SCALABLE FRAMEWORK FOR TURN-KEY HONEYNET DEPLOYMENT

A Thesis
Presented to
The Academic Faculty

by

Albert W. Brzeczko, Jr.

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
May 2014

SCALABLE FRAMEWORK FOR TURN-KEY HONEYNET DEPLOYMENT

Approved by:

Professor John Copeland, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor George Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Raheem Beyah
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Polo Chau
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Henry Owen
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: 2 April 2014

ACKNOWLEDGEMENTS

I want to thank my parents, who have given me tremendous support through the years. I owe a great deal also to Professor Russell Taylor and Randall Goldberg at Johns Hopkins University, who provided me with valuable mentoring and undergraduate research experience, which fostered in me a desire to pursue graduate school. Selcuk Uluagac has been a great mentor, friend, and help to me in honing my research ideas and turning them into reality. Finally, I wish to thank the members of my thesis committee, Drs. Copeland, Owen, Beyah, and Riley, for all of their support and guidance throughout the PhD process.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Enterprise networks present very high value targets in the eyes of malicious actors who seek to exfiltrate sensitive proprietary data, disrupt the operations of a particular organization, or leverage considerable computational and network resources to further their own illicit goals. For this reason, enterprise networks typically attract the most determined of attackers. These attackers are prone to using the most novel and difficult-to-detect approaches so that they may have a high probability of success and continue operating undetected. Many existing network security approaches that fall under the category of intrusion detection systems (IDS) and intrusion prevention systems (IPS) are able to detect classes of attacks that are well-known. While these approaches are effective for filtering out routine attacks in automated fashion, they are ill-suited for detecting the types of novel tactics and zero-day exploits that are increasingly used against the enterprise.

In this thesis, a solution is presented that augments existing security measures to provide enhanced coverage of novel attacks in conjunction with what is already provided by traditional IDS and IPS. The approach enables honeypots, a class of technique that observes novel attacks by luring an attacker to perform malicious activity on a system having no production value, to be deployed in a *turn-key* fashion and at large scale on enterprise networks. In spite of the honeypot's efficacy against targeted attacks, organizations can seldom afford to devote capital and IT manpower to integrating them into their security posture. Furthermore, misconfigured honeypots can actually weaken an organization's security posture by giving the attacker a staging ground on which to perform further attacks. A turn-key approach is needed for organizations to use honeypots to trap, observe, and mitigate novel targeted attacks.

CHAPTER I

INTRODUCTION

Enterprise networks exist in many diverse forms. They can belong to private sector companies, non-profit entities such as universities, government, military installations, and healthcare organizations. They can be entirely localized to one office, be connected globally across campuses by virtual private network (VPN) tunnels, employ cloud hosting, or some mix of the preceding. What is consistent among all enterprise networks, however, is that they host a set of data and infrastructure that is of high value to the organization to which it pertains, and critical to the execution of that organization's missions. Gaining access to this data and infrastructure is of potentially high value to malicious actors who would seek to exploit it for their own gain.

1.1 Summary of Automated and Targeted Attacks

Attacks on a computer network can be differentiated along the lines of automated vs. targeted. Typically, automated attacks are carried out by malicious computer software, such as a virus, worm, or bot, that propagates itself to vulnerable hosts and receives commands either by way of a centralized command-and-control infrastructure or peer-to-peer network. These automated attacks seek to infect as many machines as possible and typically do not have a set list of victims in mind. Once successful, the attacks typically cause such generic actions as delivering spam messages, carrying out distributed denial-of-service attacks, offering proxies through which to transact illegal/protected content, etc. Typically, automated attacks are carried out with crude, well-known attack patterns, and prey on systems that are poorly maintained, such as those machines that are not kept up-to-date with software security patches. In many cases, automated attacks can be thwarted by traditional signature-based

means of intrusion detection, but are typically successful due to the sheer number of machines that remain vulnerable due to inadequate patch level or misconfiguration.

Targeted attacks (sometimes referred to as advanced persistent threats), on the other hand, have a very specific victim or set of victims in mind, and likewise, carry out a very specific set of actions that may be unique to the target being exploited. A famous example of a targeted attack includes Stuxnet [31], which only activated when it was propagated to a very specific set of SCADA systems, used a highly specific means of propagation (USB flash drives, per the standard operating procedures of the intended target), and carried out malicious activities specific to the nuclear facilities that those systems were controlling. Targeted attacks by their very nature tend to utilize much stealthier approaches that often involve polymorphism to evade signatures, novel attack vectors, and zero-day exploits that are not detected by existing approaches typically offered by organizational IDS and IPS solutions.

1.2 Deception techniques and barriers to entry in enterprise network security

While enterprise networks certainly face a large number of threats due to automated attacks, the coverage for these by existing IDS/IPS approaches is typically adequate *provided everything is configured properly.* Targeted attacks, on the other hand, are a class unto their own, and traditional approaches prove inadequate for the task of detecting them. As demonstrated in the Operation Aurora [25][59] attack campaign, lapses in security policy, effective social engineering techniques, or software misconfigurations can allow for even the most simplistic targeted attack to go undetected. Furthermore, enterprise networks, due to the sensitive information and mission critical assets they host, are first in line for the types of novel, targeted attacks that can exfiltrate data, disrupt business operations, or give attackers access to computing resources (for which they have not paid nor for which they are liable) to perform illicit activities. The stealth with which these attackers can perform the targeted attacks,

furthermore, allows their activities to remain undetected possibly for months or years.

A technique that does lend itself well to trapping and observing novel attacks is deception, that is, masquerading a host or piece of content that in reality has no production value as something that is of interest to an attacker. This decoy host/content is better known as a honeypot [81]. By successfully deceiving the attacker to perform hostile actions against a monitored honeypot, the attack methods can be seen in isolation from the considerable volume of production traffic. Thus, even novel attacks can be observed by way of the state changes the attacker causes in the monitored decoy, and remediation measures can be inferred for true production assets. In spite of the efficacy of this technique against targeted attacks, organizations can seldom afford to devote capital and IT manpower to creating and maintaining a fully customized decoy strategy that covers their unique and dynamic threat landscape. Furthermore, misconfigured decoy assets can actually weaken an organization's security posture by giving the attacker a staging ground on which to perform further attacks. A turn-key approach to deception is therefore needed so that organizations will adapt the strategy of using a network of honeypots, or honeynet, to trap, observe, and mitigate novel targeted attacks.

1.3 Turn-key Honeynet Framework

To address the aforementioned issues in enterprise network security, we provide a scalable framework for turn-key honeynet deployment, which abstracts away the specifics of honeynet deployment from the specific organization. This framework must be adaptable to the form factors in which enterprise networks typically exist, namely privately maintained local networks and cloud-hosted virtual hosts. It must also stage content in a manner consistent with the production assets with which it seeks to blend decoys, and be as transparent as possible to a potential attacker. If these

requirements are met by the deceptive content, attackers will interact with a honeypot just as they would a production host of interest and reveal their tactics. The described framework consists of the *HoneyCluster*, a scalable cloud environment that hosts the honeypots, a traffic forwarding appliance that presents the honeypots as logically belonging to the enterprise subnet, attack data aggregation and analysis, and threat reporting.

1.4 Thesis Outline

The goal of this thesis is to mitigate concerns in enterprise networks as relates to a class of targeted attacks not being adequately addressed by existing security measures typically deployed in the enterprise. We present the background of this problem starting with a brief history of the origins of computer crime and its progression into the recent targeted attack campaigns that succeeded against enterprise networks for a prolonged time, usually by exploiting zero-day exploits that could not have possibly been detected by existing best practices[1]. After the discussion of those attacks that succeeded, we present a discussion of honeynet techniques that have succeeded in the past in detecting novel attacks in research domains, but that have not been deployed in enterprise environment due to onerous integration and liability concerns. Namely, these concerns include the inability to devote effort to deploy and maintain honeynet infrastructure, and the liability that would result from scatter attacks due to a misconfigured honeypot being used by an attacker to attack a third party and attribute the attack to the organization. We describe a framework that overcomes these concerns and enables seamless integration of honeynets with existing production assets to provide a layer of deception used to trap and mitigate novel targeted attacks. We present the results of the honeynet framework's deception and attack detection ability on a live network located at Georgia Tech. The contributions shown by our

[1]Indeed, these attacks were only discovered in the long run by deep manual analysis or worse still, some catastrophic event carrying with it major economic consequences

results include the effective detection of malicious remote hosts based upon behaviors encountered on our test network by dynamically allocated honeypot decoys, as well as a significant improvement in transparency of the traffic forwarding tunnel that places these honeypots logically within the enterprise subnet to be protected. Specifically, we demonstrate that our approach diverts 97.5% of traffic originating from malicious hosts away from production assets, while still allowing the malicious host to communicate with decoys under our control and serve as a source of threat intelligence. We show a significant reduction in two detectable network characteristics due to tunneling, namely packet loss (reduced from 91% to 1% at peak) and peak packet delay variation (reduced by approximately 87%). Lastly, we present conclusions and future work.

CHAPTER II

ORIGIN AND HISTORY OF THE PROBLEM

2.1 Evolution of Attack Methods

The practice of exploiting vulnerable telecommunication systems dates back at least as far as the late 1950s when Joe Engressia (later known as Joybubbles) discovered he could make free long distance telephone calls simply by whistling a 2600 Hz tone into the receiver. At the time, the AT&T switching fabric used in-band switching, which allowed end users such as Engressia to take control of the switches and place toll calls at no charge. As word spread of his exploit of the telephone switching system, the practice became known as "phreaking." Years later, an entrepreneur by the name of Al Gilbertson developed and sold a device known as the "blue box," which made Engressia's technique accessible to those who did not possess the natural ability to whistle at the perfect pitch needed to perform the exploit. In a 1971 interview with Esquire Magazine about the device, Gilbertson noted, "you've got anonymity...they don't know where you are, or where you're coming from, they don't know how you slipped into their lines...They don't even know anything illegal is going on" [76].

