about what are "malicious features", based on the characteristics of previously seen malicious samples. As such, the technique is unable to detect previously unseen exploit kits. Furthermore, to keep up with the deluge of traffic, the features chosen are characteristic of previously seen malfeasance, but are not necessary and sufficient to be malicious.

In order to the address these issues, one can look to extract behavioral features that describe what an exploit is actually doing. To do so, one can use honeyclient analysis. 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 (Bailey et al., 2007). Security vendors routinely crawl the Internet with large clusters of VMs in an attempt to identify malicious websites (Thomas et al., 2015; Grier et al., 2012). The result of these analyses are typically used to generate blacklists or other information deemed useful for improving a network's security posture.

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 network-based exploit kit detection, such prohibitively long processing times make them poorly suited for live operational deployments. 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 the UNC campus network, for example.

Motivated by a *real operational need* to tackle the threats posed by the significant rise in Flash-based attacks, this chapter presents 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 chapter 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 faced is that the analysis performed must be done without any human intervention and without storing personal information on non-volatile storage. These privacy restrictions are not unique to a single environment, and it means that security researchers are left with no option but to process the fire hose of network data judiciously and expeditiously. The approach described in this chapter leverages 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, the same contextual requirements afforded in the last chapter must be considered here in order to detect these files.

In designing, deploying and evaluating this framework, several obstacles were overcome and the following contributions made:

- 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 chapter is organized as follows. A literature review is presented in §5.1. The framework for enabling the use of honeyclients on-the-wire is presented in §5.2. A performance evaluation, as well as a case study of real-world attacks is provided in §5.3. Limitations and future work are discussed in §5.5. Key take-aways and lessons learned are described in §5.6.

## 5.1 Literature Review

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. (2007) 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 (Provos et al., 2008) 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. (2012) 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. (2015) 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 (Invernizzi et al., 2012; Li et al., 2012, 2013; Eshete and Venkatakrishnan, 2014; Thomas et al., 2015). The crawled websites are verified using statistical analysis techniques (Li et al., 2012) or by deploying honeyclients in VMs to monitor environment changes (Provos et al., 2008). Other approaches include the use of a PageRank algorithm to rank the "maliciousness" of crawled sites (Li et al., 2013) and the use of mutual information to detect similarities among content-based features derived from malicious websites (Wang et al., 2013). Eshete and Venkatakrishnan (2014) 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 (Wang et al., 2011).

Gassen and Chapman (2014) 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. (2014) 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. (2015) clustered exploit kit samples to build host-based signatures for anti-virus engines and web browsers.

More germane to the work described in this chapter are approaches that try to detect malicious websites using HTTP traffic. For example, Cova et al. (2010) designed a system to instrument JavaScript run-time environments to detect malicious code execution, while Rieck et al. (2010) described an online approach that extracts all code snippets from web pages and loads them into a JavaScript sandbox for inspection. 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. The approach described herein 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 (Rieck et al., 2010; Canali et al., 2011; Blum et al., 2010; Ma et al., 2009, 2011; Mekky et al., 2014; Nelms et al., 2015). More comprehensive techniques focus on extracting JavaScript elements that are heavily obfuscated or `iframes` that link to known malicious sites (Provos et al., 2007; Cova et al., 2010). Cova et al. (2010), Stringhini et al. (2013), and Mekky et al. (2014) note that malicious websites often require a number of redirections, and build a set of features around that fact. Nelms et al. (2015) studies the webpaths users take to malware downloads and builds a classifier to label them in the wild. Canali et al. (2011) describes a static prefilter based on HTML, JavaScript, URL and host features while Ma et al. (2009, 2011) 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 (Provos et al., 2008, 2007; Canali et al., 2011). These techniques take broad strokes

in terms of specifying suspicious activity. For example, Provos et al. (2008) reported a 10% false negative rate and Canali et al. (2011) reported a false positive rate of between 5% and 25%, while Provos et al. (2007) 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, the approach described herein focuses on *behavioral* aspects of malware to help reduce false positives and false negatives.

Schlumberger et al. (2012) 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. (2012) 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. (2012) implements features that are meant to determine whether code or data obfuscation has been used, and whether known vulnerable functions have been used. 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 (Van Overveldt et al., 2012)) since obfuscation is commonly used by benign files. The approach described in this chapter does not use obfuscation or known vulnerable functions to make a final decision, thus there is a lower false positive rate.

By far the most popular means of network protection are NIDS, such as Bro (Paxson, 1999) or Snort (Roesch et al., 1999), 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. (2008) 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, the described 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).

The framework described in this chapter provides the best of both worlds between statistical approaches and honeyclients by bringing the honeyclient to the network. As a result, it can identify new exploits on-the-fly and mitigate threats more swiftly than the current state of the art.

## 5.2 Approach

The 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. The hypothesis is 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, there must be a mechanism 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 the framework. The high-level depiction of our workflow is given in Figure 5.1.

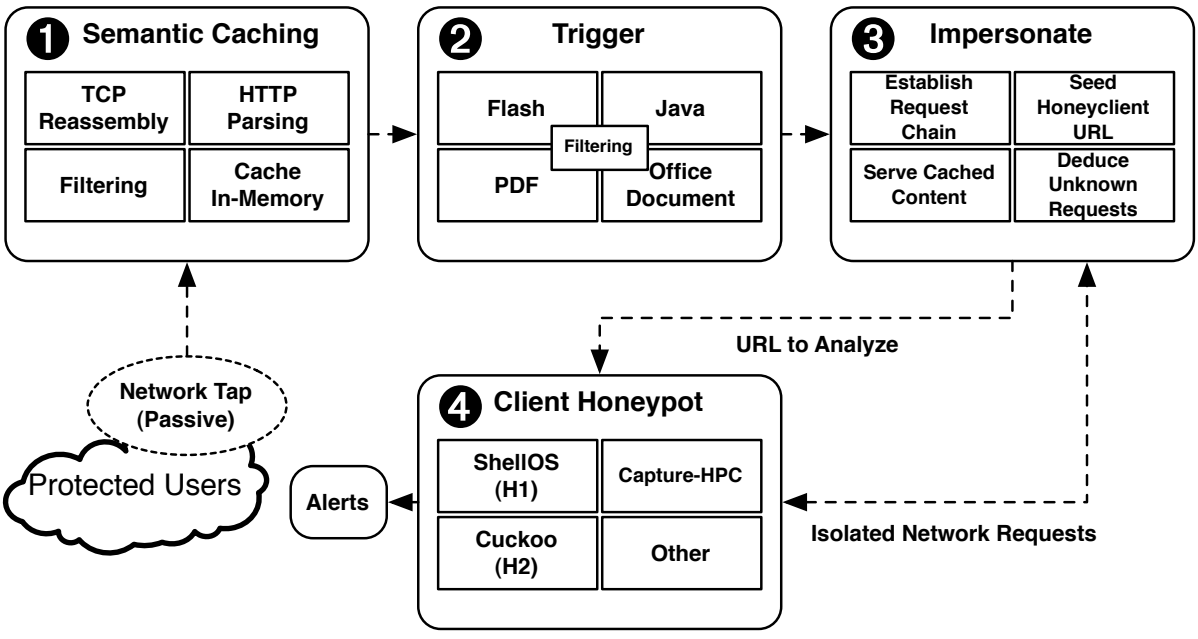



Figure 5.1: Overall workflow of enabling an on-the-wire honeyclient.

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, a 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, the challenges and solutions provided for each component in the design are discussed.

### 5.2.1 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, one cannot afford this luxury. For privacy reasons, we cannot 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. There is 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 the approach, caching observed content at the application layer is used rather than at the network layer as proposed by Maier et al. (2008). As packets cross the network border, they are reassembled 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 the data streams are reassembled at the HTTP-level, making each request header and associated response content transparent to the 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 in designing the 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 the 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. A *least recently used* (LRU) caching data structure holds 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.

Later, the chapter discusses how this cache is utilized for honeyclients in §5.2.3, but for now, the next section investigates how one can use this information to hone in on potentially malicious web traffic in an overwhelmingly benign sea of traffic flows.

### 5.2.2 Step ❷: Filtering and Triggering

One significant challenge in the design of the framework lies in the ability to scale to provide a timely analysis of each observed request. 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. The 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 they are additionally filtered using a file format specific mechanism.

To guide the efforts in designing file format specific filters, the observed downloads are measured on the UNC campus network over the course of a single school day (see section §5.3). 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 do not require filters, as it is unnecessary. There are an average of 7.4 `pdf` files a minute. 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 (Snow et al., 2011).

The same cannot be said for JavaScript files. There were 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, 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 (Antonakakis et al., 2010). Given the current challenges in analyzing Flash, JavaScript filtering is left as future work.

As noted earlier, there were 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. 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. (2009) use URL features to classify requests as malicious, while Cova et al. (2010) 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. 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). More discussion on this later in §5.3.

Instead, the framework was designed with a *whitelisting* approach in line with the goal of using honey-clients to detect previously unseen, or zero day, attacks. The 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, two filters are designed. 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. The approach to handling this second type of ad is more involved. More specifically, with the second there is a 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, Flash files observed are parsed on the network and extract individual function byte-code. The

byte-code is hashed at the function level to create a piecewise or fuzzy hash (Kornblum, 2006). Then, for each new Flash file analysis is only triggered if it has at least one function that is not in the function-level hash store. If an attacker attempts to masquerade their Flash exploit as a benign ad, it still triggers 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 the presentation, the evaluation centers on an in-depth analysis of Flash exploits in §5.3. At this point there is a cache of web objects and a desire to perform a honeyclient analysis based on the observation of a potentially malicious Flash file. It is time to investigate 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.

### 5.2.3 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 one 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, context is recreated and client and server configuration is replicated 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, the 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 (Acar et al., 2013), which is turned into a valuable resource for the framework's purpose. Due to data collection limitations on the campus network, collection of browser information is limited to 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 (Acar et al., 2013). 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 a new algorithm is introduced, dubbed the *chaining algorithm* (Algorithm 3), that operates as follows. During the two-level caching step of the workflow (step ❶ §5.2.1), 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.