Forty years after the "blue box" interview, network infrastructures have grown in complexity and the modalities used to exploit them have changed significantly. However, the overarching story remains the same. Adversaries can profit greatly by gaining unauthorized control over information systems owned by others and they can remain undetected while doing so. Indeed, nowadays the rewards are quite a bit greater than having the ability to make free long distance telephone calls (although malware that performs attacks against SIP-based PBX systems can still boast of this capability). The malware industry has long been established as a cash cow for

cyber thieves, allowing them to delegate distributed denial of service attacks, mass spam distribution, network reconnaissance, and identity theft, to name a few [50]. Militarized worms coordinated via Internet Relay Chat began appearing as early as 1999 [78]. Analogous to the "blue box" that came four decades prior, the advent of sophisticated extensible bot platforms such as SDBot, GT Bot, and AgoBot by 2002 brought botnets into the mainstream [7]. In the hands of even moderately skilled adversaries, these tools paved the way for deployment and customization of extremely powerful botnets that could reap significant financial benefits. Circa 2008, the spam distribution capabilities of the Storm botnet were estimated to have brought in daily revenues between $7,000 and $9,500 to the bot masters [50]. Meanwhile, the nature of the botnet allowed the bot masters to avoid accountability for what they had done, as the spam messages themselves were sent by unsuspecting victims' computers that were drafted into the botnet [6][50]. It is worth emphasizing that because of this mechanism, the substantial cost of actually delivering the spam messages (an estimated $25,000 daily) was not incurred by the authors of the Storm botnet, but rather, collectively by the rightful owners and service providers of the compromised systems that acted as agents of the botnet.

2.2 Emergence of the Advanced Persistent Threat

Over the past several years, the most skilled adversaries in the cyber domain have begun to move away from widespread, catch-all tactics, such as huge self-propagating botnets that pump out millions of spam messages, in favor of focused attacks targeted at a specific government, industry sector, corporate entity, or individual. Attacks of this sort are more formally dubbed advanced persistent threats (APT) [24] and owe their rapid emergence to several key factors, including the following:

1. Significant will on the part of cash-rich entities and nation states to perform acts of espionage or destruction of critical data by way of the Internet [1][39][51][66].

2. Pervasiveness and sophistication of reconnaissance tools, vulnerability scanners, and customized exploit creation techniques [10].

3. Increasing reliance on information systems has caused a higher volume of sensitive information to be transacted via the Internet and stored in large data warehouses. Targeting the custodians of this information has become very lucrative for adversaries [1][14][59].

This trend has already become apparent with the emergence of a worm named Stuxnet, aimed specifically at infecting Windows computers en route to manipulating Seimens programmable logic controllers [31]. In contrast to the widely publicized worms that preceded it, Stuxnet seeks to infect very specific targets under very specific circumstances. Stuxnet is viewed by many as the most sophisticated piece of malware to be used in a targeted attack, and it is being referred to as "the first real cyber weapon" [20]. If the history of adversarial behavior serves as any indicator, Stuxnet's techniques will be mimicked and improved upon by others before long.

Another widely publicized targeted attack, dubbed "Operation Aurora" by McAfee, involved the use of a zero-day vulnerability in Internet Explorer to steal proprietary data and user account credentials from Google China and at least 20 other organizations [59] (including such information security firms as Symantec). In 2010, Damballa performed extensive analysis of the Aurora botnet and published its findings in a comprehensive report [25]. While Operation Aurora has been widely viewed by the public as a veritable APT, Damballa's findings indicate otherwise. In particular, the Aurora botnet "uses DDNS and 'old school' coordination techniques not used by sophisticated botmasters" [25]. Furthermore, the primary attack vector used to infect victims' machines was the presentation of so-called "scareware" alerts on compromised websites. This method of malware delivery is a simplistic social engineering approach and is described as follows: "The botnet controllers preyed on the fear of users that their system is infected with malware. This method saves the botnet controllers from the

technical complexity of bypassing Windows' user account control (UAC) by using the weakest link in host security" which is the user. The misled user typically clicks OK to everything, bypassing UAC and giving the malware dropper explicit permission to execute. [25]

The "simplicity and obsolescence" of the techniques used in Operation Aurora is indicative of amateur bot masters. Nevertheless, despite a lack of technical sophistication, the Aurora botnet succeeded in infiltrating the networks of several major corporations. In addition to placing Google at odds with the Chinese government [19], Aurora has reportedly resulted in the compromise of Gmail account credentials and potential spurious modifications to or theft of Google proprietary source code [42]. If Google and other prominent technology innovators are being infiltrated successfully with such coarsely devised attacks, one might expect their security postures to fare even less favorably against more determined, organized, and sophisticated adversaries. Furthermore, since Operation Aurora, other highly publicized targeted attacks have included the Wikileaks exfiltration of highly sensitive government and industrial secrets [90], large scale compromise of personally identifiable information held by Epsilon [73], and the chain of information security breaches at Sony, among others. The Advanced Persistent Threat is clearly a widespread problem with implications ranging from invasion of privacy all the way up to breach of national security and compromise of public safety.

2.3 State-of-the-Art Corporate Information Security Measures

Many organizations have invested a great deal of time and money on information security practices that are intended to keep their critical data safe from potential threats. These practices fall under the broad term *Defense-in-Depth*, which refers to the deployment of security measures at several points in the information chain so as to prevent full compromise due to a single point of failure [65]. A familiar example of

Defense-in-Depth is multi-factor authentication, which requires that the user supply multiple independent credentials, usually verified against where the user is, what the user knows, something the user has, or some biometric characteristic(s) of the user [53][60]. The password + secure token scheme, commonly used to authenticate users to corporate virtual private networks (VPN) and other secured services, is an example of two-factor authentication involving something the user knows and something the user has. In the event that the authorized user's secure token is stolen, the token itself is insufficient for the adversary to gain access, as the user's memorized password is also needed. Of course, having multi-factor authentication is effective for ensuring that a perfectly functioning authenticator will not grant access to unauthorized users. However, as the authentication scheme may also have vulnerabilities (including the authorized user's password keeping habits), Defense-in-Depth dictates that intrusion detection systems (IDS) should also be deployed within the organization.

Intrusion detection systems are grouped into two major categories, *host-based* (HIDS) [88] and *network-based* (NIDS) [75]. Organizations rely heavily on HIDS such as virus scanners and, in some cases, file integrity checkers on select systems with seldom-changing configurations. To complement the HIDS, organizations also employ NIDS techniques, usually via applications such as Snort [75]. Both of these IDS types function by comparing all observed files or network packets against signatures of known attacks and generate alerts when a match is found. The more sophisticated HIDS and NIDS are also capable of performing more in-depth heuristic analysis. In the case of HIDS, heuristic analysis is usually done by tracking anomalous activity, such as disk and memory access patterns, and comparing it to the learned behavior of prior known-malicious samples to determine whether an attack may be occurring [35][88]. In the case of NIDS, heuristic analysis is done in a variety of ways, ranging from analysis of protocol anomalies [89] to correlating NetFlow activity with that of previously observed attacks [55]. While heuristic analysis can be

more effective than direct comparison against signatures, a tradeoff exists between how computationally intensive the intrusion detection can be without creating an undue load and thus compromising the availability of the production system it is intended to protect. Heuristic analysis is also more prone to generating false positives, and must still be trained on the behavior of known-malicious samples. Completely new attack vectors that share few characteristics with previously observed ones are unlikely to be detected even by heuristic analysis. While having HIDS and NIDS deployed concurrently can strengthen a corporation's defense posture, these systems tend to work separately from one another. Unfortunately, one of malware's main characteristics of late has been its ability to evade host-based detection mechanisms, running counter to a heavy reliance on HIDS. According to the Damballa report on Operation Aurora, "[w]hile APT malware can remain stealthy at the host level, the network activity associated with remote control is more easily identified. As such, APTs are most effectively identified, contained and disrupted at the network level" [25]. Even so, malware authors are employing increasingly sophisticated techniques on the network level as well, including fast flux networks, encrypted communication sessions, and covert channels to obfuscate or conceal what data are being transacted through the course of an attack [33][36]. Intrusion detection systems that operate solely on the host level or network level are able to observe only part of the attack, and malware that obfuscates its intent does a good job of keeping these pieces of the puzzle seemingly disjoint from one another. Compound this with the immense volume of legitimate activity that passes through an IDS, and finding attacks amid the sea of business-related transactions becomes even more improbable.

2.4 Honeynets in the Research Community

The research community that focuses on analyzing information security attacks in isolation works under a different set of constraints than the information technology

departments of everyday businesses. While these researchers may be stewards of their own production networks, expected to facilitate some useful business task therewith, they are also responsible for maintaining computer systems that are purely for research purposes. Some of the primary goals of this community include finding emerging threats, analyzing them, reporting on their severity, and providing expert advice on how to remediate the applicable vulnerabilities to minimize any further harm caused by said threats. An indispensable tool of individuals in this research community has been the honeypot. In cyber-security parlance, a honeypot is a resource that has no production value, but rather exists for the sole purpose of being discovered and exploited by adversaries [67]. Similarly, when multiple honeypot systems are networked together, the resulting network is known as a honeynet. Because the services offered by a honeypot are not demanded by any legitimate user, all attempts to access the honeypot are deemed malicious.

The two main modes of operation for a honeypot are referred to as *traditional* and *client*. Traditional honeypots are typically deployed on a publicly routable network address and offer some seemingly vulnerable service that awaits an attacker's connection attempts. In contrast, a client honeypot mimics the behavior of a user agent, such as a web browser, and actively seeks out malicious payloads hosted on the Internet. Both types of honeypots are further divided into two categories, *low-interaction* and *high-interaction*, based on how fully they represent a real system. Low-interaction honeypots, sometimes also referred to as *medium-interaction*, rely on emulated versions of the services or user agents they are exposing to attacks. These emulated versions behave as closely as possible to the real software off of which they are based, but are typically re-implemented from scratch so as to intentionally expose known vulnerabilities in the service or user agent being emulated. In this way, the low-interaction honeypot is able to participate in a dialog with the attacker in such a way as to make the attack appear to have been successful. The longer attackers believe

the honeypot to be a real service, the more they will attempt to use it to further their goals, and the more tactical information they will reveal to the honeypot's maintainers. As such, a well devised low-interaction honeypot is a very powerful monitoring tool for known threats, as well as an effective way to collect malicious payloads that are distributed through known vulnerabilities. While the intent of a low-interaction honeypot is to appear as authentic as possible, only so much can reasonably be done in an emulated environment to ensure authenticity. A high-interaction honeypot, on the other hand, is a system that runs shipped versions of an operating system and other vulnerable software. Therefore, what is being exposed to attacks is a fully authentic instance of software having vulnerabilities, some of which are potentially unknown, running on a real operating system that may have vulnerabilities of its own. The high-interaction honeypot is capable of participating in and detecting attacks that leverage zero-day vulnerabilities or vulnerabilities that are not fully understood. The research focus of high-interaction honeypots, therefore, is in the effective monitoring of everything that occurs within the honeypot (which, as was stated before, is all assumed to be malicious). The advent of virtual machine introspection by way of the Intel VT and AMD-V extensions has furthered the monitoring capabilities, allowing for very fine-grained tracing of virtualized high-interaction honeypots [34]. Furthermore, containment of the attack within a high-interaction honeypot is of paramount importance. As the compromised high interaction honeypot naturally behaves just as a real infected host would, the honeypot would be just as likely to propagate the attack to other vulnerable machines if not kept under control.

Over the past decade, honeynets have been instrumental to researchers looking to gather information about botnets [6][38][57][63][86]. By being able to infiltrate multiple participants into the botnet, researchers gain the ability to monitor even the activity of botnets that have highly decentralized command-and-control mechanisms. However, in spite of their efficacy, honeynets require a substantial level of effort and

expertise to deploy, monitor, and maintain effectively. For those unfamiliar with operating a honeynet, the sheer volume of attack data can become unmanageable, and risks of infection, attack propagation, and potential legal liabilities arising from attack propagation or unauthorized distribution of electronic materials are considerable barriers to entry. The level of effort and risk involved is one of the main reasons why honeypots tend to be relegated to academic institutions and independent security-focused research groups such as the Honeynet Project [82]. Businesses are typically inclined to shy away from honeypots altogether or implement the most shallow, risk-averse honeypot deployment strategies imaginable. Some commercial off-the-shelf (COTS) products, such as KFSensor [52] and SPECTER [79] are marketed toward enterprise networks for their ease of deployment. These products tend to be deployed as standalone honeypots, of which monitoring and maintenance are the sole responsibility of the customer. Not surprisingly, many organizations do not opt to incur these responsibilities, and honeypots have thus often been cast aside with the stigma of being ineffective or simply too risky to be leveraged in a corporate setting. However, given the rise in the prevalence of targeted attacks against the enterprise, the superior attack detection capabilities of an expertly administered honeynet present a great potential addition to enterprise security strategies.

2.5 Honeynet Technologies

This section presents a synopsis of several key existing honeynet technologies that have proven to be successful in capturing novel attacks.

2.5.1 Honeywall Roo

The *Honeynet Project and Research Alliance* (commonly referred to simply as The Honeynet Project), "a leading international security research organization," has been a driving force in honeynet research over the 12 years since its inception [45]. The

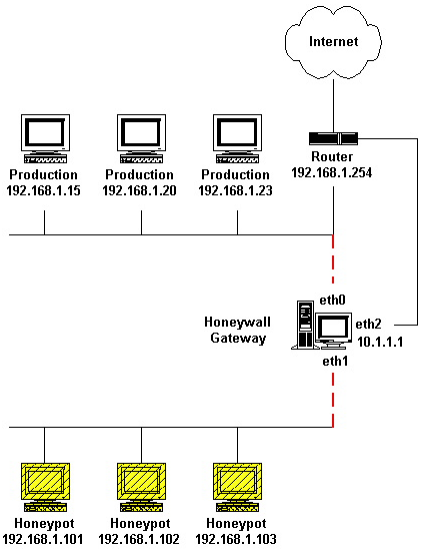

Figure 1: GenI Honeynet Architecture (originally in [85])

Honeynet Project has prescribed standards for honeynets in terms of three generations, roughly correlated to the honeynets efficacy and ease of deployment.

A first-generation, or *GenI*, honeynet is a collection of high-interaction honeypots, protected via the use of a hardened layer 2 bridging gateway known as a *Honeywall* [85]. The Honeywall is a device that performs data control, data capture, data analysis, and data collection. A diagram of the GenI honeynet architecture appears in Figure 1. The honeypot machines are isolated physically from all systems having production value by way of the Honeywall.

However, these honeypots are logically on the same network as the production systems. Second-generation, or *GenII* honeynets are functionally equivalent to their GenI counterparts, but the updated Honeywall implementation improves upon the mechanisms governing data control, data capture, and alerting of the administrator [83]. The current state-of-the-art honeynet is considered to be *GenIII*, which, in addition to all the GenII capabilities, enables a honeynet operator to administer the honeynet and view its collected data through a graphical user interface. The GenIII honeynet is facilitated by version 1.4 of the Honeywall distribution, codenamed Roo

[84]. While the current iteration of the Honeywall distribution greatly streamlines the process of setting up a honeynet, it has the following deficiencies:

1. According to the time stamps within the Trac source revision control repository, Honeywalls source code has not been modified in over two years.

2. Following from the staleness of the code base, the Linux distribution on which Honeywall Roo is based is also very outdated. The kernel version shipped with CentOS 5, and therefore Honeywall Roo, is 2.6.18, whereas the latest stable kernel version as of this writing is 3.13.2. A hardened Linux machine on the network perimeter, such as the Honeywall, should be running recent revisions of its critical software to minimize the number of known vulnerabilities that can be used to exploit it.

3. The Yum repository for Honeywall-specific packages is no longer available, thereby limiting the ability of the Honeywall Roo appliance to keep itself up to date (which is one of its advertised features).

4. The existing Honeywall configuration mechanisms lend themselves well to the small- to mid-sized deployments for which Honeywall Roo was originally envisioned. However, maintaining a cluster of logically independent honeynets beyond the order of 10 becomes a very difficult task with the existing Honeywall tool set.

5. All honeypots behind a single Honeywall Roo installation are intended to be on the same logical subnet, which exacerbates the aforementioned issue of scalability when an organization seeks to have honeypots that reside on many independent subnets.

6. While the menu-driven configuration tool simplifies the most common honeynet deployment scenarios, it does not cover all reasonable use cases. The

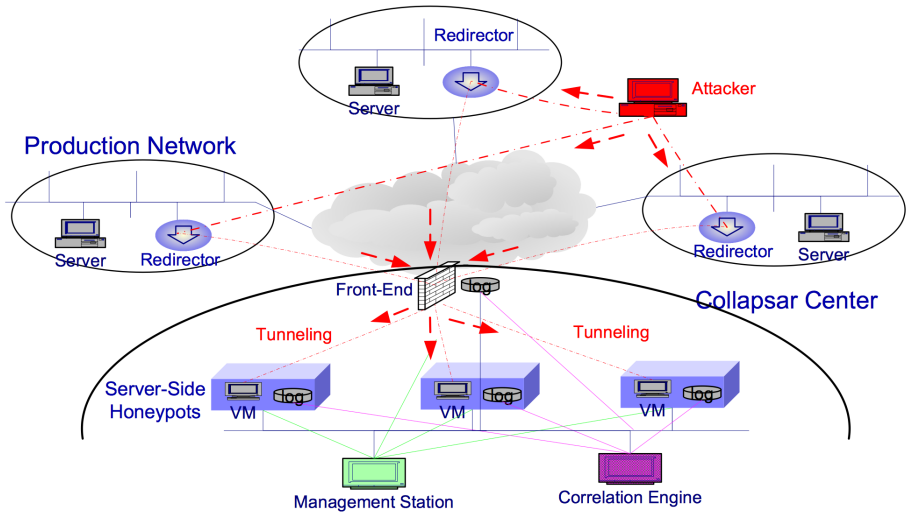

Figure 2: Collapsar honeyfarm architecture (originally in [47])

most prominent such scenario is DHCP auto-configuration of honeypots, which requires configuration above and beyond what is enabled by the bundled configuration tool.

Therefore, Honeywall Roo has room for improvement, particularly in terms of scalability to large numbers of honeynets. While many of these hurdles can be worked around with the software in its current form, doing so adds to the complexity of implementing and maintaining a honeynet, which is one of the conditions Honeywall Roo was designed to alleviate. A modern, comprehensive framework for deploying honeypots is greatly needed to pick up where Honeywall Roo left off and ease the current complexity of honeynet deployment.

2.5.2 HoneyMole

A novel honeynet concept in the 2006-2007 timeframe was that of the consolidated honeynet, or honeyfarm [26][47]. This practice is facilitated through use of a layer 2 network tunnel that originates at the point at which the honeynet is to be situated logically and terminates at the honeynet's physical point of presence.

In this way, a honeynet can appear to have nodes situated in every region of the globe while having all computational resources that host the honeynet in one centralized location. A diagram of a honeyfarm architecture is shown in Figure 2. HoneyMole was developed by the Portuguese chapter of the Honeynet Project to enable the creation of such a tunnel in a secure fashion [44]. However, this software has not been updated since February 2008. It is not straightforward to set up, nor was it suitable for long-term deployment. In addition, recent versions of OpenVPN and the Linux kernel do away with many of the conditions that originally motivated development of HoneyMole. Furthermore, the fact that OpenVPN's code base is mature, actively maintained, and used by many clients worldwide makes it a far more attractive candidate than HoneyMole.

2.5.3 Nepenthes/Dionaea

Another indispensable tool in the arsenal of the honeynet researcher has been Nepenthes [64], a modular low-interaction honeypot toolkit that enables detailed capture and analysis of known attacks by emulating vulnerable network services. Nepenthes' power lies not only in the modules included with the toolkit, but also in the fact that researchers can tailor their own modules to their specific needs. However, as Nepenthes modules are to be written in C++, many researchers who are not proficient in the language found difficulty in developing their own extensions. For this reason, Dionaea [17], a successor to Nepenthes, was developed in the summer of 2009. Dionaea allows for its extension modules to be written in Python, a language with a shallower learning curve than that of C++. Dionaea also has support for IPv6, as well as detection of malicious shell code by way of libemu [58]. Dionaea addresses the aggregation of data from multiple sensors by offering a back-end XMPP data reporting channel. As of this writing, Dionaea remains in active development under the auspices of carnivore.it [16]. As Dionaea is itself a framework for low interaction

honeypots, it can be extended to include virtually any service or protocol a security researcher can imagine (and that has a reference for implementation).