**Algorithm 3** 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
```

Next, Algorithm 3 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 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. One determines 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 §5.4, for example).

One of the major challenges in the 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, the client IP is passed 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. 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.

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.

### 5.2.4 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, a honeyclient is defined 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.

The experiments in §5.3 make use of unmodified versions of Cuckoo Sandbox[1] and ShellOS (Snow et al., 2011; Stancill et al., 2013). These two approaches were chosen 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. 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 the current analysis), *e.g.*, a heap spray is likely to have occurred.
- The process terminates or crashes.

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. Cuckoo Sandbox is used 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.

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

- The process accesses registry or system files.
- 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, ShellOS is referred to as as $H_1$ and Cuckoo as $H_2$ in §5.3. Evaluations show that monitoring system state with either of these approaches significantly improves detection performance over content-based signatures.

### 5.2.5 Prototype Implementation

The prototype implementation consists of 8192 lines of custom C/C++, Java and Golang code. The *libnids* library provides TCP reassembly. A Go IO reader interface was implemented 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 §5.2.1. 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 the campus network necessitated filtering for Flash file triggers, as described in §5.2.2. Flash parsing and fuzzy hashing is all custom code written in Go, as is the implementation that impersonates the attack server. For the headless browser, HTMLUnit[2], an open source implementation was chosen and is written in Java while incorporating 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.

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

## 5.3 Evaluation

To demonstrate the efficacy of the framework both an offline evaluation with known exploit kit traces and an online analysis on a large network were conducted. In short, findings suggest that on-the-wire honeyclients consistently out-perform signature-based systems by discovering exploited clients days and weeks ahead of those systems. Furthermore, 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 5.1). File types such PDF and EXE are typically self contained and can be analyzed directly within a sandbox without loading a full website (Snow et al., 2011). 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 — the framework does not support user interaction. Finally, as shown in Table 5.1, 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 5.1) prevalent web objects on a network, and as such, presents significant scalability challenges. While JavaScript-only drive-by-download attacks are not addressed in this work, malicious JavaScript that is used to load a trigger file (*e.g.*, Flash) is detectable. Furthermore, the lessons learned from analyzing Flash will be invaluable in future work on full scale JavaScript analysis.

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

Table 5.1: Number of instances of various file types seen on campus on a busy school day.

### 5.3.1 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 §5.2.3). Honeyclient results are then contrasted to the results of 50 antivirus engines[3].

A total of 177 HTTP publicly available packet trace samples of exploit kits[4] were inspected. 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. The dataset is representative of the diversity of real-world attacks that would be encountered if the framework were to be deployed on any large network.

| 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 5.2: Detection results for the framework when using honeyclient $H_1$ on the 10 exploit kits by detection type.

**On-the-wire Performance of Honeyclient $H_1$** Table 5.2 shows the evaluation results for the 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). 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 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 (Polychronakis et al., 2010) 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.

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

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

| Exploit Kit | Process Launched | 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 5.3: Detection results for the framework when using honeyclient $H_2$ on the 10 exploit kits by detection type.

**On-the-wire Performance of $H_2$** The results when using $H_2$ with the framework are shown in Table 5.3. 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, showed that the exploits are *unhooking* four Windows API calls that are used by attackers to determine whether they are operating in a virtualized environment.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 the 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 (§5.4) 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.

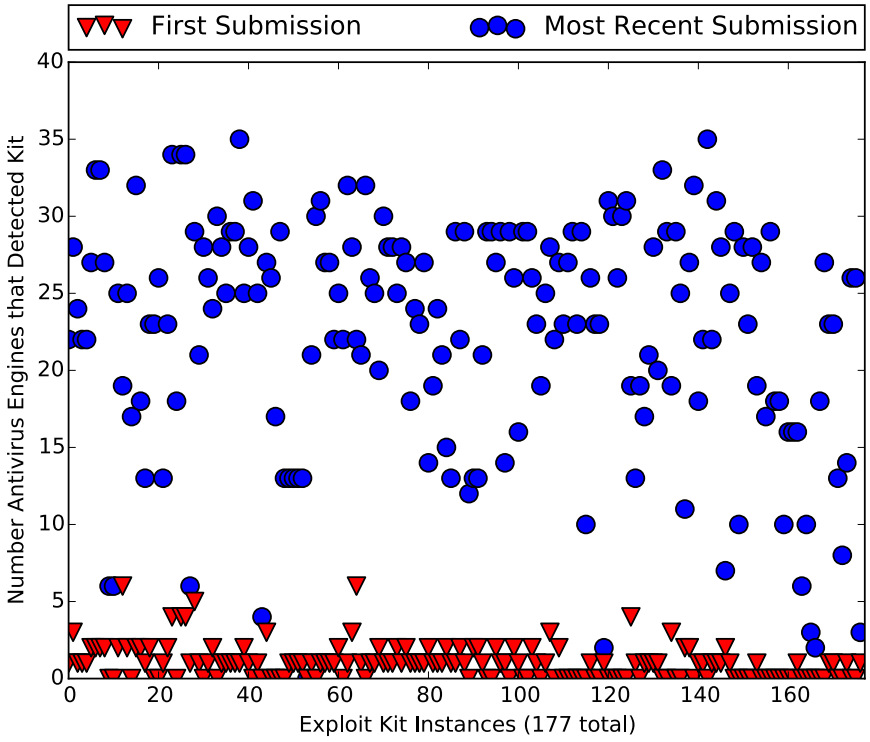
Note that the design and implementation of specific honeyclient technologies is an ongoing research topic, but the primary goal of the work is to provide a framework that effectively leverages such advancements on-the-wire. To that end, these experiments confirm the efficacy of the approach by providing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

**Content-based Signature Comparison** The performance of honeyclients using the framework is compared with that of content-based signatures, *e.g.*, antivirus engines. Each exploit file associated with all 177 HTTP traces is checked 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, 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. 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. The signature engine performance is significantly 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*.

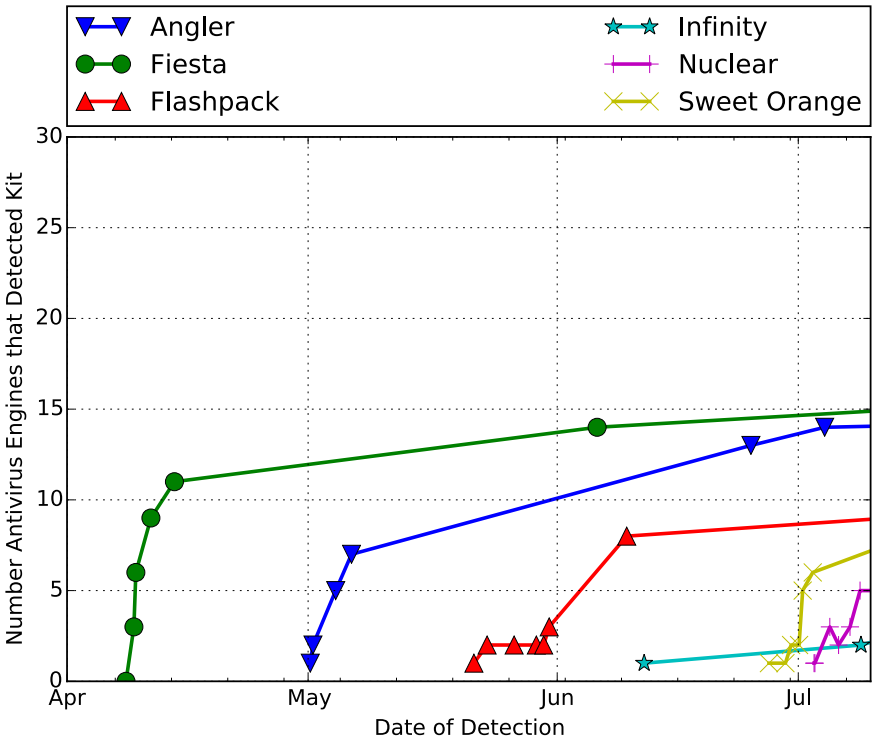
The experiments confirm the aforementioned hypothesis. The results of this analysis are depicted in Figure 5.2. Figure 5.2a shows that at initial attack time, 69 of the exploits go completely undetected by all engines. In other words, the best engine has no more than a 61% true positive rate. Another 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. 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. Six exploit kit instances from the sample set were randomly selected and analyzed to see how many engines detected the instance over time starting from the initial observation to the last, as seen in Figure 5.2b. 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

[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.

Figure 5.2: Analysis of the 177 exploits on VirusTotal.

each exploit instance. 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. Honeyclients have no pre-conceived 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 §5.2.4. The framework detects attacks on-the-wire when it matters – as they happen.

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

### 5.3.2 On Live Traffic Analysis

The next experiment focuses on detection in the face of significant background traffic. That is, experiments in this section demonstrate that the framework can successfully detect exploits from the larger haystack of benign traffic while maintaining a negligible false positive rate. The framework was run on a network for a 5 day period in November 2015. During the evaluation period, the campus network was going through infrastructure upgrades and the tap was susceptible to TCP reassembly errors due to asynchronous packet routing and the fact that we did not have collection coverage on one of the network borders. As a result, our evaluation focuses on the traffic loads that we were successfully able to reassemble and tcp reassembly errors are discussed throughout the evaluation. We saw peak traffic loads of 2 Gbps during the analysis.

The 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, the framework used the $H_1$ honeyclient running with five VMs, enabling five concurrent analyses supporting Chrome, Internet Explorer and Firefox browsers. The online analysis focuses on $H_1$ because the platform was developed internally and could easily be modified to support multiple browsers and Flash plugins, while debugging any load related issues. While not using $H_2$ will affect the overall detection rates, the 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 running the online test, one must establish the Flash filters. To do so, the Flash file download patterns of the university network were investigated by monitoring the network for a three day period in July 2015. Flash file hashes, piecewise hashes (described in section 5.2.2), and requested URLs were collected.

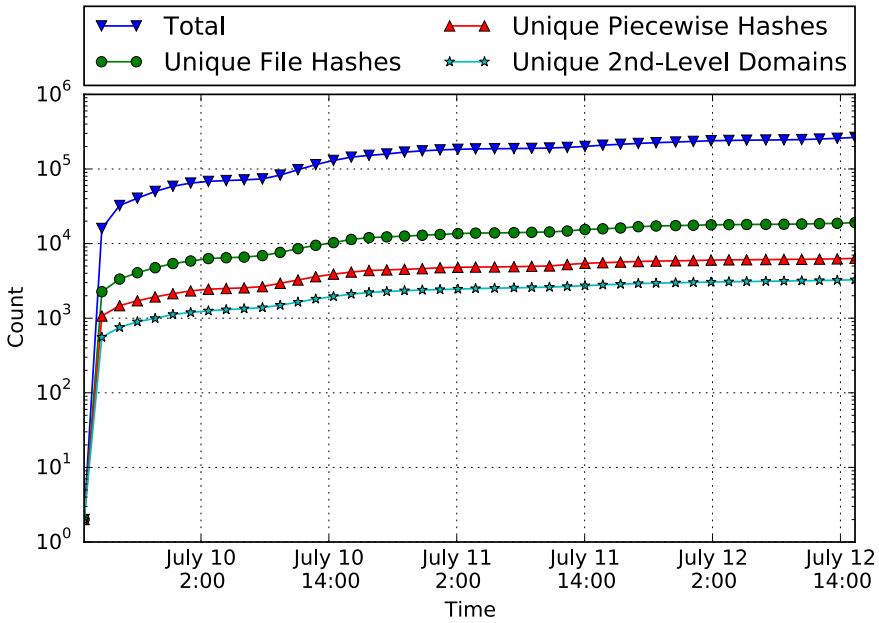


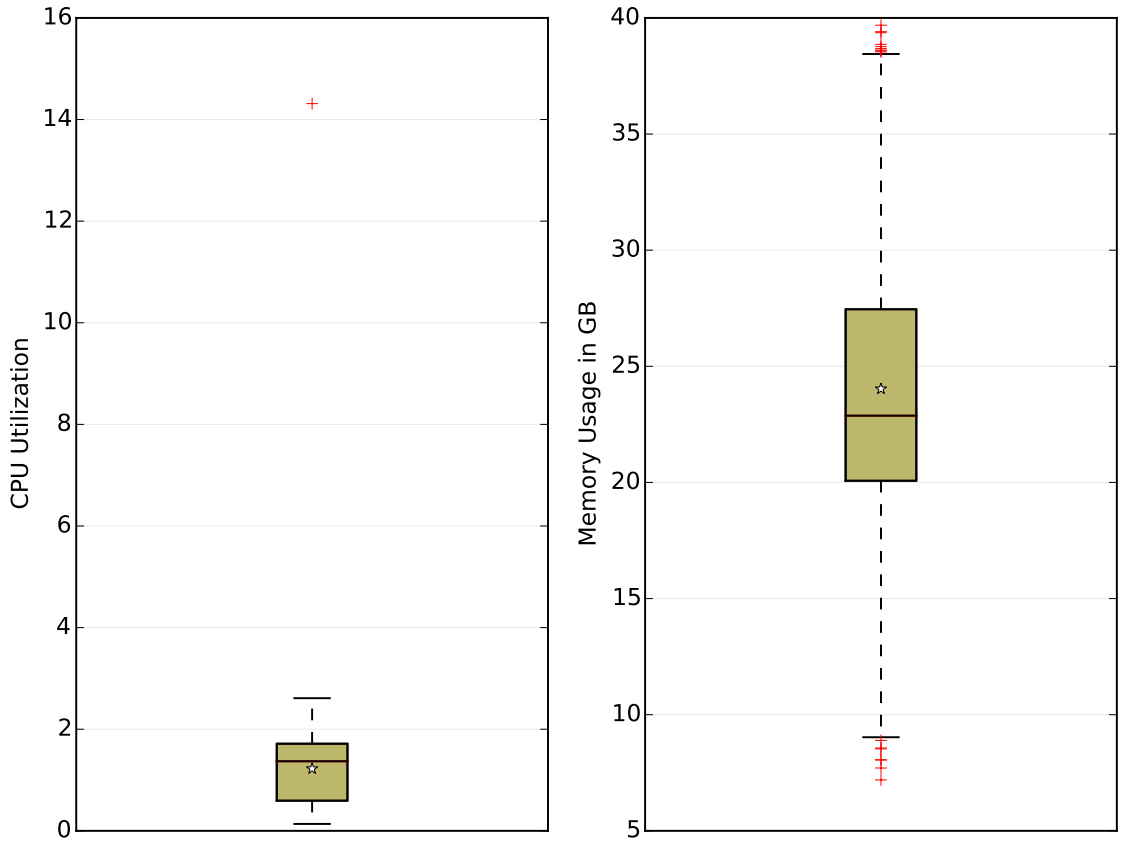

Figure 5.3: The number of unique 2nd-level domain names, Flash files, and Piecewise hashes seen on the network.

Over 270,000 Flash files were downloaded by network clients, as shown in Figure 5.3. The ad-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 5.3, 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 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, four or fewer new files are seen, while in 57% of the minutes no new files appear at all.

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

Figures 5.4a and 5.4b 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 5.4a, 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, a system with many cores is recommended to best support the semantic caching and triggering module. Memory usage (Figure 5.4b) averages 22 GBs but can reach peaks of 40 GBs, suggesting that the caching model significantly reduces memory requirements over time.



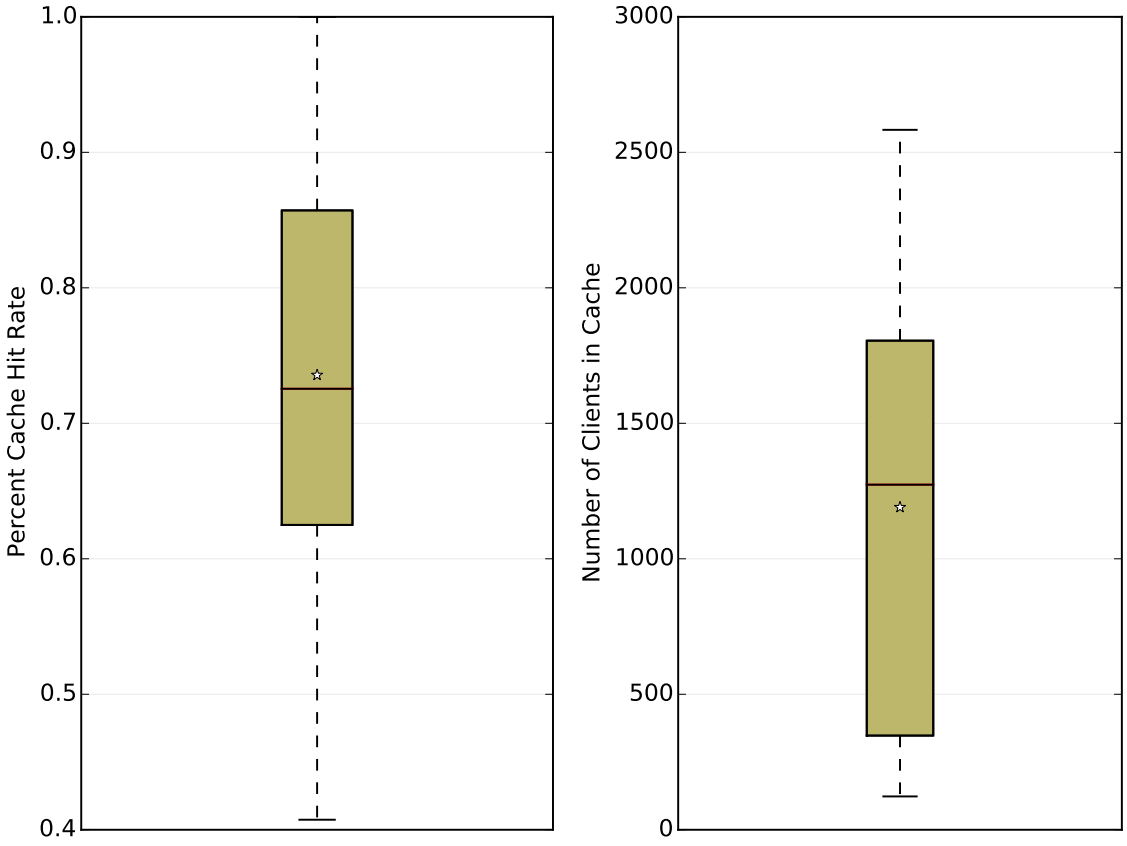

(a) Average CPU usage per minute.

(b) Average memory usage per minute.

Figure 5.4: 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 5.5a 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. It is estimated that most of these cache misses are due to the TCP reassembly errors created from the collection infrastructure issues. Another issue is that 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. These cache misses are mitigated 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 5.5b), while 90% of clients have less than 1,000 web objects in their cache at eviction. As a result, setting the client LRU cache to size N=5,000 per client maintains a reasonable memory footprint.




(a) Average cache hit rate per minute.

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

Figure 5.5: Two-level cache statistics.

Table 5.4 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 objects requiring credentials cannot be analyzed.

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

Table 5.4: 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 5.6 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 one could increase the time windows to help identify the corresponding roots, 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. The decision to not increase the window size is also tied to memory consumption. The 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. Note that trigger files that are not properly reassembled are also disregarded from analysis meaning that one could miss a potentially malicious file; however, the errors in the proof-of-concept prototype originate from from the infrastructure issues and can be mitigated with a full network view.

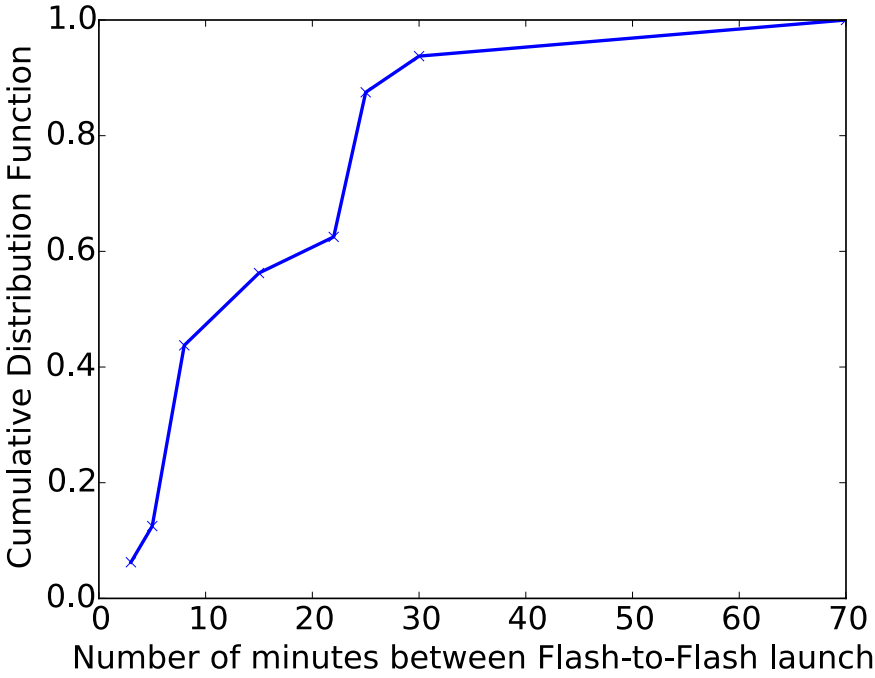



Figure 5.6: Estimate of the amount of time elapsed between Flash to Flash file launches for those files not subjected to a full honeyclient analysis.

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

| 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 5.5: List of exploits injected into the campus network and detected by the framework.

**On Detecting Malicious Flash** As part of the online evaluation, a malicious landing page was hosted 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 5.5).

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

The "victim" system runs IE10 and Firefox on a Windows 7 VM within the campus network. The victim was instrumented 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 the 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 the framework over the course of this 5 day period.

Aside from the injected metasploit malware, the approach flagged 6 malicious events, *i.e.* 1 to 2 per day. These events were missed by the 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 chapter. 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 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.

### 5.4 Case Study

This section describes a more in-depth analysis of the inner workings of the exploit kits in the empirical evaluation. Although it was originally surmised that the landing pages would likely look like advertisements, it was 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 some of the key characteristics of popular exploit kit families are described.

**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, one instance of the kit that dropped 12 binaries onto the system, while other instances launched `ping` or a command shell.

**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.

**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 5.7 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 5.7) are embedded inside HTML tags such as `p`. The JavaScript is decoded by a number of obfuscated method calls, and the resulting code is executed using an

[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);
 }
```

Figure 5.7: JavaScript snippet from Angler Exploit Kit that checks to see if files exist locally.

`eval` function call. As a result, current generation exploits must be analyzed within the larger context of the website.

## 5.5 Limitations

Many of the evasion techniques used against the framework are inherent to honeyclients in general and are being actively researched in the security community. For example, as shown in the use case, exploits will often check for evidence that the environment is a virtual machine. In the short term, one 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. (2011,

2014) 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 page before the delay has transpired. One way to combat such delays is by instrumenting the 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 the approach (Lindorfer et al., 2011).

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 the framework, 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 (Stringhini et al., 2013)) while instrumenting 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. The 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.

As discussed in §5.3, 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

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

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.

## 5.6 Discussion and Lessons Learned

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

**Key Take-Aways**

1. Current machine learning and signatured-based approaches to detecting drive-by downloads make the implicit assumption that new malicious sites seen on the Internet will look similar to something we have already seen. Unfortunately, given that a motivated human is the adversary, such an assumption is not valid, especially in targeted attacks where the attacker only needs to break into a single network and can use a one-off attack to do so. Signatures are still highly valuable for protecting or networks; however, we need to fix the signature generation model. Currently, large companies with clouds of honeyclients generate signatures and blacklists in an offline approach. We need approaches that generate signatures at the network edge while intrusions are occurring so that signatures can be sent to other networks to protect them. If we continue to treat intrusion detection as a purely data mining or machine learning problem, we will always remain several steps behind the attacker.

2. One of the ways the defender can be less reactive to the attacker is to utilize behavioral techniques for determining maliciousness such as the honeyclient-on-the-wire presented in this Chapter. Behavioral-based features describe how a website behaves when interacting with a client. These features are more powerful because they are more distinctive, and harder to change than appearance based features such as those presented in Chapter 4. Given that a smart attacker creates malicious websites, we must

continually be challenging our assumptions in what constitutes malicious behavior and propose new ways to break our own detectors.

3. In bringing honeyclients to the wire, the three key challenges are scalability, conducting a full scale analysis with context, and client/server impersonation. We can improve scalability using a combination of filtering and computing resources. It is pertinent to note that any biases placed in the filter will affect the utility of the honeyclient as a whole. For example, filters based on machine learning (Provos et al., 2008, 2007; Canali et al., 2011) bias the honeyclient to only analyze websites that "look" malicious. Given that such filters have high false negative rates due to the assumption that new malicious sites look similar to ones already seen, whitelisting approaches are a better option.

# CHAPTER 6: DETECTING BOTS USING SEQUENTIAL HYPOTHESIS TESTING

The Ponemon Institute reported that the average time taken for a company to identify an attack was 256 days after initial infection (Ponemon Institute, 2015). Indeed, there is no foolproof solution to block all attacks and the reality is that networks will be infected. Worse yet, there are few telltale signs that a machine is infected from a network vantage point. One such sign is the type of DNS traffic emanating from a machine. DNS can act as a bell-weather to the health of the network and by monitoring DNS traffic, one can uncover attacks. For example, Paxson et al. (2013) created a technique to identify DNS-based data exfiltration attacks using lossless compression. The Internet Engineering Task Force proposed DNSSEC (Arends et al., 2005) in order to digitally sign DNS messages and identify DNS cache poisoning attacks (Kaminsky, 2008; Son and Shmatikov, 2010), whereby the attacker makes a DNS cache resolver cache the wrong IP address for a domain name entry. Finally, Antonakakis et al. (2010) describes a machine learning approach to identify infected machines that use DNS to locate the remote attacker machine (called a command-and-control server) to setup the covert channel without being IP blocked by a blacklist.

This chapter focuses on the latter problem, highlighting a growing abuse of enterprise name servers whereby infected clients use automated domain-name generation algorithms to bypass defenses. The chapter investigates how to use the contextual information from per client DNS data to detect bots that use automated domain-name generation algorithms to locate their contact command-and-control infrastructure for updates and relaying information. As described in Chapter 2, an automated domain-name generation algorithm is designed to generate thousands of random domain-names given a globally accessible key. Only a few of the domain names will actually be registered, and contain IP addresses of command-and-control servers. The defender does not know which domain names will be registered, and therefore cannot block the bot without a high cost.

Even more problematic for defenders, algorithmically generated domain names (AGD) are now also used for legitimate purposes. For instance, content distribution networks (CDNs) use such techniques to provide identifiers for short-lived objects within their networks, or to perform latency experiments (Dilley et al.,

2002). Additionally, services like Spamhaus and Senderbase regularly use algorithmically generated domain names to query DNS blacklist information. The security community has largely dismissed the prevalence of these legitimate uses of such domain names, and in doing so, overlooked their effect on the ability to detect malfeasance based solely on information gleaned from a domain name. Given that most methods to detect malicious algorithmically generated domain names leverage techniques that compare distributions of domain name features extracted from benign and malicious domains, algorithmically generated domain names used in benign applications can have a large impact on the accuracy of these techniques.

More specifically, this chapter explores techniques for identifying infected clients on an enterprise network and focus on their operational impact in terms of accuracy, timeliness of detection, and scalability to large networks. First, the efficacy of existing botnet detection techniques that rely solely on the structure of the domain name as a distinguishing feature in malware identification are explored. The techniques suggested in recently proposed detection mechanisms (e.g., (Yadav et al., 2010; Yadav and Reddy, 2011)) are implemented and evaluated on traces collected at a large campus network. The impact of the rise of benign applications (e.g., for performance testing in Web browsers and for location-based services prevalent in CDNs) has on these detection techniques is also evaluated. The application of state-of-the-art detection techniques lead to high false positive rates, even when classifiers are enhanced with a combination of smoothing and whitelisting strategies. Moreover, successful classification only occurs after extended observation periods—which directly impacts the practical utility of these approaches.