2.5.4 Kippo

Kippo is a "medium interaction SSH honeypot designed to log brute force attacks and, most importantly, the entire shell interaction performed by the attacker" [54]. It binds itself on port 2222/tcp, rather than the standard port 22/tcp, so that it does not need to run with root privilege. A Kippo instance will log all activity performed against it, namely username/password combinations and, if successfully breached, all commands run upon it by an attacker. It performs an additional level of deception in that when the attacker logs out or issues an EOF character, the shell prompt switches to give the appearance of a root shell on localhost. In this way, the Kippo honeypot can collect information about what an intruder might be trying on his own system, while running the higher risk of being detected by the intruder. Kippo's main shortcomings lie in the emulation techniques it uses that increase an attacker's ability to identify it as a honeypot. An article entitled *Attacking Kippo* was posted to the Alert Security blog [13] that outlines some of the methods that can be used to identify Kippo. These techniques include passing a command on the original SSH command-line ala "ssh user@host command" which gives rise to a NotImplementedException. Kippo's lack of persistence of certain things such as newly added users can also be used to determine it is a honeypot and not a real system. In other words, adversaries can use the shortcomings in Kippo's partial emulation of an SSH daemon to determine it is performing such emulation. These specific issues can be addressed simply enough, but the fact remains that in any low-interaction honeypot, the farce will ultimately show itself and transitioning the attacker to a high-interaction approach is required to make the honeypot continue to appear genuine.

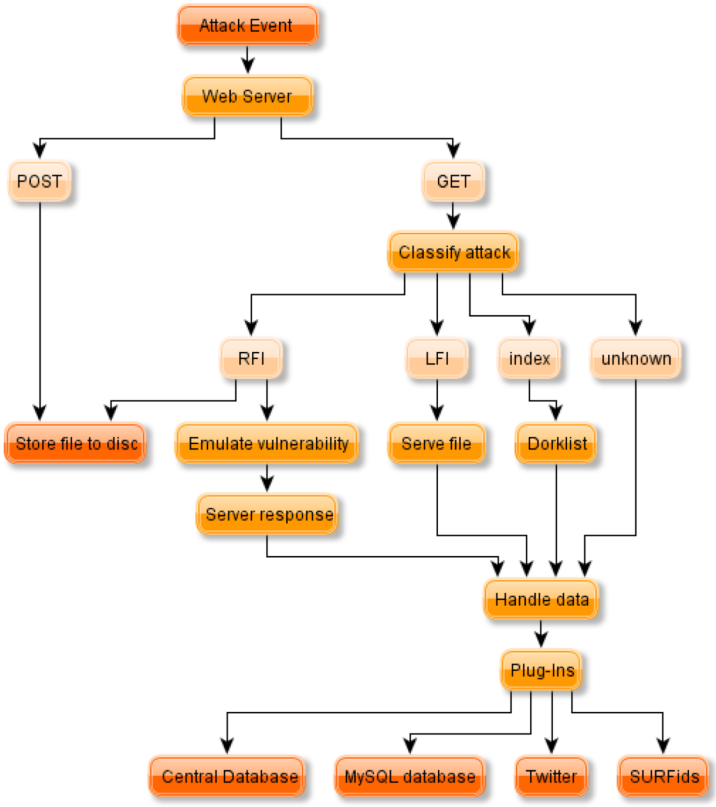

Figure 3: Flowchart of how an attack gets handled by Glastopf (originally in [71])

2.5.5 Glastopf

Glastopf is a low-interaction "honeypot which emulates thousands of vulnerabilities to gather data from attacks targeting web applications" [37]. It accomplishes this by masquerading as hosting a list of notoriously vulnerable script paths (referred to also as *dorks*). It publishes the availability of these paths to search engine crawlers, which then index and include them in search results that attackers collect as they search for these vulnerable paths. Once an attacker reaches the Glastopf honeypot, he will attempt the published paths and likely some that were not listed in the search engine index. When these new paths are detected, they are added to the corpus of vulnerable paths that get advertised, thereby attracting a new wave of attackers looking specifically for the new paths. Therefore, the longer a Glastopf honeypot has been running, the larger the attack surface it advertises and the wider the range of attacks it will attract. A flow-chart showing Glastopf's attack handling is shown in Figure 3.

2.6 Enterprise Cloud Computing Trends and Security Implications

In 2006, Amazon started the mainstream shift of information technology (IT) assets toward cloud computing when it introduced the Elastic Compute Cloud [30]. According to a study by Gartner, approximately 38% of all enterprises use some form of public cloud infrastructure, and that number is set to increase to 80% by 2015 [72]. Enterprises have incorporated cloud computing into their infrastructures largely because it allows them to acquire massive levels of computing resources without having to devote dedicated space, purchase and maintain computer hardware, and acquire adequate on-site network bandwidth as they would have when running their own data centers in house. Furthermore, the virtualization techniques employed by cloud infrastructures allow infrastructure to be provisioned to large scale, elastically, and

without rigid constraints upon network topology, allowing for more flexibility than the static constraints imposed by hosting everything their own iron. Cloud services succeed at commoditizing computational availability, and abstracting many of the hard problems of IT management away from the enterprise.

The flexibility of the cloud also makes it a highly advantageous place to employ the strategy of deception. The scalability and elasticity of typical cloud services provides computational head room for ephemeral, non-production assets (e.g. decoy hosts) that can be provisioned when production demand on the platform is below saturation, and de-provisioned so as to free resources as production demand requires. Furthermore, given the dynamism and interchangeability of individual hosts within the cloud, intruders will be able to infer far less from the way the network is structured (aside from domain name service records), and their task of reconnaissance is thus made more difficult, necessitating more aggressive techniques. The fact that the allocation of hosts on the network can and does change in very short intervals further implies that the relevance of the attacker's aggressive reconnaissance is more short-lived than in traditional network deployments. For these reasons, targeted attacks against cloud-hosted assets can be made considerably more difficult to perform without inside knowledge of the network asset allocation and provisioning strategies, and an attacker is very likely to encounter a decoy in his search for production assets.

In addition to the considerable difficulty of targeted attacks against a cloud computing environment, the threat of automated attacks (e.g. botnets) also remains a very real one [21]. While an intruder may not have knowledge of the specific allocations of hosts on a cloud's public subnet, it can reasonably infer that the subnet has targets of high value. Depending upon the cloud provider's policies, the hosted services likely have heterogeneous security measures, meaning the adversary may find a weak link that gains them access to considerable computing, storage, and network resources. For this reason, the subnets of cloud providers are highly attractive to

automated attacks [21].

In the cases of Intrastructure-as-a-service (IaaS) [3] and Platform-as-a-service (PaaS) [12] cloud models, the security of the hosted assets is typically the responsibility of the customer. That is, the enterprise must implement its own security policy on any cloud assets thus hosted. Several of the leading public cloud providers provide or offer the option of enabling built-in security measures such as traffic filtering, traditional IDS, client browser integrity checking, and visitor reputation whitelisting/blacklisting [2][9][23][48][61][68][77]. It is worth noting that each provider offers slightly different sets of security features, meaning that the customer must self-implement those required measures that are not offered on their particular hosting provider. Many organizations even prefer to have the control over their own security, rather than rely on the hosting provider to implement security measures for them [18]. However, this strategy results in a heterogeneous set of security implementations in a particular cloud deployment, which can be difficult to audit effectively.

A recent proposal to cloud security best practice is known as the software defined perimeter (SDP) [11], which "aims to give application owners the ability to deploy perimeter functionality where needed." The main objective of the SDP is, similar to the traditional fixed perimeter model, to hide internal assets and disallow external users from accessing them. In contrast to the fixed perimeter, SDP allows for finer grained control of the logical perimeter, allowing it to exclude untrusted devices that move into the physical enterprise (e.g. bring-your-own-device) as well as include trusted assets that reside outside (e.g. cloud-hosted services).

2.7 Clustering of Connection and Activity Features for Malicious Network Traffic Detection

A framework known as BotMiner [41] has been presented that very effectively identifies the behaviors of malicious bots by passively monitoring network traffic at the perimeter of an existing subnet. BotMiner collects connection information in the

form of network flows to determine who is talking to whom on TCP and UDP protocols. This network flow data is referred to as the C-plane and contains the following features:

- time
- duration
- source IP
- source port
- destination port
- number of packets
- number of bytes

In addition to the C-plane features, BotMiner also collects information about activity, or who is doing what. This activity information is dubbed the A-plane and consists of a variety of features, including

- Snort alerts
- Scanning behavior (as detected by SCADE framework)
  - Abnormally-high scan rate
  - Weighted failed connection rate
- Anomalous amounts of DNS MX queries and initiated SMTP traffic (indicative of SPAM)
- Portable Executable (PE) binary downloading

From these passive operations of C-plane and A-plane, BotMiner determines which hosts on the subnet are likely compromised, and to which botnet they may have been enlisted. BotMiner performs X-means clustering on the C-plane and A-plane, and by performing cross-plane correlation between these two, the creators of BotMiner were able to achieve 100% detection of six of the eight botnets that had infiltrated hosts on the network under test, with detection rates of 99.6% and 75% on the other two. This approach is very effective for botnet detection, but the authors acknowledge several areas of their work that could use improvement, namely that their "A-plane clustering implementation utilizes relatively weak cluster features" and that the observed A-plane activities were not exclusive to botnets, thus the possibility of generating "a lot of false positives." Therefore, the creators of the BotMiner framework hypothesize that its efficacy would be improved significantly by a richer source of A-plane features and a way to filter out the majority of benign traffic. These are two objectives in which honeypots excel.

Our proposed solution to novel, targeted attack detection operates on this hypothesis. By using honeypots to provide active feature detection and thus augment the A-plane significantly, the approach used by BotMiner can be adapted to determining which remote hosts are performing malicious activity toward the subnet being monitored. Furthermore, the determination of malice can be arrived at with more confidence, as the only remote hosts that will actually be *generating* features are those who are accessing decoys for which there is obviously no legitimate production use. In arriving at this determination, a potentially vast amount of attack data are attributed to the remote host being deemed malicious, giving not only a classification, but a thorough insight into the attacker's behavior pattern. Therefore, much more comprehensive remediation measures can be devised in response to the threat.

2.8 Related Cloud-based Honeynet Work

A small amount of prior work exists involving honeynets within cloud computing architecture. One of these works leverages the detection capabilities of honeypots to characterize attack patterns on Amazon EC2, Microsoft Azure, IBM SmartCloud, and ElasticHosts public cloud infrastructures [15]. The work focuses mainly on where attacks originate, the kinds of attacks that are made, and what differences exist across cloud providers.

Another publication models honeypots as an anomaly detection system for integration with cloud based IDS/IPS systems. It presents "the performance analysis of attack detection on the number of 800 malicious packets" [5] of honeypots vs. traditional IDS/IPS and shows a 1% improvement.

Lastly, an approach for fast dynamic extracted honeypots in cloud computing is presented [8]. Within, a framework is described that dynamically clones high-interaction honeypots from a running cloud VM instance when an attack is detected. The method relies upon metrics (effectively, signatures) derived from a static capture of attacks against a dark net. If a connection matches one of the signatures, the production host is cloned within 3-7 seconds and the malicious connection is handed off to the cloned VM for deeper inspection of the attacker's behavior. The technique presented is effective at quickly cloning the production machine, but has a few drawbacks.

- Relies upon the Xen hypervisor, reducing portability to other virtual machine architectures.
- Uses static rules gathered in past to identify attackers $\rightarrow$ effectively signature based.
- Monitoring begins after attack against a production host is underway, meaning that the attack must be observed effectively amid production traffic.

- Handoff of established connection to a new VM introduces possible identifiable characteristics.

CHAPTER III

SCALABLE FRAMEWORK FOR TURN-KEY HONEYNET DEPLOYMENT

The turn-key honeynet framework consists of four main architectural components. These are the HoneyCluster framework, which is responsible for provisioning the honeypot assets, the remote sensor appliance that injects the honeypots resident on the HoneyCluster into the logical subnet of the enterprise, attack data aggregation and presentation, which serves as the attack data storage and analysis engine of the framework, and data reporting and visualization, which presents the threat intelligence in a manner to be consumed by security practitioners and organizational decision-makers.

Detailed explanations of these components are presented in Section 3.1 and a detailed discussion of the methodologies used to test them in Section 3.2.

3.1 System Architecture

This section presents an overview of the implementation details that comprise the turn-key honeynet framework.

3.1.1 HoneyCluster Framework

Our implementation of the HoneyCluster framework is based upon Canonical's Juju [49] service orchestration framework. Juju was chosen for its ability to interface with nearly all existing cloud APIs as well as for its extensibility via service modules known as *charms*. We have implemented our own custom charms that enable reliable deployment of decoy assets within this environment. For the purposes of our proof-of-concept prototype, our custom charms enable dynamic deployment of the following

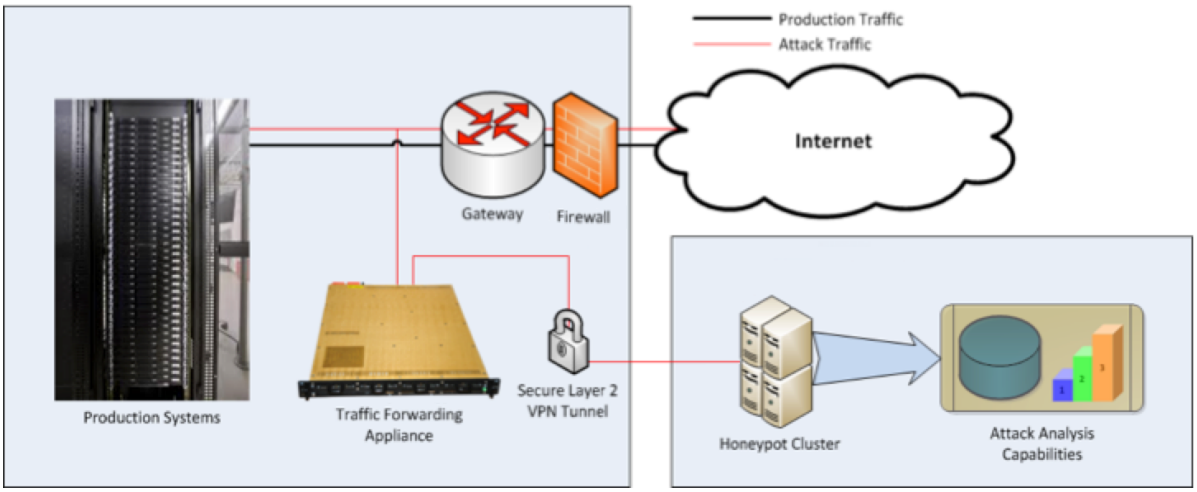

Figure 4: Enterprise honeynet architecture including traffic forwarder appliance

low-interaction (a.k.a. medium-interaction) honeypots:

- Glastopf [37]
- Dionaea [17]
- Kippo [54]

In addition to these honeypots, we have the ability to shadow virtually any production service with a decoy either by using the existing library of Juju charms or writing our own.

For security reasons, a wall of separation must exist between production and honeypot assets. Effectively, communications between the two types of assets are disallowed so that honeypots compromised by an attacker cannot be used to stage further attacks against production hosts.

While it is possible to pre-define statically what is deployed within the decoy environment, the novelty of our strategy comes in its ability to blend in with the enterprise network, whether it is deployed on-site or dynamically in the cloud. For the scope of this work, an automatic deployment strategy was implemented that surveys the N most frequently accessed decoy-eligible (e.g., unused) public subnet

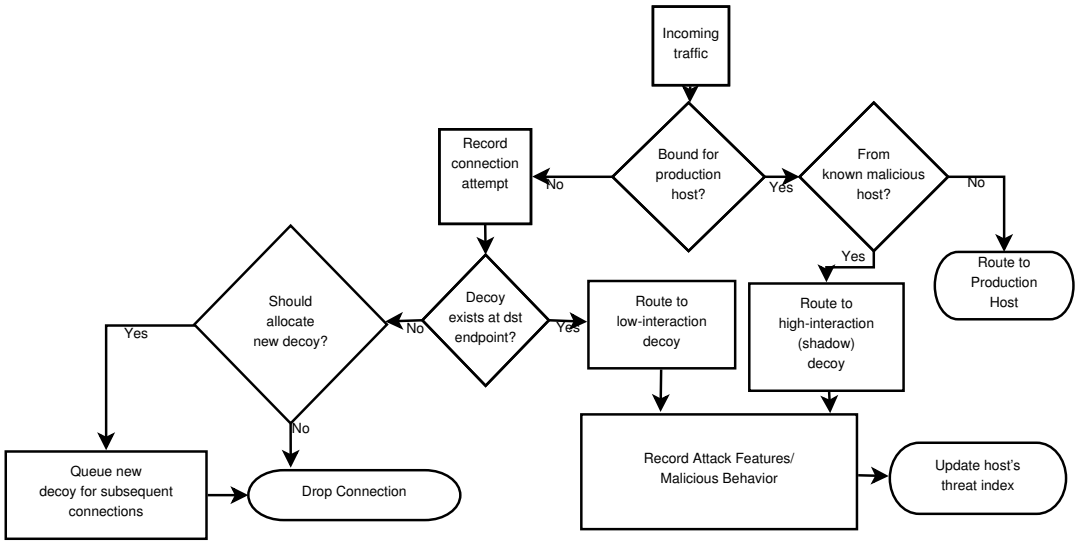

Figure 5: Flow Diagram of Incoming Traffic Handling

endpoints and responds by automatically provisioning an applicable decoy at this network endpoint. For instance, if our enterprise network is assigned to the subnet 3.4.5.0/24 and we observe a comparatively large volume of SYN/RST interactions with 3.4.5.158 on tcp/22, we are well served to deploy a Kippo (SSH) honeypot on this endpoint to gather more intelligence on the nature of these connection attempts. The value of N is configurable and based upon the size of the subnet in question, as well as the extent to which resources in the HoneyCluster need to be economized. In our implementation, we devoted 20% of the unused endpoints[1] toward decoys by default.

Once deployed within the HoneyCluster via Juju, honeynet assets are made available on the public network via a configurable firewall based upon iptables. A high level synopsis of the logic by which incoming connections are routed either to production or the decoy infrastructure is shown in Figure 5. The firewall control functionality is orchestrated by a Python library named *iptc*, but for familiarity of syntax, the following description presents the equivalent commands as though they were passed to

[1]The total address space was an IPv4 /25 subnet

iptables on the command line. For the implementation of this logic at the perimeter, firewall rules are modified automatically to expose provisioned decoy assets as though they existed on the production net. In the majority of low interaction deployments, Juju is directed via its *deploy* or *add-unit* functionality to provision a container that automatically initializes and starts an instance of the low-interaction decoy of choice. Once the provisioning process completes and the decoy is active, iptables commands equivalent to

```
1 iptables -t nat -A PREROUTING -d [ExternalIP]/32 -p tcp \
2         -m tcp --dport [ExternalServicePort] -j DNAT \
3         --to-destination [InternalIp]:[InternalServicePort]
4 iptables -t nat -A POSTROUTING -s [InternalIP]/32 \
5         -o [NetInterface] -j SNAT --to-source [ExternalIP]
```

are used to expose the honeypot to the public subnet. Kippo honeypots are a special case in that they bind internally to port 2222 to avoid needing root privileges on the container, so in the above ruleset, ExternalServicePort=22 and InternalServicePort=2222.

In addition, production assets themselves can optionally be shadowed by a decoy, to which traffic identified as originating from malicious hosts to the production asset is redirected. In similar fashion to the low-interaction case, containers are provisioned via Juju deploy or add-unit, ideally according to the same configuration used to deploy the production host, but without sensitive data such as private keys or other proprietary information. The idea is to make the shadow decoy as convincing a replica as possible without putting any real organizational asset at risk of compromise. Once the high interaction replica decoy is provisioned, the resulting iptables rules can vary. In general, filtering rules that match those of the real production asset should be duplicated, but in many cases, these will be captured as part of the in-guest configuration and outside the scope of our control. The part that our framework

is directly responsible for is redirecting malicious hosts away from the production assets and onto the high-interaction shadow decoy. This redirection is accomplished by establishing the following set of iptables rules:

```
1  iptables -t filter -A FORWARD -d [InternalProductionIP] \
2           -j ACCEPT
3  iptables -t filter -A FORWARD -d [PrivateDecoyIP] -j ACCEPT
4  iptables -t filter -A FORWARD -s [ExternalIP] -j ACCEPT
5  iptables -t nat -A PREROUTING -d [ExternalIP] \
6           -i [NetInterface] -j DNAT \
7           --to-destination [InternalProductionIP]
8  iptables -t nat -A HOSTILE -d [ExternalIP] -i [NetInterface] \
9           -j DNAT --to-destination [InternalDecoyIP]
10 iptables -t nat -A POSTROUTING -s [InternalProductionIP] \
11          -o [NetInterface] -j SNAT --to-source [ExternalIP]
12 iptables -t nat -A POSTROUTING -s [InternalDecoyIP] \
13          -o [NetInterface] -j SNAT --to-source [ExternalIP]
```

In the preceding ruleset, lines 1-4 allow forwarding of all traffic bound for the internal production and decoy IPs and from the external IP. The rule found on lines 5-7 specifies an inbound 1:1 dynamic network address translation rule to allow traffic to reach the production host. The rule on lines 8-9 specifies a 1:1 NAT translation from incoming traffic matching the HOSTILE chain (to be elaborated upon in the subsequent paragraph) to the high-interaction shadow decoy. Lines 10-11 allow outgoing traffic from the production asset to be routed outward with src=ExternalIP. Lines 12-13 allow packets originating from the high-interaction decoy to be routed outward with src=ExternalIP.