To address these shortcomings, the chapter proposes an approach that exploits the fact that botnets tend to generate DNS queries that elicit non-existent (NX) responses. The work leverages the fact that a noticeable side-effect of a bot's attempts to evade blacklisting is its tendency to have a wider dispersion of NX responses across DNS zones (compared to benign hosts). The technique is based on sequential hypothesis testing (popularized by Jung et al. (2004) for detecting external scanners) to classify internal client machines as benign or infected. In doing so, some key challenges are addressed, including the need to differentiate between benign and malicious DNS queries originating from the same client, and the ability to scale to high traffic loads — the proposed approach meets both of these challenges. Furthermore, one of the unique characteristics of the approach is that by focusing solely on NX traffic (and using novel filtering and domain collapsing techniques), it can achieve high accuracy on a fraction of the overall DNS traffic (e.g., 4%) which scales to larger networks. By contrast, existing approaches use all DNS traffic during analysis. In an effort

to reduce the cognitive load on a security analyst (performing a forensic analysis on the hosts flagged as suspicious), an approach to cluster the output of the detector is provided.

The rest of the chapter is organized as follows. In §6.1 explores the background of algorithmically generated domain names and discuss pertinent related work. §6.2 covers our data collection infrastructure and summarizes the data used in the evaluation. In §6.4 a detailed evaluation of existing techniques using domain name features and their shortcomings is provided. A new detection approach is proposed in §6.5, followed by a detailed evaluation on archived data in §6.6. Operational insights on the deployment of the technique are described in §6.6.3, with key take aways in §6.9.

## 6.1 Literature Review

A recent method for identifying malicious traffic is to take advantage of historical information about the domain name being requested. As DNS-based reputation systems have been more widely deployed, attackers have turned to algorithmically generated domains (with short lifetimes) to circumvent these blacklists. As this cat and mouse game has continued, more timely blacklist and reputation-based systems have emerged (e.g., (Felegyhazi et al., 2010; Bilge et al., 2011; Antonakakis et al., 2010, 2011)). Most of these proposals use features that are time-based, answer-based, or TTL-based to detect and model domains involved in malicious activity. Additionally, network-, zone-, and evidence-based features of DNS data are also used. For instance, both Antonakakis et al. (2011) and Yadav et al. (2010) take advantage of the fact that for high availability, botnets tend to map several domains to an IP address (or vice-versa). Defenders can therefore use the web of domains and IPs to uncover the underlying network infrastructure used by the botnet. Thomas and Mohaisen (2014) found that AGDs could be clustered globally by comparing the list of recursive name servers that requested the domains within a time window while Tu et al. (2015) found that bots had similar periodicity in DNS queries. Grill et al. (2015) notes that bots try to resolve more domains during a small time interval without a corresponding amount of newly visited IPs and build a statistical approach around that fact.

Other auxiliary information can also be used. Hao et al. (2011), for example, use the fact that domains are registered "just in time" before an attack. More recent work (Yadav et al., 2010; Yadav and Reddy, 2011; Antonakakis et al., 2012; Villamarn-Salomn and Brustoloni, 2008; Jiang et al., 2010; Mowbray and Hagen, 2014) focuses on the fact that bots tend to generate lookups to hundreds or thousands of random domain names when locating their command and control server. Yadav and Reddy (2011) rely on the burstiness of NX

responses as well as the entropy of domain name character distributions to classify bot clients. Antonakakis et al. (2012) use a five-step clustering approach that clusters NX domains based on client-level structural information, and then incorporates network-level information to better classify AGDs. Jiang et al. (2010) cluster failed DNS queries and attempt to identify subclusters with specific, presumably malicious, patterns. (Mowbray and Hagen, 2014) relies solely on domain name length to cluster AGDs.

Unlike the aforementioned works, the proposed approach does not rely on domain structure or clustering techniques to identify bots. Rather, this work focuses entirely on the NX traffic patterns of individual hosts. As a result, the approach is lightweight, and can accurately identify bots upon seeing far fewer unique domain names than prior work. NX traffic is used exclusively, thereby enabling realtime analysis by using only a fraction of all DNS traffic observed.

The application of sequential hypothesis tests (Wald, 1947) in security context is by no means new. Jung et al. (2004), for example, proposed a threshold random walk (TRW) algorithm to detect scanners on a network. The insight behind their approach is that external scanners are more likely to contact inactive IP addresses than benign hosts, and so a sequential hypothesis test can be used to observe success and failure events in such an environment. Each success or failure event moves a score towards one of two thresholds: one confirming the null hypothesis and another confirming the alternative hypothesis. After a number of events that are largely dictated by the TRW parameters, the score usually crosses a threshold, confirming one of the hypotheses. Similar ideas have been used to detect the propagation of worms (Jung et al., 2008; Schechter et al., 2004; Weaver et al., 2007), to identify opaque traffic (White et al., 2013), and to find node replication attacks in wireless networks (Ho et al., 2011).

### 6.2 Collection Infrastructure

To aid in the pursuit of understanding AGD-based bot communication and develop an algorithm to detect bots, DNS traffic was collected and analyzed from several name servers at UNC's campus for a week in March 2012 as well as an 8 month period from September 2015 to April 2016. The monitored name servers served as the primary name servers for the entire wireless network as well as student residences and several academic departments around campus. In 2012 the servers served approximately 76,000 internal clients on a weekday and 50,000 clients on the weekends. The analysis in 2015 was done on a subset of the campus DNS traffic that included the campus wireless networks, and a subset of the campus academic units which served

approximately 26,000 internal clients (and 170,000 external clients) on a weekday, and 21,400 clients (and 111,000 external clients) on a weekend.

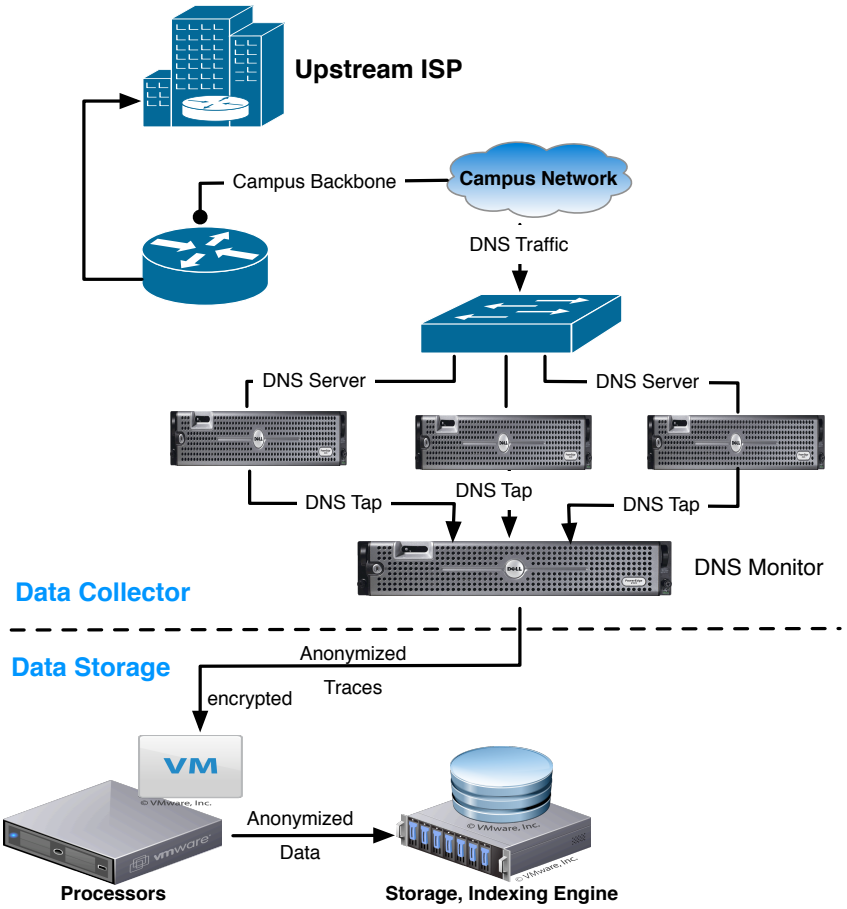



Figure 6.1: DNS Monitoring Infrastructure

## 6.3 Data Summary for Measurement Period I

The collection infrastructure (see Figure 6.1) consists of a DNS trace collector and dissector. The DNS servers monitored sits behind a load balancer, and all wireless clients using the campus network are assigned to one of these name servers during their DHCP registration. DHCP leases on this network are bound to the client's MAC address, and remain in effect for at least a few weeks. The DNS traffic from these servers is processed using a custom DNS engine. The packets in the trace are anonymized and encrypted while resident on disk.

Three consecutive days (March 18-20) were chosen for analysis for analysis. Table 6.1 summarizes some of the key statistics. The increase in traffic on March 19th corresponds to the start of the work week. Table 6.1 also shows that approximately 3% of all DNS queries result in non-existent or NX responses. A DNS server sends an NX response to a client when an entry for the domain queried by the client does not appear in the global DNS database. A mistyped domain name, for example, will lead to an NX response. Algorithmically generated domains comprise a surprisingly small amount of overall NX traffic, but they can have a large impact on the overall health of an enterprise network.

Apart from the DNS data collected on campus, a list of 2,500 known botnet AGDs was collected from publicly available blacklists. In particular, the list contains bots that are known to use DGAs for communication. Table 6.2 provides a summary of the bot families and their distribution within the blacklist. Besides the five well-known bot families represented in Table 6.2, the list was supplemented with a set of newly discovered domains. The discovered domains were found by grouping DNS responses that originated from name servers that were used by the five well-known bot families. The domains in the list are used to study features used by existing techniques to detect DGAs, as well as compare the effectiveness of these techniques to approach described in this chapter.

| | **March 18** | **March 19** | **March 20** |
|---|---|---|---|
| # of Internal DNS Clients | 49.7K | 75.4K | 77.1K |
| # of DNS Queries | 37.3M | 61.2M | 60.3M |
| # of NX response | 1.3M | 1.8M | 1.7M |
| # of distinct domains | 1.5M | 1.8M | 1.8M |
| # of distinct zones | 373.4K | 528.2K | 566.4K |
| # of distinct NX domains | 190.4K | 216.2K | 220.4K |
| # of distinct NX zones | 15.3K | 22.1K | 24.2K |

Table 6.1: DNS traffic stats for three days in March 2012.

| **Bot Family** | **# Samples** | **Sample of generated domain name** |
|---|---|---|
| `Bobax` | 1079 | `nghhezqyrfy.dynserv.com` |
| `Conficker` | 728 | `rxldjmqogsw.info` |
| `Cridex` | 389 | `frevyb-ikav.ru` |
| `Zeus` | 300 | `pzpuisexhqc69g33mzpwlyauirdqg43mvdt.biz` |
| `Flashback` | 100 | `fhnqskxxwloxl.info` |
| *Discovered* | 314 | `brmyxjyju.org` |

Table 6.2: Summary of bot samples used in the compiled blacklist.

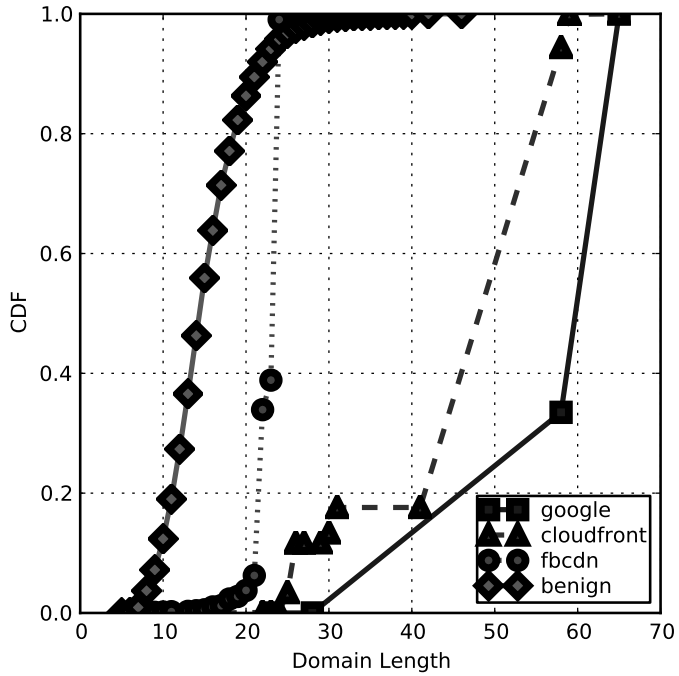



Figure 6.2: CDF of domain name lengths for benign domains

### 6.4 Classification based on Features of a Domain Name

Existing techniques focus on properties of the name in order to identify and cluster algorithmically generated domain names. For instance, Antonakakis et al. (2012) and Yadav et al. (2010) used the length of a domain name as a feature to distinguish malicious domains from benign domains. Figures 6.2 and 6.3 show the distribution of the lengths of domain names for a set of benign and malicious domains.

The benign domains shown in Figure 6.2 include domains for known CDNs and other benign domains from `alexa.com`. Notice that domain names from `alexa.com` exhibit uniformly distributed lengths between 5 and 20 characters, while CDNs exhibits longer lengths clustered around a few discrete points. The lengths of domain names used by botnet (in Figure 6.3) also cluster around a few discrete points; likely as a result of the generation processes they use. This similarity between the lengths of botnet domain names and benign CDN domain names suggests that the length of a domain name might not be a strong distinguishing feature.

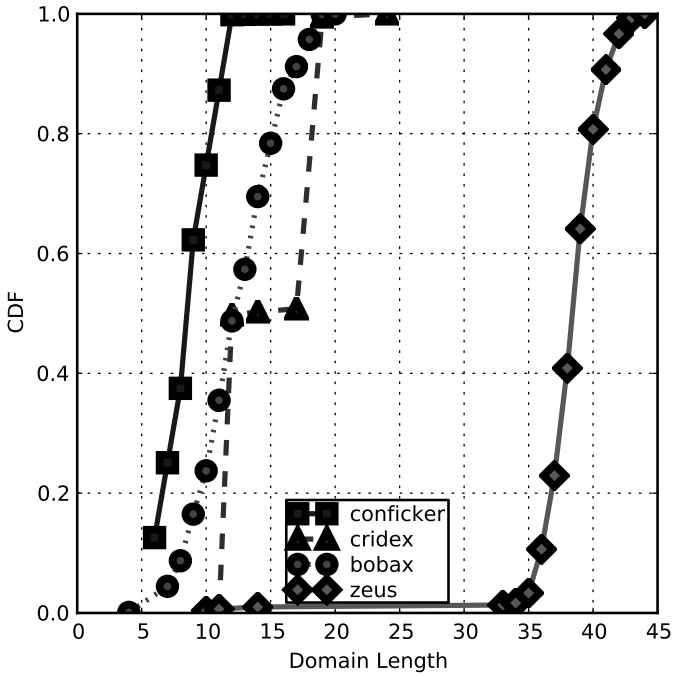



Figure 6.3: CDF of lengths for botnet-related domains

Other proposals incorporate the use of similarity metrics for detecting malicious AGDs. In what follows, three similarity metrics used in current proposals are revisited — namely Kullback-Leibler (KL) divergence, Jaccard Index (JI), and Levenshtein distance.

**KL Divergence** One approach for detecting algorithmically generated domain names is to use the Kullback-Liebler (KL) divergence to compare character frequency distributions. Kullback-Liebler divergence (Kullback and Leibler, 1951) measures the relative entropy between two probability distributions. Yadav et al. (2010), for example, use a maximum-likelihood classifier (R.O. Duda and Stork, 2007)—with KL as its distance metric—for detecting malicious AGDs. The intuition is that malicious algorithmically generated domain names have character and n-gram frequency distributions that are significantly different from character distributions derived from benign domains.

**Jaccard Index** The Jaccard Index is a similarity metric that counts the bigram occurrences in two strings and measures the amount of overlap between them. The idea is that randomized strings (or supposedly-malicious domain names) should have a set of bigrams that is different than bigrams in a normal (non-malicious) English-based string.

**Levenshtein Distance** Edit distance is a measure between two strings, which counts the number of insertions, deletions, and substitutions to transform one string to another (R.O. Duda and Stork, 2007). In the case of algorithmically generated domains, the assumption is that because a group of malicious domain names are randomly generated, their average edit distance should be higher than a group of non-malicious names.

Each of the similarity metrics operate on a group of domain names in order to achieve detection accuracy. Yadav et al. (Yadav et al., 2010; Yadav and Reddy, 2011), for example, recommend 200 to 500 domain names for best results. To create the necessary clusters for evaluation, the method suggested in (Yadav et al., 2010) is applied wherein clusters are created by mapping domain names to their corresponding server IP addresses over a specific time window. This is done because botmasters tend to register multiple domains to the same server IP address.

In order to evaluate these approaches, 42,870 domain name clusters from March 19, 2012 were analyzed and contained 13 sink-holed instances (or clusters) of the `conficker` bot. A sinkhole is a name server that redirects malicious traffic to some address under control of the defender, in order to contain the malware. Each cluster was manually inspected to ensure no other bot instances were found. The ground truth was supplemented with four clusters (each containing 300 entries) of AGDs sampled from the list of known botnets (see §6.3). Additionally, since the Kullback-Liebler and Jaccard Index based classifiers require both benign and malicious training models, the benign training model was built using the top 10,000 domains from `alexa.com` and the malicious training model using the list of 2,500 domains from the blacklist.

**Findings** Table 6.3 shows the results of using a Kullback-Leibler-based classifier, which achieved the highest accuracy in the evaluation. The classifier is able to identify the presence of all of the malicious samples, but even then, it has an exceedingly high false positive rate of 28%. A large fraction of CDN traffic is incorrectly classified as malicious, which is one factor contributing to the high false positive rate. A natural way to improve the performance of the classifier would be to whitelist popular CDNs (Yadav and Reddy, 2011). Figure 6.4 shows the result of using the different classifiers with varying domain cluster sizes and whitelisted CDNs. It was found that, even with filtering, the KL classifier achieves a 12.5% false positive rate with a cluster size of at least 200 domain names. As shown later in Section 6.4.1, such large cluster sizes have implication on detection rates, processing speeds, and accuracy.