The HOSTILE chain is used to mark hosts that have been determined by their activity to be malicious. Entries are added to this chain automatically when their threat index exceeds the threshold by inserting the following iptables rule:

```
1  iptables -t nat -A PREROUTING -s [HostileIP] -j HOSTILE
```

Suppose we encounter a remote host that has performed enough suspicious activities on our decoys to amass a sufficiently high threat index. Upon passing the threshold, the system automatically blacklists the offending host by inserting a rule like the above for the host in question. Packets originating from that host would then be picked up by the above PREROUTING rule and placed on the HOSTILE chain. Rather than having the packets forwarded to the real production asset, they will be sent to the decoy and the production host will thus be protected from further malicious activity.

The intrinsic benefit to this technique of redirecting malicious traffic to decoy honeypots in the way described above is that the decoy assets carry out no production goal whatsoever, so any compromise that may occur does not endanger assets of real value. Furthermore, rather than simply denying the traffic from passing the firewall, and therefore gaining no further knowledge from it, these attack interactions are allowed to proceed against value-less assets that gather behavioral information. For more information on how the interactions are used to determine suspicion level of a particular host, please see section 3.1.3 "Attack Data Aggregation and Presentation."

3.1.2 Remote Sensor Appliance

Complementing the HoneyCluster is the remote sensor appliance that is to be deployed on each remote subnet participating in the honeynet. This component performs the logical injection of honeynets onto private, on-site enterprise networks by forwarding traffic from the HoneyCluster to the site at which the honeypots are to appear on the logical network. This component is required to have transparency and reliability.

To make deployment of new tunnels as simple as possible, we devised an automated installation mechanism for the remote tunnel endpoint. We settled on Arch Linux as the distribution of choice because it has a very flexible installation framework, AIF,

which allows for rapid creation of customized installation procedures. Its software packages are also updated very frequently, which is of paramount importance when creating a hardened network appliance that will be deployed on the network perimeter. Another advantage of AIF is that it allows for its installation scripts to be referred to by uniform resource identifiers (URI), meaning the installation procedures and configuration data can be retrieved from a centralized source and customized on a per-sensor basis.

We developed an ISO image that leverages AIF to perform installation and configuration of a remote tunnel endpoint with minimal input. The user performing the installation need only specify the network interface configuration and a secure password. We then deployed one of these turn-key forwarders in a virtual machine (VM) situated on a publicly routable subnet and established a tunnel between it and a VPN endpoint at Georgia Tech Research Institute (GTRI), approximately one mile away. Once the tunnel was established, we bridged a Windows XP SP2 VM on a machine physically located at GTRI to the remote network. It acquired a DHCP lease and took on an IP address as if it were connected directly to the network at the remote location. This tunnel was operational between October 2010 until February 2012, with very low (<1%) downtime. The slight downtime that was observed is attributed to planned network maintenance of the hosting assets and sporadic power outages rather than fundamental malfunctioning of the VPN tunnel itself.

In addition to the reliability of the tunnel, its performance characteristics are also of great significance. Namely, the impact of the tunnel on packet round trip times and throughput must be acceptable. To draw conclusions about the viability of the tunneled connection, we measured the throughput and packet delay variation (PDV, also known less formally as packet jitter) as a function of stream bandwidth, both on a native inter-site link as well as over an OpenVPN tunnel encapsulated on the same link. In addition, we quantified the average round trip time over the link in both

native and tunneled modes of operation. Having adequate connection throughput is of paramount importance. However, the observed packet jitter and round trip times are also of subtle importance. The existence of an easily measurable systematic anomaly in these properties of the connection may well give an attacker a means to fingerprint the overall honeynet setup as out-of-sorts with the rest of the network. This, in turn, would likely cause illicit activity to be diverted from the honeynet.

Network interfaces in Linux are configured by default with the *pfifo_fast* classless queueing discipline and impose no rate limitation upon the interface, instead, relying upon transport protocols at Layer 4 to govern themselves via flow control. However, due to well known challenges of tunneling TCP over TCP [43], this is not a viable approach for VPNs, and the use of UDP therefore is highly preferable. The result of this lack of rate limitation, as will be demonstrated in the results chapter, is a very significant degradation in throughput and a large spike in packet delay variation.

An alternative queueing discipline in Linux that allows for traffic shaping, whereby a maximum transfer rate can be specified, is the token bucket filter (TBF) [56]. The kernel assigns a token bucket whose size depends upon the target transfer rate, and tokens are replenished periodically. As long as the bucket has available tokens, a byte can be scheduled for transmission, and once the tokens are depleted, subsequent byte transmissions are deferred until such time as the tokens have been replenished. The deferment behavior of the TBF queue discipline is effective at limiting bandwidth of the flows, but can have adverse effects upon packet delay variation and round trip time.

A classful extension to TBF is hierarchical token bucket (HTB) filter [56], which allows more flexibility in defining base and ceiling rates. Our shaping policy for the traffic sent over the VPN is thus based upon HTB, and applied to the tap interfaces of both the VPN endpoints, in order to prevent the CPU saturation that gives rise to the undesirable networking characteristics.

3.1.3 Attack Data Aggregation and Presentation

In a live setting, Honeynets generate large volumes of data that cannot be comprehended without imposing some sort of structure to aid automated processing. Our framework collects data from a wide range of sources, including many different low-interaction honeypot sensors, system logs, network captures, and external intelligence sources such as GeoIP and malware analysis tools, to name a few. Therefore, we are required to maintain a coherent view of all these different data while allowing extensibility to new sources of data. To achieve this, we opted to use a document storage database for its flexibility and ease of migration in the event of schema changes. MongoDB [62] has been used to good effect in many large-scale instances while maintaining acceptable performance. It also has a very familiar JSON-like API and client bindings for nearly every compiled and interpreted language in existence, which enables integration with existing technologies.

Each of the deployed honeypot sensors reports behavioral information about attacking hosts on an event-by-event basis, by inserting it into its respective document storage collection in MongoDB. This behavioral information is then used to build a set of attack features for each host,

$$[f_1..f_N]$$

(see Appendix A for schemas and examples of how sensor data is translated to features). Hosts are then placed in groups based upon the features generated on their behalf. Their relation to other hosts are determined by the Jaccard similarity coefficient [69] between their feature sets, defined as

$$\frac{|A \bigcap B|}{|A \bigcup B|} \; where \, A \equiv [f_{a1}..f_{aM}] \;\; and \, B \equiv [f_{b1}..f_{bN}]$$

Two hosts having a Jaccard index of similarity 1.0 are thus placed in the same group, and two hosts with Jaccard index of 0.0 share no features in common (e.g.

Decoy Type	Feature Type	Threat Index
dionaea	accept (httpd)	0.1
	accept (ftpd, mssqld, mysqld)	0.5
	accept (smbd, epmapper)	1.0
	profile	1.0
	mysql_command	1.0
	dcerpcrequest	2.5
	dcerpcbind	2.5
	offer	2.5
	reject	0.1
	login	2.5
	mssql_fingerprint	1.0
	mssql_command	1.0
	downloads	2.5
	service	2.5
	listen	2.5
	connect	1.0
	sip_command (cmd)	2.5
	sip_command (addr, via)	0.5
glastopf	request_raw	0.1
	request	0.1
kippo	login_succeeded	5.0
	terminal_size	0.5
	lost_connection	0.5
	client_version	0.5
	login_failed	1.0

Table 1: Threat Index Contributions of Common Attack Features

the intersection of their feature sets is the empty set). From these groupings and similarity metrics, we form associations among hosts by the nature of the attack(s) performed.

In addition to the similarity metrics, we compute a composite threat index for each feature set by computing a weighted sum of the threat indices corresponding to a host's observed attack behavior as follows:

$$ThreatIndex = \sum_{i=1}^{N} T_{fi} * f_i \ (T_{fi} \ derived \ according \ to \ Table \ 1)$$

The threat index calculated from a given feature set is imputed to all the adversary hosts that are members of the group defined by that feature set. The attack behavior features that our framework prototype gathers are provided mainly by the low-interaction honeypots, Dionaea, Kippo, and Glastopf. Among these low-interaction honeypot sensor features, Dionaea provides the most diverse features as it emulates a multitude of vulnerable services against which an attacker can launch exploits. Dionaea "reject" events are generated by a remote host's attempt to establish a connection on a port Dionaea is not monitoring (e.g. connections on Microsoft RDP, port 3389/tcp). These reject events are usually indicative of scanning behavior by a remote host and are individually assigned a comparatively low suspicion score. The reasoning behind this low contribution is that individually probing a closed port is not in itself a high-risk action, and that aggressive scans (a truly suspect activity) will amass a sufficiently large number of reject events to have a significant impact on the overall threat score. Dionaea "accept" events represent, as one would expect, a successful connection to one of the services emulated by Dionaea. These cast a varying degree of suspicion on the connecting host, as httpd accepts may be generated due to an innocuous web crawler, whereas spurious connections to a MS SQL or Samba server are considerably less likely to be the result of something legitimate or benign. A moderate level of suspicion is ascribed to post-connection events such as mssql_command, mssql_fingerprint, mysql_command, as we are running decoy instances of these services on which requests are being serviced, thus there is no content to be requested by a legitimate user. "Profile" events are generated by the detection of shellcode within a payload provided by a particular host, and these events are scored as highly suspect given there is no legitimate reason for this activity. Dionaea includes emulation of the Distributed Computing Environment/Remote Procedure Call (dcerpc) protocol used by Microsoft Windows for remote administration and has been a very successful attack vector in the past. Therefore, any attempts to bind to

or make requests against this service are viewed as highly suspect.

The threat indices corresponding to each feature are summarized in Table 1. Once this threat index reaches a sufficiently high value (in our case, 50.0 is the threshold), subsequent traffic from that host is automatically diverted away from production assets and toward shadow decoys. The effectiveness of this traffic redirection is discussed in detail in the results chapter.

3.1.4 Data Reporting and Visualization

For reporting and visualization, we implemented a RESTful JSON API that interfaces with a rudimentary web page implemented in Ember.js [29]. We relied on a package known as Ember Charts [28] to interface with the RESTful API and perform the actual charting of data.