The classifier using Jaccard's Index achieved the second highest accuracy amongst the techniques evaluated; however, as Figure 6.4 suggests, the accuracy came at a high cost—a true positive rate of 92%

| Domain Source | Daily | |
|---|---|---|
| | True Positives | False Positives |
| Bot Traffic | 1.0 | 0.28 |
| Facebook (CDN) | 0.65 | 0.35 |
| Cloudfront (CDN) | 0.36 | 0.64 |
| Amazon (CDN) | 0.72 | 0.28 |
| Google IPv6 (CDN) | 0.18 | 0.82 |

Table 6.3: Results of the KL classifier for Mar.19, 2012.

with corresponding false positive rate of 14%. Furthermore, the Jaccard-based classifier is the slowest of all techniques tested, which limits its ability to be used in an online fashion.

Figure 6.4 also shows the classification results using an edit distance approach. The plot shows the true and false positive rates when varying the edit distance threshold. In that evaluation, the edit distance values were generated for groups of botnet and benign domain names within the training sets and used that to determine a threshold value that would separate normal traffic from malicious AGDs. Interestingly, 70% of the benign groups had an average edit distance score of eight or below. Of the malicious groups, `conficker` averaged a score of eight, while `bobax` and `cridex` score between nine and eleven. `Zeus` was a consistent outlier with scores above 35.

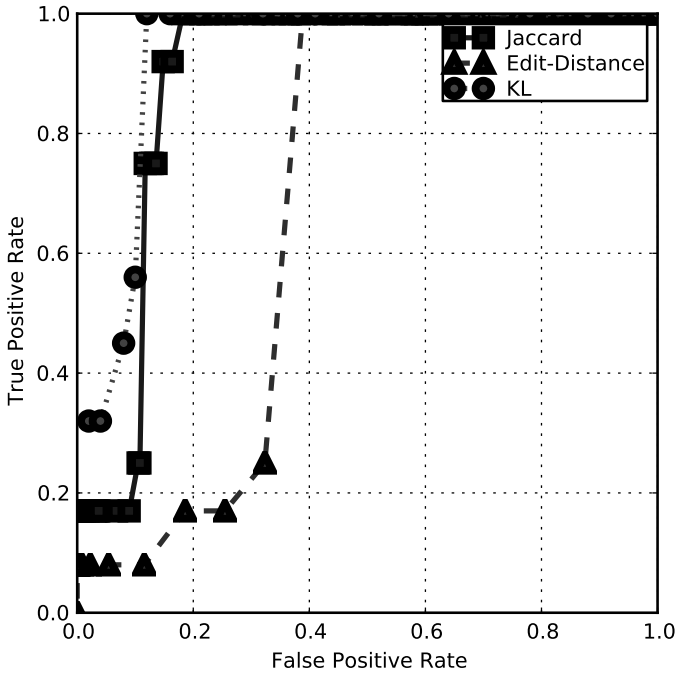



Figure 6.4: ROC Curves for Jaccard Index, Edit Distance and KL Divergence using the daily dataset and CDN filtering.

### 6.4.1 Shortcomings of Existing Methods

Overall, the application of a KL-based classifier performed reasonably well, providing classification decisions for all the domain clusters within a few minutes. The problem, however, is that it required on the order of a few hundred domain names in *each* cluster to provide accurate results. To see why this is problematic, note that it may take several hours before a cluster meets the minimum threshold required to achieve the classification results given in Table 6.3; in particular, during a one week period there were eight `conficker` instances, one `cridex` and one `spambot`. Two of the `conficker` instances queried less than 200 randomly generated domain names, while the other six instances took almost three hours to query 100 domain names, and 3.5 days to query 500 domain names. The `cridex` and `spambot` instances generated less than 10 domain name lookups during 3.5 days. This rate of activity requires many days of monitoring before classification can occur, rendering the technique unusable for detecting and blocking malicious activity from these sources.

From an operational perspective, the Jaccard Index approach is appealing because of its ease of implementation and reasonable performance. The simplicity, however, comes at the cost of computation time: it took several hours to classify all the domain clusters in just one day's worth of DNS traffic. Another disadvantage is the fact that the approach is highly sensitive to the training dataset and the number of domain names in the cluster being evaluated.

Methods based on edit distance, on the other hand, have the advantage of not requiring training data and can operate on small clusters of names. That said, the edit distance approach was the least effective of the techniques evaluated. Its high false positive rates are tightly coupled with the difficulty of selecting an appropriate threshold value. For real-world deployments, the need to constantly monitor and fine tune these thresholds significantly diminishes its practical utility. This technique was also extremely slow, taking several hours to process the dataset.

The analyses indicate that the examined approaches are not robust enough to be used in production environments. This is particularly true if additional auxiliary information (e.g., realtime reputation information from various network vantage points in the DNS hierarchy (Antonakakis et al., 2012)) is not being used to help address real-world issues that arise when dealing with the complexities of network traffic—where friend or foe can be easily confused. These techniques all make the fallacious assumption that anomalous

behavior equates to malicious activity and so the use of algorithmically generated names for benign purposes undermines this assumption.

## 6.5 Approach

To address the accuracy and performance issues inherent in the aforementioned approaches, a lightweight algorithm is presented and based on sequential hypothesis testing, which examines traffic patterns rather than properties of a domain name in order to classify clients. The intuition behind the approach is that a compromised host tends to "scan" the DNS namespace looking for a valid command and control server. In doing so, it generates a relatively high number of unique second-level domains that elicit more NX responses than a benign host. As a result, the problem lends itself to using sequential hypothesis testing (Wald, 1947) to classify clients as bots based on online observations of unique NX responses.

The general idea is illustrated in Figure 6.5. In Step ❶, the amount of data analyze is reduced by over 90%, retaining only NX response packets. Next, the client IP address and zone of the domain name are extracted from each packet (Step ❷) and then NX responses filtered for well-known (benign) domain names (Step ❸). The zone information of the remaining domain names are used to adjust the client's score. The score is adjusted up or down based on whether the client has seen the zone before (Step ❹). Finally, the new score is compared to both a benign threshold and a bot threshold. If either threshold is crossed, then the client is classified; otherwise, the client remains in the pending state waiting for another NX response (Step ❺).

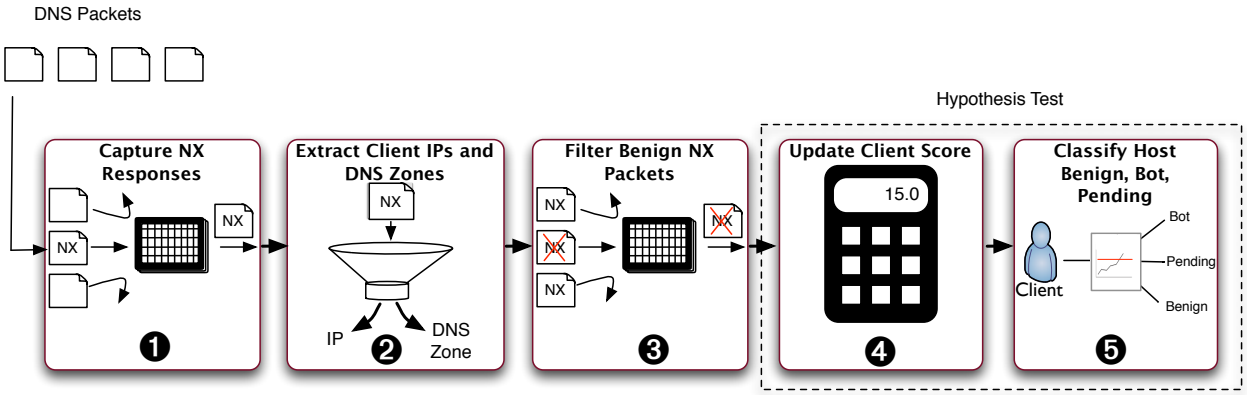



Figure 6.5: High-level overview of the workflow.

The goal is to accurately classify a host as a bot or benign while observing as few outcomes as possible. To that end, the problem is approached by considering two competing hypotheses, defined as follows:

**Null hypothesis** $H_0$ = the local client $l$ is benign.

**Alternative hypothesis** $H_1$ = the local client $l$ is a bot.

A sequential hypothesis test observes success and failure outcomes ($Y_i, i = 1, ...n$) in sequence and updates a test score after each outcome. A success pushes the score for client $l$ towards a benign threshold while a failure pushes the score towards a bot threshold. In the current context, a success and failure outcome are described as follows:

**Success** $Y_i = 0$ Client $l$ receives an NX response for a DNS zone it has already seen.

**Failure** $Y_i = 1$ Client $l$ receives an NX response for a unique DNS zone.

For simplicity, a DNS zone is considered as any portion of the DNS namespace that is administered by a single entity (e.g., the `google.com` zone is administered by Google).

The size of the step taken towards the thresholds is decided by the values $\theta_0$ and $\theta_1$. The value of $\theta_0$ is defined as the probability that a benign host generates a successful event while $\theta_1$ is the probability that a malicious host generates a successful event. More formally, $\theta_0$ and $\theta_1$ are defined as:

$$\begin{aligned} P_r[Y_i = 0|H_0] = \theta_0, P_r[Y_i = 1|H_0] = 1 - \theta_0 \\ P_r[Y_i = 0|H_1] = \theta_1, P_r[Y_i = 1|H_1] = 1 - \theta_1 \end{aligned} \tag{6.1}$$

Using the distribution of the Bernoulli random variable, the sequential hypothesis score (or likelihood ratio) is calculated as follows:

$$\Lambda(Y) = \frac{Pr[Y|H_1]}{Pr[Y|H_0]} = \prod_{i=1}^{n} \frac{Pr[Y_i|H_1]}{Pr[Y_i|H_0]} \tag{6.2}$$

where $Y$ is the vector of events observed and $Pr[Y|H_i]$ represents the probability mass function of event stream $Y$ given $H_i$ is true. The score is then compared to an upper threshold ($\eta_1$) and a lower threshold, ($\eta_0$). If $\Lambda(Y) \leq \eta_0$ then accept $H_0$ (i.e., the host is benign) , and if $\Lambda(Y) \geq \eta_1$ accept $H_1$ (i.e., the host is malicious). If $\eta_0 < \Lambda(Y) < \eta_1$ then no decision is made (i.e., pending state) and one must wait for another observation.

The thresholds are calculated based on user selected values $\alpha$ and $\beta$ which represent the desired false positive and true positive rates respectively. The parameters are typically set to $\alpha = 0.01$ and $\beta = 0.99$. The upper bound threshold is calculated as:

$$\eta_1 = \frac{\beta}{\alpha} \tag{6.3}$$

while the lower bound is computed as:

$$\eta_0 = \frac{1-\beta}{1-\alpha} \tag{6.4}$$

A key challenge in this setting is that because internal hosts are monitored, all client-side DNS traffic is visible, including the benign queries (e.g., from web browsing sessions) as well as the malicious queries of the bot. However, since the benign activities mostly result in successful DNS responses, one can safely filter such traffic and focus on NX responses (where the bot has more of an impact). This strategy has the side effect of discarding the vast majority of DNS packets, thereby allowing operation at higher network speeds. The traffic is further filtered by only processing second level DNS zones, rather than fully qualified domain names (FQDNs). One can focus on second-level domains since most bots generate randomized second-level domains in order make it more difficult to blacklist them and to hamper take-down efforts.

One can also take advantage of the fact that NX traffic access patterns for benign hosts follows a Zipf's distribution. Indeed, over 90% of NX responses in the data are to 100 unique zones. The bot DNS queries lie in the tail of the Zipf curve, hidden by the vast amounts of benign traffic. To quickly sift through this mountain of data, a Zipf filter is applied comprising the most popular zones[1] and matches are removed using a perfect hash. Finally, each time a client is declared benign its state is reset, forcing it to continuously re-prove itself.

## 6.6 Evaluation - Measurement Period I

Unlike the approaches (Yadav et al., 2010; Yadav and Reddy, 2011; Antonakakis et al., 2012) discussed earlier, the current approach classifies client IPs based on NX traffic patterns. As such, ground truth in this case is a list of clients exhibiting botnet-like behavior. To attain ground truth for the analyses that follow,

[1] In the empirical evaluations, the top 100 zones are used

any hosts that did not receive NX responses were excluded, and then any connections that received NX responses from white-listed NX zones (e.g., senderbase.org) were discarded. The white-list was created by manually inspecting the top 100 zones of domain names that elicit NX responses[2]. The domain names were then cross-referenced from the remaining clients against well-known blacklists. While this approach was helpful in identifying known bots, it clearly is of little help in identifying new bots that were yet discovered in the wild on the date of the analysis. To address this possibility, two techniques were applied. First, lookups were performed on domains that received NX responses in March to see if any of those domains were now sink-holed. And second, the remaining clients were hand-labeled on whether they had similar name structure as existing AGDs, generated a sequence of at least two or more domains names that followed a similar structural convention (e.g., character set and length of the domain name), and received NX responses. In the end, a total of 255 clients were found: 66 clients on March 18th, 101 on the 19th and 88 on the 20th.

**On Parameter Selection** Both $\theta_0$ (the probability that a *benign* host sees a success event) and $\theta_1$ (the probability that a *malicious* host sees a success event) are parameters that must be set appropriately in any real-world deployment. Therefore, they must be calculated for each deployment of the sequential hypothesis framework. These parameters can be robustly computed from a relatively small amount of traffic. Recall that in §6.5, a successful outcome was defined as one where a host receives NX responses for a zone it has already contacted at least once in the past, and a failure outcome every time a NX response is generated for a zone not seen previously. To estimate these parameters, one can simply track NX responses on a per-client basis for a set window of time, counting successes and failures.

From the empirical analyses, the majority of DNS traffic is in fact benign, and the AGD traffic comprises less than $2\%$ of the overall traffic. This should be true within most enterprise networks, and, as a result, $\theta_0$ is calculated by simply computing the percent of successful connections for all NX traffic observed in that window of time. In our analysis, $\theta_0 = 0.9$.

Estimating $\theta_1$, on the other hand, is more difficult. If an operator is fortunate enough to have an oracle by which she could separate benign from malicious hosts and build ground truth for her network, then she could infer $\theta_1$ by computing the percent of successes generated by malicious hosts; however, in the real world, access to such an oracle is difficult, if not impossible; hence, $\theta_1$ must be estimated by other means. In this work, it was found that by discarding all clients that generate less than $\delta$ failure events, one can achieve a

[2] The stability of the white-list was confirmed using historical NX traffic from within the network spanning several months.

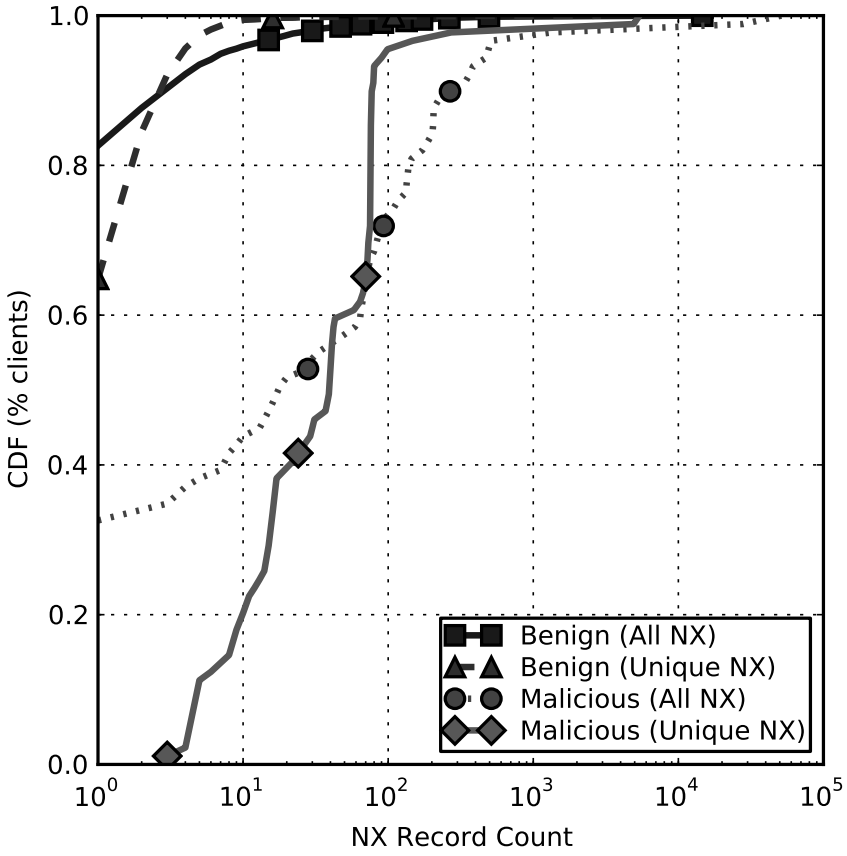



Figure 6.6: NX zone counts for benign and malicious clients.

reasonable approximation of $\theta_1$ from the remaining traffic. This is based on the fact that bots tend to generate far more failure events than benign hosts.

Figure 6.6 offers insight into why the application of sequential hypothesis testing makes sense. Notice that ninety-five percent of benign hosts receive NX responses for four or less unique zones, while 98% of bots receive NX responses for four or more hosts over a day. By monitoring only NX traffic, there is a clear delineation between benign and infected hosts. Based on this observation, $\delta = 4$ for the approximation of $\theta_1$ within the network, and $\theta_1 = 0.6$ for our analysis.

### 6.6.1 Offline Analysis

In order to evaluate the accuracy of the classifier, a $k$-fold cross-validation was used. Cross-validation is a method typically used to assess the performance of a classifier by partitioning data into $k$-subsets. One subset is used for training, while the others are used for testing. This process is repeated $k - 1$ times until

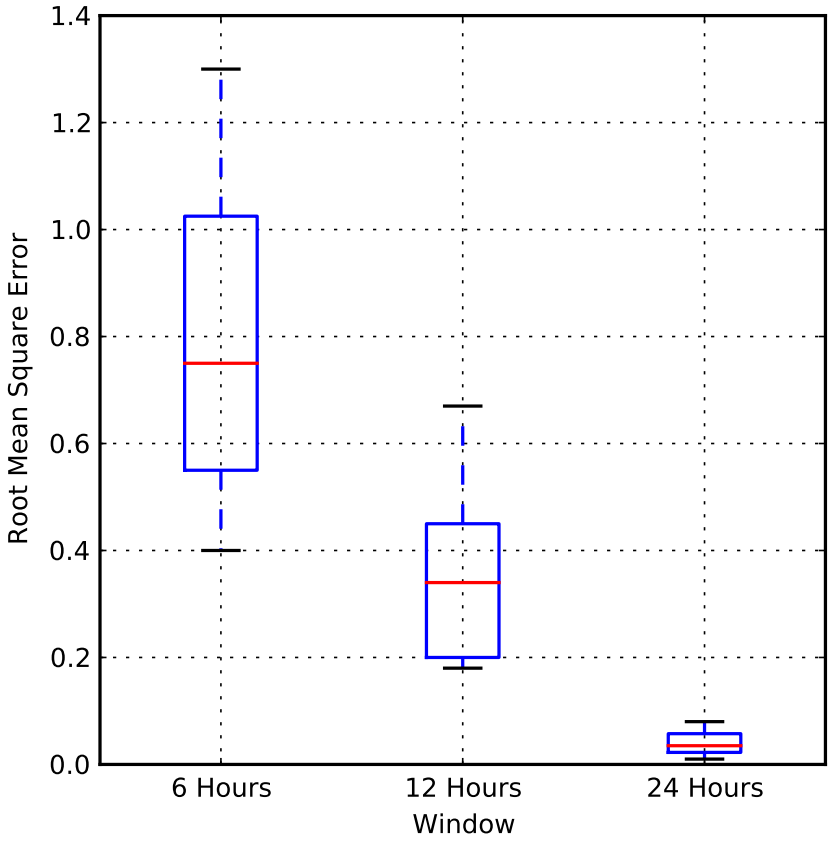



Figure 6.7: Box-and-whisker plot of the error estimation for k-fold cross-validation for varying training window sizes

each of the $k$ subsets has served as a training set. In the results that follow, we estimated $\theta_0$ and $\theta_1$ based on the designated training set, then fixed these values on the testing data.