3.2 Experimental Testbed Setup

This section details the testing methodologies used to evaluate the performance of key architectural features of the framework.

3.2.1 Attack Detection Performance

We implemented prototypes of our HoneyCluster and Data Aggregation components in Python and deployed them within a local Linux containers (LXC) environment managed by the Juju framework. We also hosted example production assets on an unfiltered /25 public IPv4 subnet on the Georgia Tech ECE network. In this way, the production assets were subjected to live traffic, and therefore typical threats faced by internet-facing services on real networks. To represent legitimate network traffic, we carried out requests to each of the production assets according to the expected use cases of the content being hosted. To verify basic functionality of the system, as well as provide a ground truth of its detection capabilities, we subjected the system to a red teaming exercise, generating malicious traffic by using the *nmap* and *metasploit* tools

Service	Honeypot type	Decoys Allocated
smbd	Dionaea	32
mssqld	Dionaea	32
mysqld	Dionaea	32
dcerpc	Dionaea	32
sip	Dionaea	32
ftpd	Dionaea	32
sshd	Kippo	12
httpd	Glastopf	17

Table 2: Decoys Automatically Provisioned in Response to Live Attack Traffic

in the BackTrack5 R3 penetration testing suite. This probing and reconnaissance behavior was detected by the system, triggering automatic honeypot provisioning and malicious feature detection in line with the actions performed in the red teaming exercise.

For the purposes of the experiments, our automatic decoy provisioning policy was to select the N most accessed address/port combinations on the unused portions of the subnet. In our IPv4 /25 public address block (128 total IPs), we allocated 10 test machines that served as production assets. For simplicity and because we had such a large proportion of unused IPs in the subnet, we opted to reserve the entire port space of a production host as unavailable for honeypot provisioning. Therefore, subtracting these hosts as well as the network, router and broadcast addresses from the total allocation left us with 115 fully unused addresses that were available for honeypot deployment[2]. To fit in the envelope of computing hardware that was available for decoys, we opted for a target of 20% utilization of the IP space, giving us room for about 23 decoys per service. After approximately 30 days in live operation, the framework had automatically allocated decoys as shown in Table 2.

[2]In a true production scenario, we would expect to have many more production hosts, but we would also have the option to allocate the unused ports on the production assets toward honeypots.

3.2.2 Transparency of Remote Sensor Appliance

We deployed two OpenVPN endpoints approximately one mile apart. One of the nodes was on a residential connection, while the other was deployed at Georgia Tech Research Institute. These two nodes were separated by eleven network hops. We instantiated each VPN endpoint on Arch Linux within a VMWare virtual machine[3], which we gave 512 MB of memory, 8 GB of thin-provisioned disk space and two network interfaces. Each physical host had a four-core Intel Core 2 CPU. As was optimal for our hardware and network configuration, we configured each VPN endpoint to use Blowfish encryption, communicate over UDP, and encapsulate Layer 2 bridged traffic. In this way we established a channel for honeypot sensors physically located at GTRI to be joined to, and appear as belonging to, the residential subnet a mile away.

To measure baseline throughput and packet delay variation, we used a pair of *iperf* [46] instances to transact bidirectional constant bit rate UDP streams of bandwidths ranging from 1 to 65 megabits per second. We ran the iperf instances in Windows XP SP2 virtual machines, separate from the VPN endpoints so as to minimize contention for memory and CPU resources. For measurements involving the tunneled links, we connected the iperf instances to one another through the VPN by bridging their VMs' network interfaces to those of the VPN endpoints. For measurements involving the native links, we assigned public IPs to the measurement VMs on their respective subnets and allowed them to communicate openly via the Internet. With these two configurations, we were able to compare the performance of the inter-site VPN tunnel with that of the native link, which is representative of the connectivity of assets on the real enterprise network.

Once we recorded the baseline measurements for throughput, PDV, and RTT,

[3]A virtual machine environment was used due to our implementation requirements to host multiple endpoints on one physical host, but these results can be replicated in either a virtualized or bare metal environment.

we applied our classful HTB queueing discipline, with rate and ceiling parameters of 10 megabits/second, to the network tap interfaces of each VPN endpoint[4]. We then repeated the steps described above to measure throughput and PDV characteristics of the VPN tunnel under the HTB queueing discipline.

[4]We chose 10 megabits as a conservative value, but any maximum bandwidth that does not saturate the CPU will have the desired effect.

CHAPTER IV

RESULTS

This chapter presents the results derived from the testing methodologies presented in Section 3.2. In Section 4.1, the framework's ability to detect attacks is detailed and in Section 4.2, the metrics in which the transparency of the traffic forwarding appliance are presented.

4.1 Attack Detection Performance

After allowing the active deception system to remain exposed to active threats on our public subnet for approximately 1 month (Dec 2013), our decoy infrastructure encountered connections from 6,739 unique remote hosts, of which 6,569 had available IP Geolocation data. The majority of these attacking hosts were located in the top 5 countries, namely China (1,930 hosts or 29.3%), United States (803 hosts or 12.2%), Russian Federation (428 hosts or 6.5%), Brazil (301 hosts or 4.6%), and India (222 hosts or 3.4%). A chart of countries that represented greater than 1% of the total malicious host set is shown in Figure 6.

The overall set of observed attack behaviors fell into 537 distinct groupings. These groupings ranged from having 1 feature to having over 800. Among the 193 one-feature (singleton) groups, all the behavior features involved an accepted or rejected connection attempt on a particular port, indicating a single-port probe. Due to this characteristic, the singleton group class is representative of the intelligence gathered by logging disallowed connections at the perimeter. The single-port scanners represented 5,703 of the encountered hosts (84.6% of the total). Of the single port scanners, 4,882 hosts (or 85.6%) amassed a very low threat index under 1.0. This low threat index indicates a non-determined attacker that has not presented enough of a threat

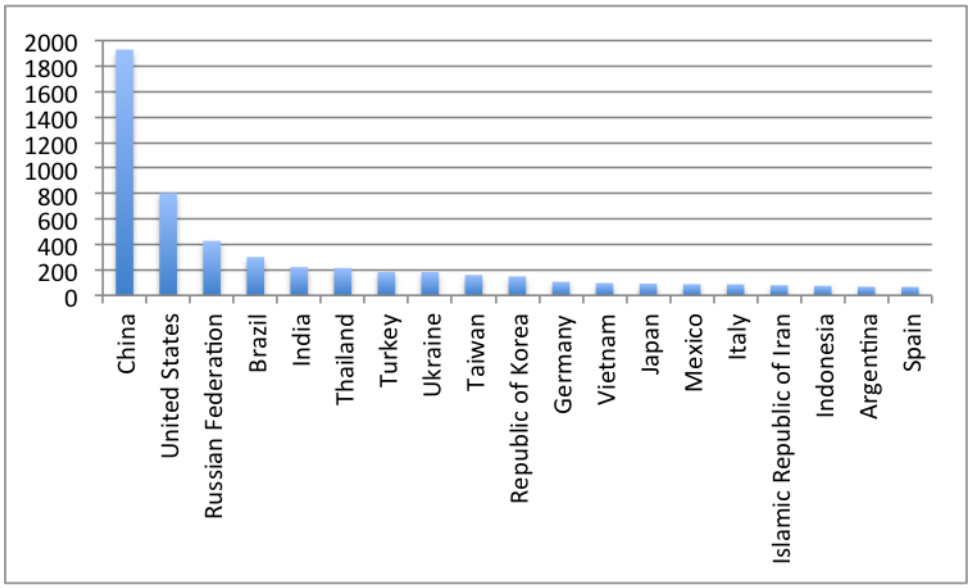

Figure 6: Overall Distribution of IP Geolocation Countries for Attacking Hosts

to warrant blacklisting. Given that this subset of hosts represents such a large portion of the overall sample, it is not surprising that the statistics on country of origin did not differ much from the overall[1]. Nevertheless, an interesting characteristic emerged from these low-threat entities, which is that the two most commonly accessed ports by this class of attacker were 3389/tcp (52.2%) and 4899/tcp (29.3%). In other words, over 80% of hosts responsible for our most benign traffic patterns may have proven not to be so benign if our attack surface included the Microsoft and Radmin screen-sharing protocols that operate on these ports. This result underscores the fact that our technique gains considerably more intelligence by expanding the attack surface vs. merely rejecting packets at the perimeter. Even within the class of singleton features, several hosts amassed very high threat scores (some as high as 8178.0) through massively repeated probing of low-interaction decoy services such as SIP and smbd.

We observed 1,255 hosts with threat indices in the range [1.0, 50.0). Among this set of sub-threshold hosts, we observed 52,858 attack events having the feature

[1]Top 5 countries were China with 1,706 hosts (32.1%), United States with 631 hosts (11.9%), Russian Federation with 353 hosts (5.1%), and Thailand with 205 hosts (3.9%).

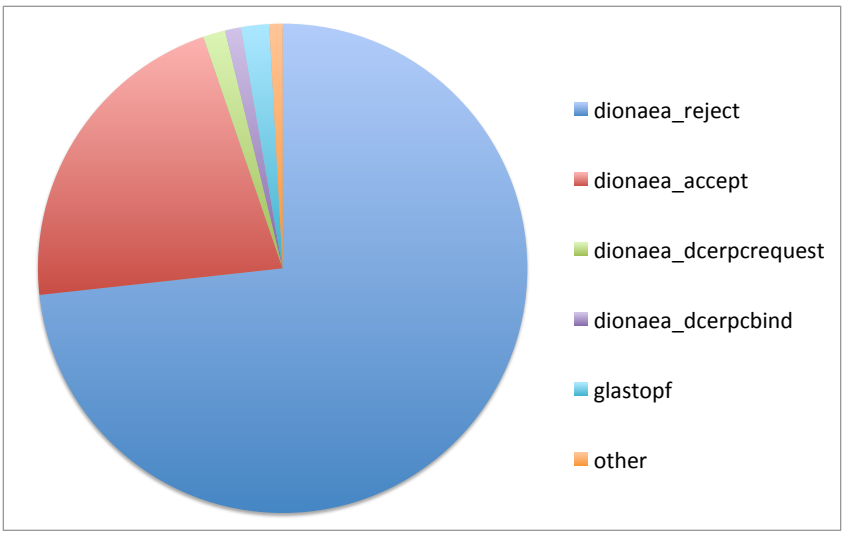

Figure 7: Attack Features for Hosts with ThreatIndex $\in$ [1.0, 50.0)

breakdown observed in Figure 7. The vast majority of the detected behaviors were connection rejects (73%) and connection acceptances (21%), still indicative of probing behavior. Hosts in this threat range representing this probing behavior totaled 1,141 (91%). The remaining features in this threat index range began to implicate hosts exhibiting low volumes of decidedly malicious behavior against the decoy hosts (e.g. *dcerpcrequest*s and *dcerpcbind*s).