A set of experiments were performed to estimate an appropriate training window size. $\Delta = 6$, 12, and 24-hour intervals were chosen as window-size candidates, dividing the dataset by each. The ground-truth was split data based on the clients observed within those time windows. Similarly, $\theta_0$ and $\theta_1$ were estimated for each of the time windows using the technique discussed earlier. A $k$-fold cross-validation was run for each of the intervals (where $k = 10, 5, 3$) and the prediction errors between them were compared.

Figure 6.7 shows the results of each experiment. The prediction errors are computed as the root mean square error over two repeated runs and plotted as a Box-and-Whisker plot to show the mean and variance within each experiment. Experiments indicate that a training window of $\Delta = 24$ hours yields the best results with an average root mean square error of 0.034. The accuracy of the classifier is given in Table 6.4.

| $k$-fold validation | **Window Size ($\Delta$)** | **TP** | **FP** |
|---|---|---|---|
| $k = 3$ | 24 hours | .94 | .002 |
| $k = 5$ | 12 hours | .86 | .031 |
| $k = 10$ | 6 hours | .81 | .048 |

Table 6.4: Accuracy for $k$-fold cross validation experiments for varying training window sizes ($\Delta$).

A window size of 24 hours provides the best results, because it takes into consideration the diurnal patterns in network traffic; therefore, the remainder of the experiments use 3-fold cross-validation.

**On Classification Speed** One of the major drawbacks of existing approaches is the amount of time that elapses before a host can be classified (see §6.4). Although there is no definitive information on exactly when a client is infected, infection time is approximated as the moment of the first unique NX response for a particular client. It was found that, on average, the sequential hypothesis technique detects bots within three to four unique NX responses (with a maximum of nine).

Figure 6.8 shows the time (in seconds) taken to classify a client as a bot. The majority of bots are correctly classified within a few seconds of seeing the first unique NX response—primarily because they perform tens of queries at once. Some bots, however, take a more delayed approach, making singular queries at uniform time intervals. In this case, it can take several hours to detect them.

That said, since bots must receive instructions from a command-and-control server, a more appropriate measure might be to compute the time elapsed before the bot successfully connects with its command center — called the "rendezvous point." One desires the ability to detect the bot before it makes that connection.

To perform such analyses, a random sample of 20 prominent bots was chosen from each of the three days and located their rendezvous point by hand. Figure 6.9 shows the difference between the rendezvous time and classification time. In 10 [of 60] cases, the rendezvous takes place before the bot is detected.In 16 cases, the approach detected the bot at the same time as the rendezvous point, while in the remaining cases, the host was declared as a bot seconds before the actual contact with the command-and-control server was made. In 83% of the cases, bots are detected either shortly before or during the liaison with their command-and-control servers. The differences in detection time from the 19th to the 20th are due to a large AGD-based compromise that occurred on campus on the 20th. The event was detected by the approach and the results were shared with campus network operators. Unfortunately, operators have not provided feedback on the operational value of the supplied results.

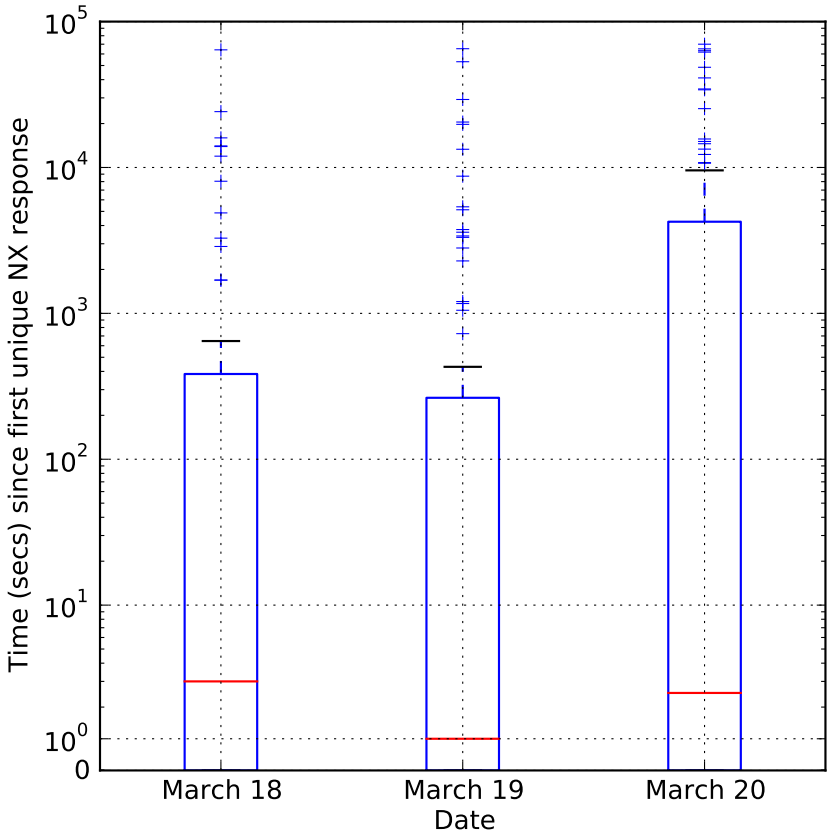



Figure 6.8: Classification time after first unique NX response.

**On Hosts Pending Classification** The fraction of clients remaining in the pending state at the end of a given time window is analyzed. At the end of each day (i.e., $k = 3$), 10% of the hosts were in the pending state. Of those clients, 70% had a response from one (unique) NX zone, 90% two or less, and 99% four or less. All but one of the 18 bots (from the ground truth) that had not been classified by the sequential hypothesis test (6 [of 66] on the 18th, 10 [of 101] on the 19th, and 2 [of 88] on the 20th), were in the pending state. These 18 clients had generated, on average, two or less unique NX responses in the allocated time window.

Upon closer inspection it was found that 95% of the pending hosts were in that state for at least 2 1/2 hours and some for almost the entire 24-hour period. This implies that as the pending hosts age, strategies are required for removing these hosts from the pending list in order to reduce the memory footprint. One strategy is to use an approach similar to our Zipf Filter, and generate a filter based on the top $n$ unique zones in the pending host list. With a cursory analysis using the top 100 pending zones, 30% of the hosts in the pending

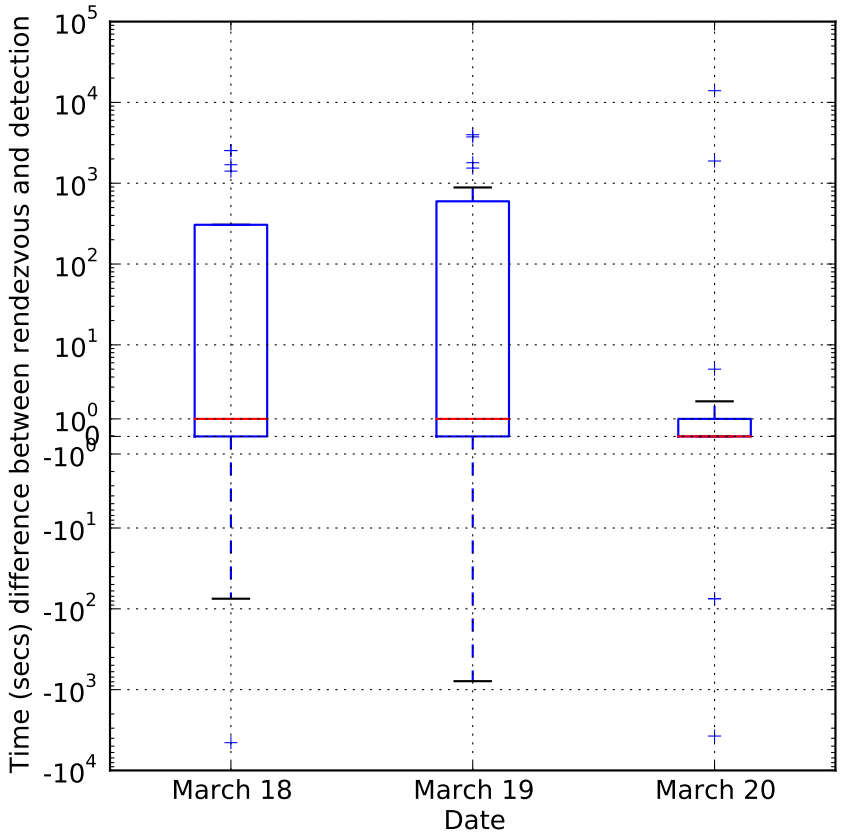



Figure 6.9: Time between classification and rendezvous.

state were removed. Another option is to randomly prune a certain percentage of the pending hosts based on their age or their unique NX response count. Such extensions are left as future work.

**Comparison to Existing Work:** To perform a direct comparison to approaches that make use of NX traffic, an approximation of the time binning algorithm of Yadav and Reddy (2011) was implemented. The work extends the Edit-Distance technique (see §6.4) to individual clients by exploiting the fact that bots tend to make queries in bursts. Their assumption is that by incorporating NX responses, domain samples can be gathered quicker than with successful DNS queries alone.

The prerequisite clusters were created by collecting all queries that elicited an NX response within 64 seconds (before and after) of a successful rendezvous query for each client (Yadav and Reddy, 2011). The edit distance measure is then applied to the clusters, and the average edit distance value for each cluster is compared with a threshold to determine whether the cluster is malicious or not. Clusters were built for

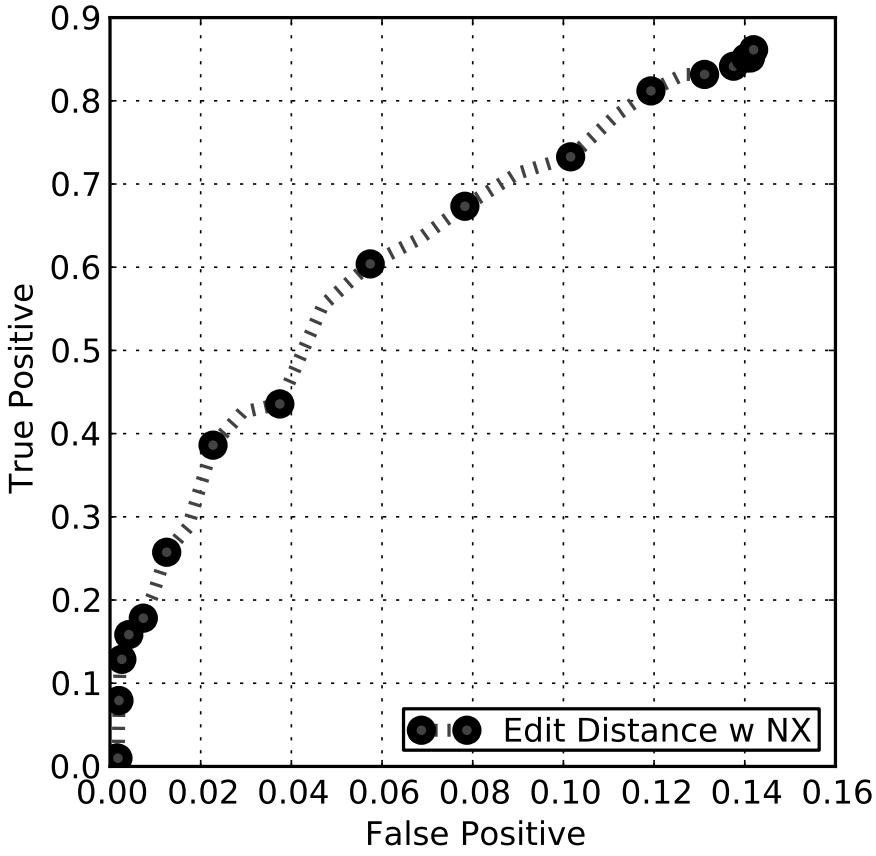



Figure 6.10: ROC curve for edit distance using NX responses.

each potential rendezvous point in the March 19th dataset. Clusters that contacted well-known white-listed domains (e.g., `facebook.com`) were filtered using the 100,000 most popular domain names (41,758 zones) from the March dataset. This left 455,500 domain name clusters spanning 10,758 unique client IP addresses.

Figure 6.10 shows the true and false positive rates when adjusting the edit distance threshold value. As with the other edit distance approaches (see §6.4), this extension also resulted in a high false positive rate (of over 14%). Even with the extra domains collected from the NX traffic, at most 80 AGDs per cluster were collected—far below the 200 domain names required for accuracy (Yadav and Reddy, 2011). Only 17 of the clients had clusters with more than 50 domain names. An additional limitation is that Yadav and Reddy (2011)'s approach requires storage of both successful and NX domain names, which adversely affects its runtime performance. By contrast, the hypothesis testing approach stores only the DNS zones for each client, and only require updating a hypothesis test score for each observed event.

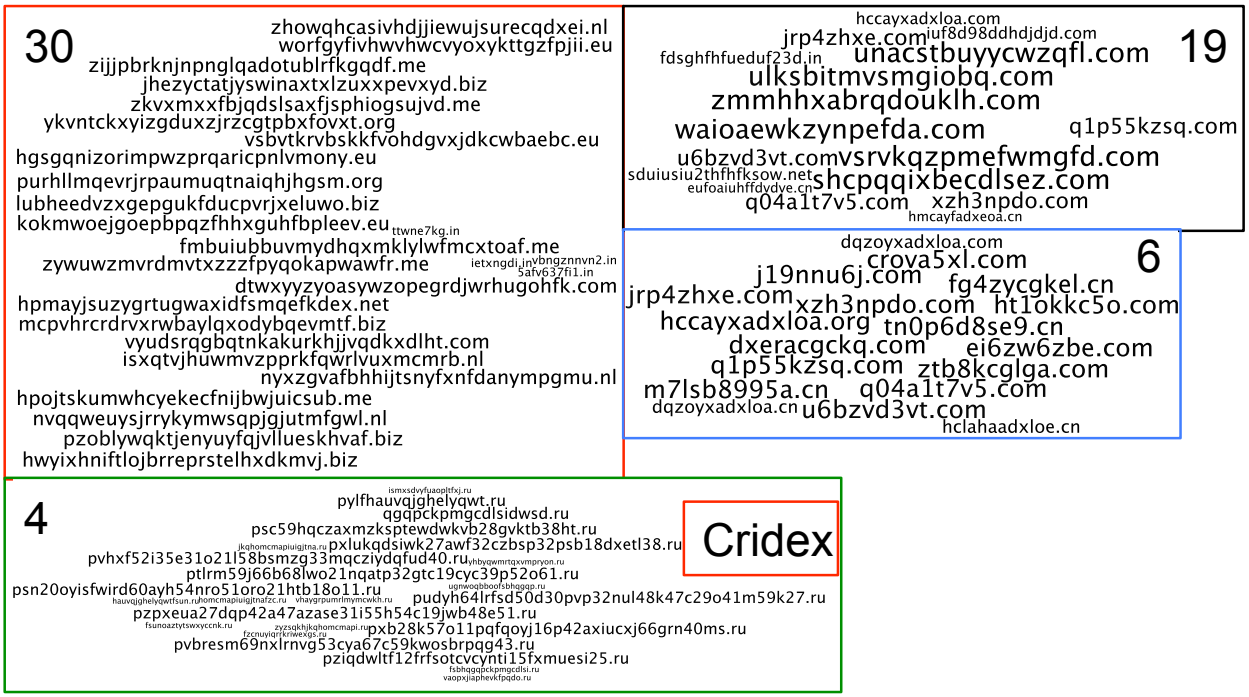



Figure 6.11: AGDs found with hierarchical clustering technique. Size of domain name indicates its prevalence in the cluster. Count indicates number of clients found in cluster.

### 6.6.2 Visualizing AGD Traffic

In an enterprise setting, a security analyst usually must investigate the list of hosts declared as bots by any of the aforementioned techniques. After the detection process has completed, and to help reduce the cognitive load on the analyst, a technique for grouping clients based on their AGD traffic is provided. The technique capitalizes on observations made while investigating the output of the algorithm, namely that (1) multiple clients tend to be infected with the same type of bot, and (2) the infected hosts generate the same domain lookups because of the use of a global seed.

These observations lend themselves to a natural grouping procedure for a set $S$, where $S$ denotes the clients declared as bots during some time window:

- $\forall i \in S$, let $S_i$ be the tuple $\langle l, n_0, n_1 ... n_m \rangle$ where $l$ is the client's IP, and $n_0, ... n_m$ the list of NX zones queried.
- Let $G = \cup\, n_0, \ldots n_N \in S$.
- For each client $l$, let $b_l$ be a bitmap of length $N$ representing the zones in $G$ and set the bits to 1 for the domains that the client has queried.

- Let the distance between two clients $l_1$ and $l_2$ be $distance(l_1, l_2) = \frac{1}{B_{l1,l2}}$, where $B_{l1,l2}$ is the sum of the number of bits set of the resulting ANDed bitmaps.

- Set $S$ is clustered using hierarchical clustering (Everitt et al., 2011).

Using this approach, the data for March 20th was clustered. The 747 clients were grouped creating 23 clusters of two or more clients. Of those clusters, four contain 59 of the 88 bots found in the ground truth. Figure 6.11 shows a sampling of the AGDs generated by the clients in each cluster. AGDs in the largest fonts are ones that appear in all clients in the cluster. The smaller the font, the less appearances the domain made.

To attain more information about the botnet families for these clusters, publicly available blacklists and anti-virus websites were searched for information on the domains. Lookups on the domains were performed (e.g., using `dig`) to see if they were sink-holed. Three of the four clusters were sink-holed, and the fourth had known `cridex` AGDs (e.g., `aecgrgbjgaofrilwyg.ru`).

The remaining 29 bots (in the ground truth) did not cluster. 18 of those hosts generated similarly structured domains, but no two hosts generated the exact domains (see Table 6.5). Little information was found on the origins of these domains. Another 3 clients contain multiple domains that are sink-holed to an address linked to the `TDSS` botnet (Golovanov and Soumenkov, 2011).

| IPs | Example AGDs |
|---|---|
| *IP 1* | `kt2syggf436dtag458.com` |
| *IP 2* | `kt2syggf436dtag182.com` |
| *IP 1* | `jhbvyvuyvuyvuvujvuvrf6r66.com` |
| *IP 2* | `bbgyujh6uh7i5y67567y5b7.com` |
| *IP 3* | `csfsdfvdbdbbfbnmcnq8858.com` |
| *IP 1* | `27613082671222563732.com` |
| *IP 2* | `79735931367645588627.com` |
| *IP 3* | `13348318318656728693.com` |
| *IP 1* | `e7722746d7c642c2a6793cb8935c45da.com` |
| *IP 2* | `80b8024c08484f029d1c229f5030c741.com` |
| *IP 3* | `c62fb768db0c4d179bfb200fcc415c9f.com` |

Table 6.5: AGDs that clustered by domain length.

### 6.6.3 Analysis of Live Traffic

To further demonstrate the utility of the technique, an online version was implemented and deployed it on the campus network. For the live test, an Endace 9.2X2 Data Acquisition and Generation (DAG) card connected to a host machine was used. This setup was used to monitor DNS traffic at the border of the

campus network. The DAG captures DNS packets at line rates and places them in a shared memory buffer without relying on the host. As a result, one can take full advantage of the host (a 2.53 Ghz Intel Xeon core processor with 16GB memory) for packet inspection. As DNS packets are placed into the shared memory buffer by the DAG card, they are assigned to an available core to perform the initial dissection. If the packet requires further processing, it is passed from core to core in a pipeline, where each core is assigned a specific task. This design easily scales by dynamically assigning packets and tasks across multiple cores.

As Sommer et al. (2009) note, utilizing multi-core architectures to provide parallelism is important in order to be able to provide online network analysis at line speeds. To that end, the network capture and analysis engine supports multi-threaded processing and uses two basic thread models: a staged pipeline to stitch together processing stages (dissection, signature matching, statistics etc), and a pool model to parallelize processing within each stage. Each stage is run on a different core and we implement lock-free ring buffers (Valois, 1994) to ensure high throughput across the pipeline buffer and ensure data synchronization. The lock-free data structure was built using Compare-and-Swap (CAS) primitives provided by the underlying x86 architecture. The packet dissection is performed by protocol specific finite state machines (FSMs). Layers within a network packet are modelled as states and transitions between states are modelled as events. Using FSMs allows one to add and remove protocol dissectors easily and provides one with the ability to dynamically assign "processing depth" for an individual packet. For example, the DNS FSM allows the programmer to decide how far into the packet to dissect.

The online evaluation spans a period of 24 hours in November, 2012. The traffic reflects well-known diurnal patterns, with a large mid-day peak of approximately 80,000 DNS connections per minute. However, NX traffic accounts for less than 10% of the overall traffic, which highlights one of the benefits of using such data for botnet detection. Throughput analysis shows that the prototype can operate on live traffic with zero packet loss and $< 15\%$ CPU utilization. Note that by using NX traffic, DNS zones (rather than fully qualified domain names), domain name caching, and Zipf filters, one is able to store state information on the order of megabytes versus gigabytes. In larger deployments, one could use space efficient data structures (e.g., bloom filters) to keep track of state for several million IP addresses. This is left as an exercise for future work.

Analysis of our results show 63 cases of suspected or known malicious traffic. Included in the findings were the `TDSS` and `Z` bots, numerous spambots, an `OSX.FlashFake` trojan and a `FakeAV` trojan. Furthermore, the prototype detected traffic of `RunForestRun` (Unmask Parasites, 2012) and `BlackHole` (Sophos Inc., 2012). One noteworthy discovery was that of the so-called Italian typo-squatting trojan (Eckelberry,

2007) that uses domains that are misspellings of existing domains (e.g. `gbazzetta.it,gazzxetta.it`). Interestingly, the domain names used by this trojan would have relatively low edit distance scores making it difficult to detect them using the similarity-based techniques in §6.4.

### 6.7 Data Summary and Evaluation for Measurement Period II

In order to assess the longterm utility of the approach, a prototype monitored the campus network for 8 months from September 2015 to April 2016 and monitored on average 25,000 internal network clients per day. The prototype used a NX zipf filter that contained 256 NX domains; furthermore, the prototype was restarted on a daily basis to reset the state. IP addresses flagged malicious were logged to daily files along with the domain names that caused the flagging. Before discussing the results, the next section shows why the parameters used in the 2012 study are still valid today.

**On Parameter Selection** To assess the proper parameter selection for the updated study, we conducted an analysis similar to that in 2012 using one week of data from October 1-7, 2015. Table 6.6 shows the breakdown of the 7 day dataset. Note that even though the 2015 analysis was done on a subset of the UNC network, DNS traffic levels have increased over 2012 with the number of DNS requests increasing by an average of 19 million per day. The dataset was broken into individual days, and an offline analysis of the data identified 18 known bot instances over the week. Bots were labelled using three different approaches as done in the original study: 1.) client domains were cross-referenced against a publicly available AGD list of popular botnets[3], 2.) client domains were checked to see whether they were sink-holed, and 3.) the remaining client domains were hand-labeled based on whether they had similar name structure as existing AGDs, generated a sequence of at least two or more domains names that followed a similar structural convention (e.g., character set and length of the domain name), and received NX responses. Given the labeled clients, we generated the updated graph version of Figure 6.6 for October 2015 as shown in Figure 6.12. The Figure shows that 98% of benign hosts receive NX responses for four or less unique zones, while 100% of bots receive NX responses for at least four unique zones. As in 2012, there is a strong delineation between benign and infected hosts at 4 failure events (i.e., $\delta = 4$). Using $\delta$ as a guide, $\theta_0$ and $\theta_1$ are calculated as described in Section 6.6 leading to similar values ($\theta_0 = 0.9$ and $\theta_1 = 0.65$) as those calculated in 2012 ($\theta_0 = 0.9$ and $\theta_1 = 0.6$). These parameters are conservatively set to detect bots in as little as 4 DNS samples and on average

[3] `https://dgarchive.caad.fkie.fraunhofer.de/`

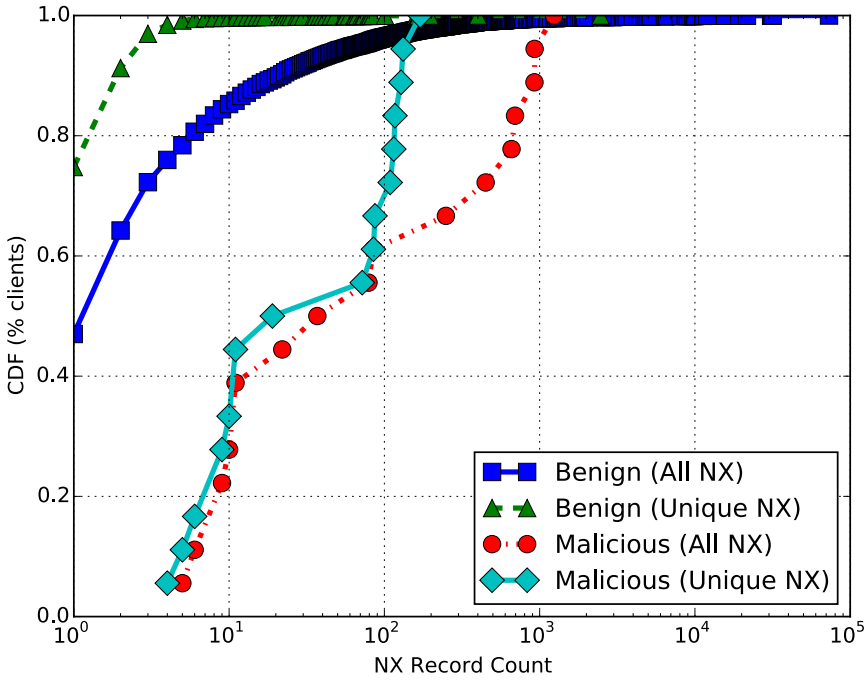



Figure 6.12: NX zone counts for benign and malicious clients for October 1-7, 2015.

5.6 samples, while providing a reasonable false positive rate (see below). The parameters could be adjusted to increase the number of detection samples required, and further reduce the number of false positives given that over 86% of bots analyzed in this period generate 9 or more unique NX responses; however, by leaving the number of samples required low, the approach can detect other malicious behavior that is not bot related as discussed below.

| | **Oct 1** | **Oct 2** | **Oct 3** | **Oct 4** | **Oct 5** | **Oct 6** | **Oct 7** | **Δ vs. 2012** |
|---|---|---|---|---|---|---|---|---|
| # Internal DNS Clients | 25.8K | 25.2K | 21.4K | 21.4K | 25.7K | 26K | 25.9K | -42.8K |
| # DNS Queries | 71.8M | 69.7M | 67.6M | 68.9M | 72.6M | 75.7M | 77.5M | +19.1M |
| # NX response | 495K | 478.2K | 347.2K | 350.8K | 437.4K | 449.8K | 424.9K | -1.1M |
| # unique domains | 682.8K | 690.4K | 540.9K | 593.1K | 698K | 676.8K | 644.3K | -1.0M |
| # unique zones | 159.5K | 152.4K | 127.7K | 138K | 162.3K | 162.1K | 156.1K | -338.2K |
| # unique NX domains | 58.7K | 56.6K | 42.3K | 40.3K | 58.4K | 53.2K | 50.5K | -157.4K |
| # unique NX zones | 3.1K | 3.1K | 2.4K | 2.3K | 3.2K | 3.1K | 3.1K | -17.6K |

Table 6.6: DNS traffic stats for seven days in October 2015.

### 6.7.1 Offline Analysis

The accuracy of the classifier was assessed with a 7-fold cross-validation on the dataset using a training window size of $\Delta = 24$ hours. Figure 6.13 shows the prediction errors computed using the root mean square error. Experiments provide an average prediction error of 0.063 or twice that of the 0.034 prediction error

found in 2012. The true positive and false positive rates were calculated as 0.95 and 0.0039, which are comparative to the 0.94 and 0.002 rates calculated in 2012.

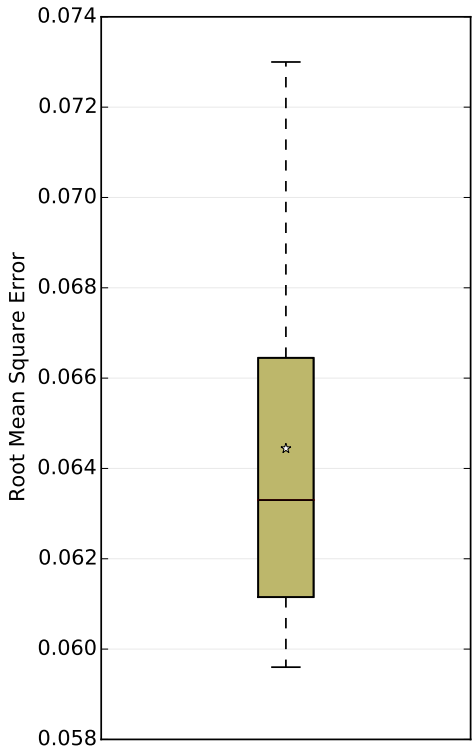



Figure 6.13: Box-and-whisker plot of the error estimation for 7-fold cross-validation for October 1-7, 2015.

### 6.7.2 Online Analysis

In the following subsection we discuss some of the key findings over the 8 month study. This includes an analysis of the behaviors of detected bots, and other potential malicious network behaviors detected by the approach.

**DGAs:** Overall, the prototype flagged 560 client exhibiting domain generation behavior. The vast majority of the traffic did not come from bot traffic, but rather as URLs for potentially malicious websites. An analysis of domain names using Google's search engine shows that attackers are embedding multiple AGDs in website forums leading to exploit kits or websites (see Table 6.7 for example AGDs). We found 177 different IP addresses querying such domains.

The second most prominent type of AGD in the data was generated by 140 different IP addresses and was seen on 281 days over the 8 month analysis period (see *Unknown* in Table 6.7). Unfortunately, little is

known about what application is generating this activity currently, because web searches and comparisons with a publicly available AGD list of popular botnets[4] have not lead to information about the source.

There were AGDs from three known botnets found in the data including 3 Bedep instances, 2 Tempedreve instances, and 15 Suppobox instances. The Bedep malware family is known for perpetrating click fraud and acting as a downloader for other malware. It was the malware payload for the Angler exploit kit in early 2015, and is known for using information from a European foreign exchange rate website as a global seed value to generate domain names (Schwarz, 2015). Bedep generates random domain names between 12 and 18 characters (excluding top level domain) as seen in Table 6.7. Bedep's AGD traffic appeared on 3 different days (October 1 and 26 2015, March 26 2016) by 3 different IP addresses. Temperdreve is a trojan that focuses on information from the victim's computer by gathering system information, taking screenshots, and performing a man in the middle browser attack (Pereira, 2015). The trojan generates AGD using a hardcoded seed (see sample domains in Table 6.7) and appeared on the network for one day on October 20, 2015.

The Suppobox Trojan was by far the most popular botnet found on the network with 15 instances found over the analysis period. Suppobox uses concatenated word lists, rather than random characters, to generate domain names based on a date and time. All known instances of the trojan generate domain names in the `.net` top-level domain name (Geffner, 2014). Different flavors of the Trojan use different word lists. For example, Suppobox Christabelle (see Table 6.7) concatenates formal names together, while Suppobox Milk uses dictionary words. The Trojan's DNS activity was flagged everyday between September 21st and October 26th, 2015 with 10 instances appearing over that time. The other five instances appeared sporadically over the month of February 2016.

**Reverse DNS Lookup Scanning?** Not all malicious traffic detected by the approach is DGA related. We observed a new type of NX traffic that was not present in the first study in 2012. Specifically, there are a set of internal clients (130 clients were flagged over the 8 month period) on the network that conduct reverse DNS lookups on external IP addresses and then try to contact those hosts using the domain names returned in the DNS lookup. The external IP addresses are associated with Internet Service Provider (ISP) customers, as is evident by the types of domain names queried by the internal clients (see examples in Table 6.7). Queries on the domain names result in NX responses, which are flagged by the prototype. An analysis of three of the flagged internal clients that conducted the reversed DNS lookups suggests that they are likely email

[4] `https://dgarchive.caad.fkie.fraunhofer.de/`

| Category | Example AGDs |
|---|---|
| *Malicious Websites* | www.56cfghsdfghd3fhfghdfgs.com<br>www.gs78hgfds3gs3f4g3sdfgsdfgsdgf.com |
| *Unknown* | oozahynbq77v-3x6vdrda.0n1qwz0ql.com<br>4vv6nf0tv9bhmyfx.1dgfipjf5se5lmpldrkfs.com |
| *Bedep* | lmqzavjmwigzxmn9.com,<br>vrhaljaogjboy.com,<br>dhdfblfaoevumoyqu.com |
| *Temperdreve* | whuzujat.org,<br>gownvjwm.info<br>kpgrape.com |
| *Suppobox Christabelle* | charlottebenjaminson.net<br>charlottecartwright.net<br>stephaniebenjaminson.net |
| *Suppobox Milk* | deadhear.net<br>rockhear.net<br>rockrule.net |
| *Reverse Scanning traffic* | 207-201-236-45.static.twtelecom.net<br>fixed-190-247-92.iusacell.net<br>host-169-217.static.telecet.ru |

Table 6.7: Example flagged NX domain names.

servers. At various intervals during the day, external computers attempt to contact the internal clients on email ports, and remain connected for minutes at a time. Of the 130 internal client IPs flagged, 22 of them were flagged on 50 or more days, while 14 IPs were flagged between 100 and 225 days over the 8 month period, suggesting that the internal clients are highly active on the campus network. Furthermore, the external IP addresses originate from Brazil, Taiwan, Thailand, China, Indonesia, and Russia and many are reported on public blacklists for various abuses including spam, denial of service, and brute force ssh attacks.

**Throw-away Domains:** Attackers will register domain names for malicious websites knowing that they will eventually be blocked by the defender. The attacker's goal is to infect as many victims before their website is discovered. Some of the only remnants of a website after it has been taken down, are the existing hyperlinks to the to the website, and the DNS NX responses corresponding to such requests. The prototype was able to identify 132 IP addresses that made requests to malicious and now defunct websites. This flagging is useful for a few reasons. First, infected machines may continually make DNS requests for external sites that no longer exist. For example, there were two hosts on the network that made requests to the domain name `sy3knom99.no-ip.biz`. Such requests are indicative of the Generic.dx!f0238dad1b trojan. The trojan was active daily over the months of February, March and April in 2016.

As mentioned earlier, it is almost impossible to detect all attacks as they occur. NX traffic provides analysts with clues of what websites and servers potentially existed on the Internet in the past. Combining the

sequential hypothesis detection approach along with a historical network traffic storage framework (Taylor et al., 2012) would allow analysts to verify whether network clients had successfully accessed such sites in the past, and potentially conduct a larger scale retroactive analysis of the client to see whether it is infected. Although retroactive analysis is not ideal, it may help to reduce the gap between the time it takes to launch a successful attack, and the time it takes to detect the attack.

In summary, AGDs are still prevalent on the campus network and the prototype was able to detect 20 new bot instances as well as 2 instances of the Generic.dx!f0238dad1b trojan. The prototype also flagged hundreds of clients that made requests to domains of malicious websites.

### 6.8 Limitations

A straightforward evasive strategy is for a bot to spread its DNS queries across a large time window, essentially implementing a low and slow approach. While this is a viable strategy, doing so drastically slows a bot's ability to communicate with its command-and-control server — resulting in a clear win for defenders. Another strategy is to attempt to increase state tracking overhead by making DNS requests from spoofed IPs. That said, in modern networks practical IP spoofing is readily detectable, especially when media access control (MAC) address registration is enforced. Alternatively, if IP spoofing is a significant concern, one could enforce DNS over TCP for local hosts connecting to internal resolvers.

### 6.9 Discussion and Lessons Learned

In this chapter, currently available techniques for detecting malicious, algorithmically generated, domain names were examined with a focus high accuracy and timeliness — key operational requirements. While contemporary techniques can detect the presence of malicious domain names, they incur high false positive rates, and require long observation periods before classification can occur making them infeasible as detectors in an operational environment. To address these shortcomings of contemporary approaches, a lightweight technique based on sequential hypothesis testing is presented. The approach takes advantage of the fact that bots generate a relatively high number of unique NX responses when searching for a command-and-control server. Extensive empirical evaluations show that hosts can be classified in as little as three to four NX responses, on average. Moreover, the lightweight nature of the approach makes it well-suited for real-world deployment.

**Key Take-Aways**

1. Using the character distribution and structure of the domain name are not good mechanisms for detecting bots that use AGDs because these features can be easily changed. Attackers are adapting and are now generating domain names based on random words, or names, and can easily make them look like benign domain names. By contrast, the use of NX traffic is inherent to the attacker's algorithm for connecting to a command-and-control server. Utilizing the contextual patterns inherent in the NX patterns affords a much more robust, necessary, and sufficient feature set for identifying such malware, and forces the attacker to investigate new approaches for communication, thus increasing the bar for the attacker.

2. The sequential hypothesis testing approach described in this chapter can classify a bot after viewing a few DNS NX responses. In doing so, the approach can detect bots much faster than contemporary approaches and with low false positive rates. The approach also stands the test of time as it is still detecting new bot instances 3 years after the initial study indicating that attackers are still abusing DNS to set up command-and-control channels.

3. In this chapter, NX traffic was used to identify bots, but I believe NX traffic may also provide clues in detecting other potentially malicious behaviors on a network. Attackers are constantly relocating their malicious websites in order to avoid detection and must use temporary (or throw-away) domain names to do so. The remnants of these domain names are seen in NX traffic, and more research is needed to determine how we can best capitalize on NX traffic to better protect our networks. One thought is to combine network forensics along with NX traffic to allow for analysis of past events. Doing so would allow an analyst to see if anyone on the network accessed a domain when it was active, potentially identifying infected clients that would have gone unnoticed otherwise.

# CHAPTER 7: CONCLUSION AND FUTURE DIRECTIONS

This dissertation highlighted the large performance gap between the ideal network-based malfeasance detector and the current state of the art for web-based exploit kits. Indeed, there are several operational challenges — e.g., the high cost of errors, diversity of traffic, semantic gap of interpreting results — that must be dealt with in order to provide more robust and usable network-based detectors. In order to reduce that gap, three new techniques that utilize the contextual information supplied by web and DNS traffic for detecting web-based exploit kits, and their corresponding bots were proposed. Chapter 4 showed that by utilizing the structural relationships in HTTP traffic, one could significantly reduce the high cost of errors over current state-of-the-art approaches, and provide an analyst with contextual information about which website loaded the exploit thus reducing the semantic gap. The key limitation of the approach is that any exploit must have already been seen in order to be detected. In order to address that limitation, a new approach was proposed in Chapter 5 that provides the same contextual benefits from utilizing the structural relationships in HTTP traffic, but can detect new exploits using behavioral features by caching and replaying possible web-based exploits in a honeyclient. Behavioral features describe what the exploit is doing rather than what it looks like, and can provide much more useful semantic information to a security analyst in an operational setting, while significantly improving false positive and false negative rates over current state-of-the-art approaches. However, extracting behavioral features is challenging because there is a significant time investment involved in running an exploit, meaning any approach will not be able to keep pace with the sheer number of possible exploit files seen by a network monitor. As a result, Chapter 5 proposes novel filtering techniques based on white-listing that allows a honeyclient to keep pace with the deluge of HTTP traffic on the network.

While Chapters 4 and 5 focus on the operational challenges faced with detecting exploit kits pre-infection, Chapter 6 focuses on the operational challenges post-infection for bots that abuse the domain name system to randomly generate domain names and by-pass defenders. Chapter 6 showed that current state-of-the-art detection approaches require hundreds of domain name samples and use domain name structure to detect bot-related traffic. Unfortunately, these approaches can take hours or even days to collect enough domain

name samples to accurately detect such nefarious activity, making them unusable for timely bot detection in enterprise network. In response, Chapter 6 proposes a new approach that uses the contextual patterns of DNS NX (non-existent) responses to detect bots in approximately three samples enabling security analysts to identify bots before they can contact the attacker. Furthermore, the approach can detect bots using a small fraction of the overall DNS traffic. As hypothesized, using the contextual patterns in HTTP and DNS traffic has significant impact on the detection performance of network-based malfeasance detectors, and thus the ideas proposed in this dissertation will help security analysts be more efficient and effective in their jobs. There are many challenges ahead, and the rest of this section discusses possible directions for future research.

**Intelligence Sharing** One of the tasks that I repeatedly did during the creation of this dissertation was to read blogs, and search the Internet for information about recent attacks as well the IP addresses, URLs and domain names for dataset labeling. This necessary task took hundreds of manual hours in front of a web browser. In this future work, I want to explore techniques that automatically extract network security related information from web resources such as security blogs, social media, bulletins and news reports to be then transformed into rules or signatures that can be directly applied to protect our networks, or label datasets for analysis. There are many challenges in this task, but the most significant challenge is that the current information extraction techniques from unstructured text require large training sets for good performance. Such data may not be available in a security context. There is also the challenge of how to turn the extracted information into meaningful rulesets that can be applied to dataset labeling or network defense.

Intelligence gathering is by no means new. Searchable web-based platforms such as National Vulnerability Database and IBM's X-Force Exchange[1] allow security analysts to access terabytes of information about URLs, IPs, applications, and vulnerabilities while also participating in security communities for discussion. Since these systems are proprietary, there is little information about how such knowledge bases are gathered. Closer in nature to my proposal are works by Joshi et al. (2013), Jones et al. (2015) and Ritter et al. (2015) that automatically extract information from semi-structured databases, security bulletins and tweets. Joshi et al. (2013) extracts information from the National Vulnerability Database and security bulletins to automatically publishes a RDF linked data representation of cybersecurity concepts and vulnerability descriptions. This work does not address how extracted information can be used for network-based signatures and has relatively high false positive rates. Jones et al. (2015) present a new bootstrapping algorithm for extracting security

[1] `https://exchange.xforce.ibmcloud.com/`

entities and their relationships from text that incorporates the security analyst in order to improve classification performance. Unfortunately, the work is preliminary and there is no indication it will scale. Ritter et al. (2015) use a weakly supervised classification model to categorize security events such as denial of service, data breaches and account hijacking from twitter feeds, but does not extract semantic information from the tweets that could be used to automatically defend networks. By contrast, Bridges et al. (2014) solve the inverse problem of annotating unstructured text on security topics using a structured vulnerability database. Such an approach could be useful in generating unstructured training sets for information extraction. Overall, this area of research has recently emerged providing opportunities to make real-world operational impact for security analysts.

**Extending the Model for Network-based Honeyclient analysis** Chapter 5 demonstrated that honeyclients can be effectively applied to network-based malfeasance detection using behavioral feature sets; however, there are still challenges ahead. Cisco reports that 72% of all exploit kit attacks are launched through Flash (Biasini et al., 2015) because Flash plugins are vulnerable and predominant on client machines. Due to these vulnerabilities, companies are slowly turning away from Flash towards HTML5, and JavaScript. As Flash usage declines, attackers will increasingly look for vulnerabilities in JavaScript leaving defenders unprepared. JavaScript provides the same challenges for defenders as Flash except there are millions of JavaScript files downloaded on networks per day, and JavaScript can be embedded in HTML. As described in Chapter 5, current detection approaches for JavaScript are broadly placed in offline honeyclient analysis (Invernizzi et al., 2012; Li et al., 2012, 2013; Eshete and Venkatakrishnan, 2014; Thomas et al., 2015) or online machine learning approaches (Rieck et al., 2010; Canali et al., 2011; Blum et al., 2010; Ma et al., 2009, 2011; Mekky et al., 2014; Nelms et al., 2015). I want to investigate how one could extend the honeyclient model from Chapter 5 to scale for JavaScript. As a first step, I would like to conduct a longitudinal study on client usage patterns of JavaScript to gain a better understanding of how many unique JavaScript files are downloaded on a daily basis, where they come from, and how often they change. Such a study will provide clues on how to build JavaScript filters that would enable the honeyclient technique to scale to large networks. One relevant study is that of Soni et al. (2015) who studied JavaScript files from the top 500 Alexa websites. They found that only 7% of JavaScript files change overtime, and use that fact to build JavaScript white lists to protect a client browser from malicious scripts. It is unclear how such results hold for the diverse set of JavaScript seen on a large enterprise network. Even with JavaScript filters, the honeyclient model will likely still not

scale to the quantity of JavaScript. As a result, another interesting project would be to investigate how the honeyclient model could be adapted to the cloud (Sherry et al., 2012).

**Network Forensics** Defenders are a long way from creating the ideal network-based intrusion detector that can detect all intrusions as they occur. Recent examples of successful attacks on companies include Sony (Kroft, 2015) and Anthem (Newcomb, 2015) and future incidents are an assurity as Ponemon Institute (2015) reports that it takes companies on average 256 days to identify a malicious attack. As a result, defenders are left to recreate the breach and decipher the true impact of the attack. This requires the collection, storage and rapid querying of network traffic. Attackers have two big advantages that make tracking them difficult. They have the advantage of time — attackers only need to exploit a single machine in a network to gain access, and they can conduct surveillance for months, slowly compromising more computers until they find their target data to exfiltrate. Attackers also have the advantage because they can bury their footprints in mountains of traffic making uncovering an attack akin to finding a needle-in-a-haystack.

Academic research into network forensics focuses on improving the collection and query speeds of netflow. Netflow records contain a summary of the packets from a unidirectional or bidirectional connection between two computers and has fields like source/destination IP addresses, source/destination ports, protocol, and packet counts. Storage systems for netflow range from scan and filter (Shimeall et al., 2010) to fully indexed datastores (Bethel et al., 2006; Deri et al., 2010; Reiss et al., 2007; Giura and Memon, 2010; Fusco et al., 2011). With the increasing sophistication of attacks, researchers have looked beyond netflow to storing payload attributes, and modelling those payloads as documents to allow search engine style queries (Taylor et al., 2012) or by extracting payload features and correlating them as keys in a key-value store (Schales et al., 2015). While these approaches represent a step in the right direction, more research is needed to help shorten the time it takes an analyst to identify a malicious attack.

One significant research challenge is how to store, and manage the millions of records a network generates per day over many sources — network traffic, intrusion detection logs and network logins. Given that attacks can take months to uncover, we must be able to store at least a years worth of data; however, most of that data only serves to help camouflage the attacker's presence in a network. I want to investigate smart data compression and reduction techniques for reducing the overall data footprint to enable better data management and archiving. Data compression and reduction have received little attention in the research community. Cooke et al. (2006) store traffic in different granularities (i.e., full packets, netflow, aggregates, and events)

depending on time and space requirements. Similarly, Papadogiannakis et al. (2010) propose RDDtrace, a tool that stores raw packets. As traffic ages, packets are removed to make them more space efficient. Maier et al. (2008) also store packet data, but only store the first N bytes, noting that a relatively few network connections comprise approximately 80% of all network traffic at any one time. Gugelmann et al. (2013) explore ways to compress HTTP traffic based on the "trustworthiness" of the websites involved. Although interesting, the approach only scratches the surface for building smarter network forensic storage frameworks, that provide a small data footprint, but allow for rich querying capabilities. I want to investigate behavioral or semantic lossy compression approaches to reduce data footprints based on client usage patterns and data uniqueness properties rather than temporal considerations. By doing so, we can make it easier to uncover potentially malicious events. There are relatively few works on semantic compression with most coming from the area of graph compression. Gilbert and Levchenko (2004) investigate semantic compression of network graphs for visualization, by ranking vertices based on degree, shortest path, or importance. They also explore similarity-based compression schemes such as redundant vertices, geographic clustering, and shared medium. Navlakha et al. (2008) compress graphs by creating a "graph summary" which is an aggregate graph in which each node corresponds to a set of nodes in G, and each edge represents the edges between all pair of nodes in the two sets while Borici and Thomo (2014) compress graphs by modelling their semantic structural features as hypergraphs. While these approaches are applicable to the task at hand, I believe more domain specific analysis and compression is needed.

The other important research area for network forensics is how to provide a security analyst with fast and powerful query capabilities across many different data sources (e.g., HTTP, DNS, netflow, login info, system logs) and over a time dimension that can span several months. I am interested in data organization techniques and indexing structures to improve query performance over heterogeneous datasets while enabling correlations across time and space. During targeted attacks on companies such as Anthem, the attacker will compromise one or more hosts to gain access to the network, and then slowly compromise other hosts over time, pivoting from one machine to the next. Security analysts need to recreate these attacks, and track the attacker's movements, meaning that network forensic frameworks need the ability to link data sources over time, and make connections between network entities (e.g., computers, MAC addresses, logins).

One potentially useful data structure that has remained unexplored in network forensics is a heterogeneous time-series graph. In time-series graphs, nodes represent internal devices or external servers, and edges are represented as a timestamped set of events between these nodes. Time-series graph research is relatively new.

Most current research focuses on "temporal" (Simmhan et al., 2015) graphs, and how to load and run a large scale analysis on a full graph across a cluster of machines. A temporal graph is an in-memory graph (Han et al., 2014; Cheng et al., 2012; Riedy et al., 2012; Macko et al., 2015), where timestamps are used to create graph snapshots. Analysis is typically vertex or sub-graph centric (Simmhan et al., 2015). Graphs are stored using a multitude of structures. For example, Riedy et al. (2012) present Stinger, an in-memory graph analysis tool that stores a graph using a vertex array, and edge blocks while Macko et al. (2015) stores multiple graph revisions using a compressed sparse row representation. Cheng et al. (2012) describe a streaming based graphing system that takes graph snapshots at pre-determined intervals and Prabhakaran et al. (2012) investigate graph layout and partitioning to support optimized graph processing and querying. Han et al. (2014) describe Chronos, an in-memory data structure for temporal graphs. Chronos focuses on the memory layout of temporal graphs and found that using a temporal locality scheme for vertices rather than a structural locality scheme improved in-memory performance. None of these works explore the challenges of storing and querying time-series graphs on disk. There is surprisingly little work on the topic. GoFS (Simmhan et al., 2014) is a graph-oriented files system for distributed storage of time-series graphs on commodity clusters. It is designed to load subgraphs in the local host's graph partition for analysis but was not designed for dynamically changing graph nodes and subgraph querying. Future research should explore how to organize time-series graphs on disk to support queries that help the analyst reduce the time spent identifying attacks.

## BIBLIOGRAPHY


Acar, G., Juarez, M., Nikiforakis, N., Diaz, C., Gürses, S., Piessens, F., and Preneel, B. (2013). Fpdetective: Dusting the web for fingerprinters. In *ACM Conference on Computer and Communications Security*.

Aleph One (1996). Smashing the stack for fun and profit. *Phrack Magazine*, 49.

Antonakakis, M., Perdisci, R., Dagon, D., Lee, W., and Feamster, N. (2010). Building a Dynamic Reputation System for DNS. In *USENIX Security Symposium*.

Antonakakis, M., Perdisci, R., Lee, W., Vasiloglou, N., and Dagon, D. (2011). Detecting Malware Domains at the Upper DNS Hierarchy. In *USENIX Security Symposium*.

Antonakakis, M., Perdisci, R., Nadji, Y., Vasiloglou, N., Abu-Nimeh, S., Lee, W., , and Dagon, D. (2012). From Throw-Away Traffic to Bots: Detecting the Rise of DGA-based Malware. In *USENIX Security Symposium*.

Arends, R., Austein, R., Larson, M., Massey, D., and Rose, S. (2005). DNS Security Introduction and Requirements. RFC 4033, The Internet Society.

Asai, T., Abe, K., Kawasoe, S., Arimura, H., Sakamoto, H., and Arikawa, S. (2002). Efficient substructure discovery from large semi-structured data. In *IEEE International Conference on Data Mining*.

Augsten, N., Barbosa, D., Bohlen, M., and Palpanas, T. (2011). Efficient top-k approximate subtree matching in small memory. *IEEE Transactions on Knowledge and Data Engineering*, 23(8).

Axelsson, S. (1999). The base-rate fallacy and its implications for the difficulty of intrusion detection. In *ACM Conference on Computer and Communications Security*.

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

Bethel, E., Campbell, S., Dart, E., Stockinger, K., and Wu, K. (2006). Accelerating Network Traffic Analytics Using Query-Driven Visualization. In *IEEE Symposium on Visual Analytics Science And Technology*.

Biasini, N., Esler, J., Mercer, W., Olney, M., Taylor, M., and Williams, C. (2015). Threat spotlight: Cisco talos thwarts access to massive international exploit kit generating $60m annually from ransomware alone. `http://blogs.cisco.com/security/talos/angler-exposed`.

Bilge, L., Kirda, E., Kruegel, C., and Balduzzi, M. (2011). EXPOSURE: Finding Malicious Domains using Passive DNS Analysis. In *Symposium on Network and Distributed System Security*.

Blum, A., Wardman, B., Solorio, T., and Warner, G. (2010). Lexical feature based phishing url detection using online learning. In *ACM Workshop on Artificial Intelligence and Security*.

Borici, A. and Thomo, A. (2014). Semantic graph compression with hypergraphs. In *IEEE International Conference on Advanced Information Networking and Applications*.

Born, K. and Gustafson, D. (2010). Detecting DNS Tunnels Using Character Frequency Analysis. In *Annual Computer Security Applications Conference*.

Bridges, R. A., Jones, C. L., Iannacone, M. D., Testa, K. M., and Goodall, J. R. (2014). Automatic labeling for entity extraction in cyber security. In *ASE International Conference on Cyber Security*.

Buchanan, E., Roemer, R., Shacham, H., and Savage, S. (2008). When good instructions go bad: Generalizing return-oriented programming to risc. In *ACM Conference on Computer and Communications Security*.

Bui, D. B., Hadzic, F., Tagarelli, A., and Hecker, M. (2014). Evaluation of an associative classifier based on position-constrained frequent/closed subtree mining. *Journal of Intelligent Information Systems*, 45(3).

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

Chehreghani, M., Lucas, C., and Rahgozar, M. (2011). Oinduced: An efficient algorithm for mining induced patterns from rooted ordered trees. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 41(5).

Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., and Chen, E. (2012). Kineograph: Taking the pulse of a fast-changing and connected world. In *ACM European Conference on Computer Systems*.

Chi, Y., Yang, Y., and Muntz, R. R. (2004). Hybridtreeminer: an efficient algorithm for mining frequent rooted trees and free trees using canonical forms. In *International Conference on Scientific and Statistical Database Management*.

Chi, Y., Yang, Y., and Muntz, R. R. (2005). Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowledge Information Systems*, 8(2).

Chi, Y., Yang, Y., Xia, Y., and Muntz, R. R. (2003). Cmtreeminer: Mining both closed and maximal frequent subtrees. In *Pacific Asia Conference on Knowledge Discovery and Data Mining*.

Clark, J. (2013). Malicious javascript flips ad network into rentable botnet. `http://goo.gl/8mFLvQ`.

Cohen, S. (2013). Indexing for subtree similarity-search using edit distance. In *ACM Conference on Management of Data*.

Cooke, E., Myrick, A., Rusek, D., and Jahanian, F. (2006). Resource-aware multi-format network security data storage. In *SIGCOMM Workshop on Large-scale attack defense*.

Cova, M., Kruegel, C., and Vigna, G. (2010). Detection and analysis of drive-by-download attacks and malicious javascript code. In *World Wide Web Conference*.

da Jiménez, A., Berzal, F., and Cubero, J.-C. (2010). Frequent tree pattern mining: A survey. *Intelligent Data Analysis*, 14(6):603–622.

De Maio, G., Kapravelos, A., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2014). PExy: The Other Side of Exploit Kits. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

Deepak, A., Fernández-Baca, D., Tirthapura, S., Sanderson, M., and McMahon, M. (2013). Evominer: frequent subtree mining in phylogenetic databases. *Knowledge and Information Systems*.

Deri, L., Lorenzetti, V., and Mortimer, S. (2010). Collection and exploration of large data monitoring sets using bitmap databases. In *International Conference on Traffic Monitoring and Analysis*.

Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., and Weihl, B. (2002). Globally distributed content delivery. *IEEE Internet Computing*, 6(5).

Duncan, B. (2014). Malware-traffic-analysis.net blog. `http://goo.gl/fXdSZz`.

Eckelberry, A. (2007). Massive italian typosquatting ring foists malware on users. http://goo.gl/4ZzMI.

Eshete, B. and Venkatakrishnan, V. N. (2014). Webwinnow: Leveraging exploit kit workflows to detect malicious urls. In *ACM Conference on Data and Application Security and Privacy*.

Everitt, B., Landau, S., Leese, M., and Stahl, D. (2011). *Cluster Analysis*. Wiley Series in Probability and Statistics. Wiley.

Felegyhazi, M., Kreibich, C., and Paxson, V. (2010). On the potential of proactive domain blacklisting. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*.

Fusco, F., Vlachos, M., and Stoecklin, M. (2011). Real-time creation of bitmap indexes on streaming network data. *The VLDB Journal*.

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

Geffner, J. (2014). End-to-end analysis of a domain generating algorithm malware family. RSA Conference 2014.

Gemignani, G., Klee, S. D., Veloso, M., and Nardi, D. (2015). On task recognition and generalization in long-term robot teaching. In *International Conference on Autonomous Agents and Multiagent Systems*.

Gilbert, A. C. and Levchenko, K. (2004). Compressing network graphs. In *Knowledge Discovery and Data Mining*.

Giura, P. and Memon, N. (2010). NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring. In *Symposium on Recent Advances in Intrusion Detection*.

Golovanov, S. and Soumenkov, I. (2011). TDL4 Top Bot. See http://goo.gl/23BaA.

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

Grill, M., Nikolaev, I., Valeros, V., and Rehak, M. (2015). Detecting dga malware using netflow. In *IFIP/IEEE International Symposium on Integrated Network Management*.

Gugelmann, D., Schatzmann, D., and Lenders, V. (2013). Horizon extender: Long-term preservation of data leakage evidence in web traffic. In *ACM Symposium on Information, Computer and Communications Security*.

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

Hadzic, F., Hecker, M., and Tagarelli, A. (2015). Ordered subtree mining via transactional mapping using a structure-preserving tree database schema. *Journal of Information Sciences*, 310.

Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., and Chen, E. (2014). Chronos: A graph engine for temporal graph analysis. In *ACM European Conference on Computer Systems*.

Hao, S., Feamster, N., and Pandrangi, R. (2011). Monitoring the Initial DNS Behavior of Malicious Domains. In *ACM Internet Measurement Conference*.

Hido, S. and Kawano, H. (2005). Amiot: induced ordered tree mining in tree-structured databases. In *IEEE International Conference on Data Mining*.

Ho, J.-W., Wright, M., and Das, S. (2011). Fast detection of mobile replica node attacks in wireless sensor networks using sequential hypothesis testing. *IEEE Transactions on Mobile Computing*, 10(6).

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

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

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

Ihm, S. and Pai, V. S. (2011). Towards understanding modern web traffic. In *ACM Internet Measurement Conference*.

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

ISC (2011). Google Chrome and (weird) DNS Requests. `http://goo.gl/j48CA`.

Jiang, N., Cao, J., Jin, Y., Li, L. E., and Zhang, Z.-L. (2010). Identifying suspicious activities through dns failure graph analysis. In *International Conference on Network Protocols*, pages 144–153.

Jiménez, A., Galiano, F. B., and Talavera, J. C. C. (2012). Mining frequent patterns from xml data: Efficient algorithms and design trade-offs. *Expert Systems with Applications*, 39(1).

Jones, C. L., Bridges, R. A., Huffer, K. M. T., and Goodall, J. R. (2015). Towards a relation extraction framework for cyber-security concepts. In *ACM Cyber and Information Security Research Conference*.

Joshi, A., Lal, R., Finin, T., and Joshi, A. (2013). Extracting cybersecurity related linked data from text. In *IEEE International Conference on Semantic Computing*.

Jung, J., Milito, R., and Paxson, V. (2008). On the adaptive real-time detection of fast-propagating network worms. *Journal of Computer Virology*, 4.

Jung, J., Paxson, V., Berger, A. W., and Balakrishnan, H. (2004). Fast Portscan Detection Using Sequential Hypothesis Testing. In *IEEE Symposium on Security and Privacy*.

Kaminsky, D. (2008). Black ops 2008–its the end of the cache as we know it. *Black Hat USA*.

Kirat, D., Vigna, G., and Kruegel, C. (2011). Barebox: Efficient malware analysis on bare-metal. In *Annual Computer Security Applications Conference*.

Kirat, D., Vigna, G., and Kruegel, C. (2014). Barecloud: Bare-metal analysis-based evasive malware detection. In *USENIX Security Symposium*.

Kornblum, J. (2006). Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3, Supplement.

Krishnan, S., Taylor, T., Monrose, F., and McHugh, J. (2013). Crossing the Threshold: Detecting Network Malfeasance via Sequential Hypothesis Testing. In *IEEE/IFIP International Conference on Dependable Systems and Networks*.

Kroft, S. (2015). The attack on sony. `http://www.cbsnews.com/news/north-korean-cyberattack-on-sony-60-minutes/`.

Kullback, S. and Leibler, R. (1951). On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86.

Kutty, S., Nayak, R., and Li, Y. (2007). Pcitminer: prefix-based closed induced tree miner for finding closed induced frequent subtrees. In *Australasian Conference on Data Mining and Analytics*.

Lemos, R. (2010). The doubleclick attack and the rise of malvertising. `http://goo.gl/1HzmLF`.

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

Li, Z., Zhang, K., Xie, Y., Yu, F., and Wang, X. (2012). Knowing your enemy: understanding and detecting malicious web advertising. In *ACM Conference on Computer and Communications Security*.

Lindorfer, M., Kolbitsch, C., and Milani Comparetti, P. (2011). Detecting environment-sensitive malware. In *Symposium on Recent Advances in Intrusion Detection*.

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

Ma, J., Saul, L. K., Savage, S., and Voelker, G. M. (2009). Beyond blacklists: learning to detect malicious web sites from suspicious urls. In *Knowledge Discovery and Data Mining*.

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

Macko, P., Marathe, V., Margo, D., and Seltzer, M. (2015). Llama: Efficient graph analytics using large multiversioned arrays. In *IEEE International Conference on Data Engineering*.

Maier, G., Sommer, R., Dreger, H., Feldmann, A., Paxson, V., and Schneider, F. (2008). Enriching network security analysis with time travel. In *ACM SIGCOMM Conference on Data Communication*.

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

Mockapetris, P. and Dunlap, K. J. (1988). Development of the domain name system. In *SIGCOMM Symposium on Communications Architectures and Protocols*.

Mowbray, M. and Hagen, J. (2014). Finding domain-generation algorithms by looking at length distribution. In *IEEE International Symposium on Software Reliability Engineering Workshops*.

Narouei, M., Ahmadi, M., Giacinto, G., Takabi, H., and Sami, A. (2015). Dllminer: structural mining for malware detection. *Journal of Security and Communication Networks*, 8(18).

Navlakha, S., Rastogi, R., and Shrivastava, N. (2008). Graph summarization with bounded error. In *ACM SIGMOD International Conference on Management of Data*.

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

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

Newcomb, A. (2015). Anthem hack may have impacted millions of non-customers as well. `http://abcnews.go.com/Technology/anthem-hack-impacted-millions-customers/story?id=29212840`.

Nguyen, L. and Shimazu, A. (2011). Improving subtree-based question classification classifiers with word-cluster models. In *Natural Language Processing and Information Systems*, volume 6716. Springer Berlin Heidelberg.

Nieto, J. (2013). Zeroaccess trojan - network analysis part ii. `http://goo.gl/LYssOV`.

Paik, J., Choi, W., Lee, E., and Kim, U.-M. (2008). Extraction of frequent tree patterns without subtrees maintenance. In *Future Generation Communication and Networking Symposia*, volume 2.

Papadogiannakis, A., Polychronakis, M., and Markatos, E. P. (2010). Rrdtrace: Long-term raw network traffic recording using fixed-size storage. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*.

Pawlik, M. and Augsten, N. (2011). Rted: A robust algorithm for the tree edit distance. In *International Conference on Very Large Databases*.

Paxson, V. (1999). Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24).

Paxson, V., Christodorescu, M., Javed, M., Rao, J., Sailer, R., Schales, D. L., Stoecklin, M., Thomas, K., Venema, W., and Weaver, N. (2013). Practical comprehensive bounds on surreptitious communication over dns. In *USENIX Security Symposium*.

Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., and Hsu, M.-C. (2001). Prefixspan: mining sequential patterns efficiently by prefix-projected pattern growth. In *IEEE International Conference on Data Engineering*.

Pereira, T. (2015). Tempedreve botnet overview and malware analysis. Technical report, AnubisNetworks.

Polychronakis, M., Anagnostakis, K. G., and Markatos, E. P. (2010). Comprehensive shellcode detection using runtime heuristics. In *Annual Computer Security Applications Conference*.

Ponemon Institute (2015). 2015 cost of data breach study: Global analysis. Technical report, Ponemon Institute LLC.

Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., and Haridasan, M. (2012). Managing large graphs on multi-cores with graph awareness. In *USENIX Annual Technical Conference*.

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

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

PWC (2013). IAB internet advertising revenue report: 2012 full year results. Technical report, PricewaterhouseCoopers, Interactive Advertising Bureau.

Raywood, D. (2012). Major league baseball website hit by malvertising that may potentially impact 300,000 users. http://goo.gl/upKVXe.

Reiss, F., Stockinger, K., Wu, K., Shoshani, A., and Hellerstein, J. M. (2007). Enabling Real-Time Querying of Live and Historical Stream Data. In *International Conference on Scientific and Statistical Database Management*.

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

Riedy, J., Meyerhenke, H., Bader, D., Ediger, D., and Mattson, T. (2012). Analysis of streaming social networks and graphs on multicore architectures. In *IEEE International Conference on Acoustics, Speech and Signal Processing*.

Ritter, A., Wright, E., Casey, W., and Mitchell, T. (2015). Weakly supervised extraction of computer security events from twitter. In *World Wide Web Conference*.

R.O. Duda, P. H. and Stork, D. (2007). *Pattern Classification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

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

Roesch, M. et al. (1999). Snort: Lightweight intrusion detection for networks. In *USENIX Conference on System Administration*.

Schales, D. L., Hu, X., Jang, J., Sailer, R., Stoecklin, M. P., and Wang, T. (2015). FCCE: highly scalable distributed feature collection and correlation engine for low latency big data analytics. In *IEEE International Conference on Data Engineering*.

Schechter, S. E., Jung, J., and Berger, A. W. (2004). Fast detection of scanning worm infections. In *Symposium on Recent Advances in Intrusion Detection*.

Schlumberger, J., Kruegel, C., and Vigna, G. (2012). Jarhead analysis and detection of malicious java applets. In *Annual Computer Security Applications Conference*.

Schwartz, M. J. (2013). Android malware being delivered via ad networks. http://goo.gl/CrfKzo.

Schwarz, D. (2015). Bedeps dga: Trading foreign exchange for malware domains. https://www.arbornetworks.com/blog/asert/bedeps-dga-trading-foreign-exchange-for-malware-domains/.

Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. (2012). Making middleboxes someone else's problem: Network processing as a cloud service. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*.

Shimeall, T., Faber, S., DeShon, M., and Kompanek, A. (2010). *Analysts' Handbook: Using SiLK for Network Traffic Analysis*. CERT Network Situational Awareness Group.

Simmhan, Y., Choudhury, N., Wickramaarachchi, C., Kumbhare, A., Frincu, M., Raghavendra, C., and Prasanna, V. (2015). Distributed programming over time-series graphs. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*.

Simmhan, Y., Kumbhare, A., Wickramaarachchi, C., Nagarkar, S., Ravi, S., Raghavendra, C., and Prasanna, V. (2014). Goffish: A sub-graph centric framework for large-scale graph analytics. In *International Conference on Euro-Par Parallel Processing*.

Snow, K. Z., Krishnan, S., Monrose, F., and Provos, N. (2011). Shellos: enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*.

Sommer, R. and Paxson, V. (2010). Outside the closed world: On using machine learning for network intrusion detection. In *IEEE Symposium on Security and Privacy*.

Sommer, R., Paxson, V., and Weaver, N. (2009). An architecture for exploiting multi-core processors to parallelize network intrusion prevention. *Concurrency and Computation: Practice & Experience*, 21(10).

Son, S. and Shmatikov, V. (2010). The Hitchhiker's Guide to DNS Cache Poisoning. In *International Conference on Security and Privacy in Communication Networks*.

Soni, P., Budianto, E., and Saxena, P. (2015). The sicilian defense: Signature-based whitelisting of web javascript. In *ACM Conference on Computer and Communications Security*.

Sophos Inc. (2012). Exploring the blackhole exploit kit. `http://goo.gl/ZhLvp`.

Stancill, B., Snow, K. Z., Otterness, N., Monrose, F., Davi, L., and Sadeghi, A.-R. (2013). *Check My Profile: Leveraging Static Analysis for Fast and Accurate Detection of ROP Gadgets*. Symposium on Recent Advances in Attacks and Defenses.

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

Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., and Vigna, G. (2009). Your botnet is my botnet: Analysis of a botnet takeover. In *ACM Conference on Computer and Communications Security*, pages 635–647.

Stringhini, G., Kruegel, C., and Vigna, G. (2013). Shady paths: leveraging surfing crowds to detect malicious web pages. In *ACM Conference on Computer and Communications Security*.

Subercaze, J., Gravier, C., and Laforest, F. (2015). Mining user-generated comments. In *IEEE/WIC/ACM International Conference on Web Intelligence*.

Symantec MSS Global Threat Response (2014). Six months after blackhole: Passing the exploit kit torch. `http://goo.gl/nAsxj0`.

Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*.

Tan, H., Hadzic, F., Dillon, T. S., Chang, E., and Feng, L. (2008). Tree model guided candidate generation for mining frequent subtrees from xml documents. *ACM Transactions on Knowledge Discovery from Data*, 2(2).

Tatikonda, S., Parthasarathy, S., and Kurc, T. (2006). Trips and tides: New algorithms for tree mining. In *ACM International Conference on Information and Knowledge Management*.

Taylor, T., Coull, S., Monrose, F., and McHugh, J. (2012). Toward efficient querying of compressed network payloads. In *USENIX Annual Technical Conference*.

Taylor, T., Hu, X., Wang, T., Jang, J., Stoecklin, M., Monrose, F., and Sailer, R. (2016a). Detecting malicious exploit kits using tree-based similarity searches. In *ACM Conference on Data and Application Security and Privacy*.

Taylor, T., Snow, K. Z., Otterness, N., and Monrose, F. (2016b). Cache, trigger, impersonate: Enabling context-sensitive honeyclient analysis on-the-wire. In *Symposium on Network and Distributed System Security*.

Termier, A., Rousset, M.-C., Sebag, M., Ohara, K., Washio, T., and Motoda, H. (2008). Dryadeparent, an efficient and robust closed attribute tree mining algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 20(3).

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

Thomas, M. and Mohaisen, A. (2014). Kindred domains: Detecting and clustering botnet domains using dns traffic. In *World Wide Web Conference*.

Trend Micro (2014). The aftermath of the blackhole exploit kits demise. `http://goo.gl/DsjYUp`.

Tu, T. D., Guang, C., and Xin, L. Y. (2015). Detecting bot-infected machines based on analyzing the similar periodic dns queries. In *International Conference on Communications, Management and Telecommunications*.

Unmask Parasites (2012). Runforestrun and pseudo random domains. `http://goo.gl/xRWtw`.

Valois, J. (1994). Implementing lock-free queues. In *International Conference on Parallel and Distributed Computing Systems*, pages 64–69.

Van Overveldt, T., Kruegel, C., and Vigna, G. (2012). Flashdetect: Actionscript 3 malware detection. In *Symposium on Recent Advances in Intrusion Detection*.

Villamarn-Salomn, R. and Brustoloni, J. (2008). Identifying botnets using anomaly detection techniques applied to dns traffic. In *IEEE Consumer Communications & Networking Conference*.

Wald, A. (1947). *Sequential Analysis*. John Wiley and Sons, Inc.

Wang, C., Hong, M., Pei, J., Zhou, H., Wang, W., and Shi, B. (2004). Efficient pattern-growth methods for frequent tree pattern mining. In *Advances in Knowledge Discovery and Data Mining*. Springer.

Wang, D. Y., Savage, S., and Voelker, G. M. (2011). Cloak and dagger: Dynamics of web search cloaking. In *ACM Conference on Computer and Communications Security*.

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

Weaver, N., Staniford, S., and paxson, V. (2007). Very fast containment of scanning worms, revisited. In *Malware Detection*, volume 27. Springer.

White, A., Krishnan, S., Bailey, M., Monrose, F., and Parros, P. (2013). Clear and Present Data: Opaque Traffic and its Security Implications for the Future. In *Symposium on Network and Distributed System Security*.

Xiao, Y., Yao, J.-F., Li, Z., and Dunham, M. H. (2003). Efficient data mining for maximal frequent subtrees. In *IEEE International Conference on Data Mining*.

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

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

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

Yadav, S. and Reddy, A. N. (2011). Winning with dns failures: Strategies for faster botnet detection. In *International Conference on Security and Privacy in Communication Networks*.

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

Zaki, M. J. (2005). Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 17(8).

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

Zeidanloo, H. and Manaf, A. (2009). Botnet command and control mechanisms. In *International Conference on Computer and Electrical Engineering*.

Zhao, P., Yu, J. X., and Yu, P. S. (2007). Graph indexing: tree + delta graph. In *International Conference on Very Large Databases*.

Zou, L., Lu, Y., Zhang, H., and Hu, R. (2006a). Prefixtreeespan: A pattern growth algorithm for mining embedded subtrees. In *Web Information Systems*, volume 4255. Springer Berlin Heidelberg.

Zou, L., Lu, Y., Zhang, H., Hu, R., and Zhou, C. (2006b). Mining frequent induced subtrees by prefix-tree-projected pattern growth. In *International Conference on Web-Age Information Management Workshops*.