Our system assigned 533 hosts a threat index of greater than or equal to 50, meaning that these hosts were classified as decidedly malicious and that production assets should be protected from their network traffic. The top five country origins[2] of this highly malicious group included China (123 hosts, 23.7%), United States (103 hosts, 19.8%), Taiwan and India (tied with 29 hosts, 5.6%), and Brazil (23 hosts, 4.4%). What is particularly notable in this breakdown is that the proportion of threats from within the United States in this category rose appreciably to rival that of China, which consistently had the next country beaten by a factor of 2-3 in other threat classes. This sheds light on the fact that many of the most determined attackers at

[2]IP geolocation data were available for 520 of the 533 hosts with threat index greater than or equal to 50.

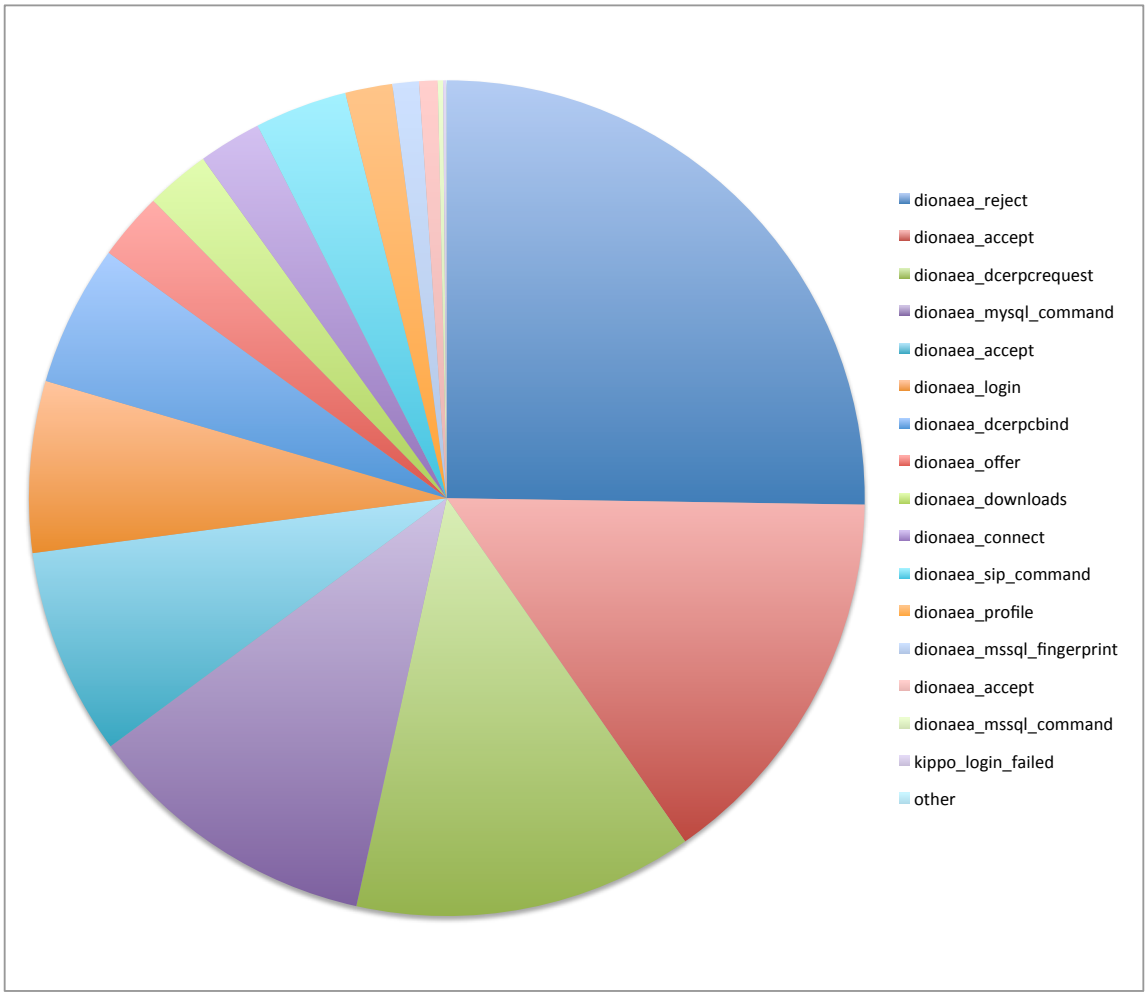

Figure 8: Attack features for hosts with ThreatIndex $\in [50.0, +\infty)$

very least have succeeded in compromising US assets that they then leverage against other targets residing in the United States.

Collectively, hosts exceeding a threat index of 50 generated a set of 810,336 features consisting of the feature classes shown in Figure 8. In contrast to the sub-50 threat index feature set, the rejections (25%) and connections (15%) comprised much smaller shares of the total. This relative reduction is caused by the fact that hosts in this subset are recorded performing malicious activities in much greater quantities. In particular, the *dcerpcrequest/dcerpcbind* events, commands executed against the *mysql* decoy service, and login attempts now each represent 5% to 13% of the total feature set. This drastically different set breakdown that occurs at or above a threat index of 50.0 led to its choice as a reasonable threshold for blacklisting.

Our technique succeeded in redirecting traffic from these 533 hosts away from production hosts once they surpassed the threat index threshold of 50.0. The attack campaigns of these 533 hosts lasted, on average, 2 days, 19 hours. On these campaigns, our technique's overall time until detection averaged approximately 12 hours, and on average, after the offending host carried out 31.43% of its attack events against our decoy sensors. Given that after detection, the host that exceeds threat index of 50.0 is diverted from production hosts, *these detection characteristics resulted in redirection of 1241.39 MiB of 1273.17 MiB bound from these malicious hosts, representing an overall reduction of attack traffic by 97.5%.*

In the approximately 1 month time of performance, our framework ingested 1,023 md5-unique malicious executable files from attackers. These files were uploaded to the decoys via attackers' exploitation of their vulnerable services (mainly by offering a file for download by the vulnerable windows RPC service). "VirusTotal is a free service that analyzes suspicious files and URLs and facilitates the quick detection of viruses, worms, trojans and all kinds of malware" [87]. It does so by scanning each sample with a wide array of approximately 50 virus detection engines. The majority of

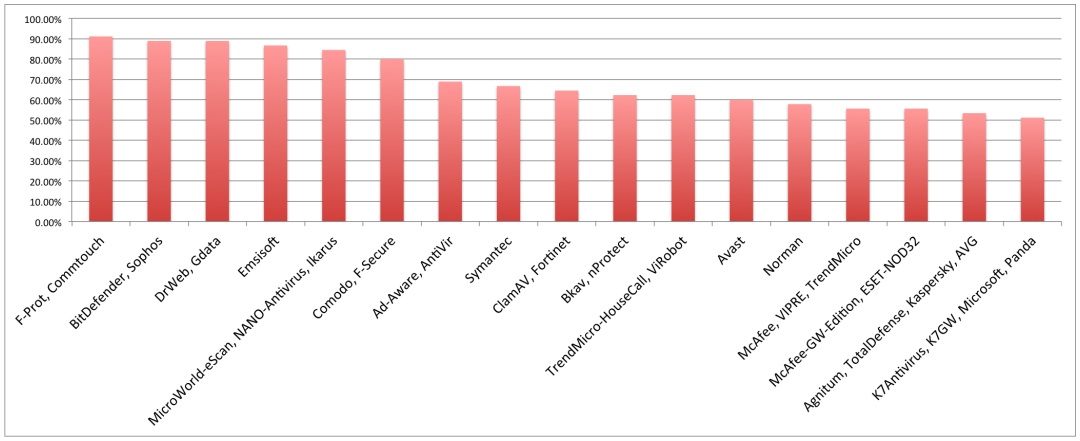

Figure 9: Detection rate of ingested malware samples by scan engine

Detection String	#j of Occurrences
Downloader	11
Trojan.Spy.XXP	11
Win32.Sality.3	11
Trojan.Win32.Generic!BT	12
Email-Worm.Win32.Atak	12
Troj/DLoad-IK	13
Worm.Generic.281334	15
Trojan.Generic.5188720	17
Gen:Variant.Graftor.54235	15
UnclassifiedMalware	22
W32/HLL-SysDlrSharer!Eldorado	30

Table 3: Most Common Virus Scanner Result Strings of Files Ingested by Framework

the samples ingested by our framework (978/1023 or 95.6%) had not been previously scanned by VirusTotal. Among the remaining 45 samples (4.4%), the percentage of virus scanners that found each sample to be malicious ranged from 0% to 95.7% and the cumulative detection percentage across all samples was 59.7%. On a per-engine basis, the detection rates ranged from 2% to 91%. The engines that had a greater than 50% detection rate are shown in Figure 9. The majority of detected files were determined to be trojan horses and malware downloaders/droppers. The most prevalent virus scanner result strings[3] from files in our sample set are listed in Table 3.

4.2 Remote Sensor Appliance

The set of throughput results without the HTB queue discipline, presented in Figure 10, show a large discrepancy between the VPN tunneled link and the native link (communication via the Internet). Specifically, the performance of the VPN tunnel degrades severely when stream bitrate exceeds the capabilities of the CPU, and as the stream bandwidth increases, the throughput of the tunneled connection tends to zero. At the highest tested stream bandwidth of 65 Mbps, the tunneled connection is able to transfer only 3.63 Mbps, down from its peak of 44.6 Mbps (-91.8%). By contrast, at the same 65 Mbps stream bandwidth, the native link throughput degrades to 46.0 Mbps from its saturation peak of 47.1 Mbps (-2.3%). An attacker who gains control of a honeypot can arbitrarily generate a high bandwidth stream, observe the ten-fold reduction in throughput, and infer that the network route involves a VPN tunnel. From there, the attacker can continue a denial of service upon the VPN, or choose to discontinue interactions with what he suspects to be a honeypot.

A similarly dramatic degradation is observed in the PDV characteristics of the tunneled connection (Figure 11) without the use of HTB queueing. Whereas the

[3]The total number of result strings exceeds the total number of detected files because multiple scanners that detect the same file may attribute different strings to it.

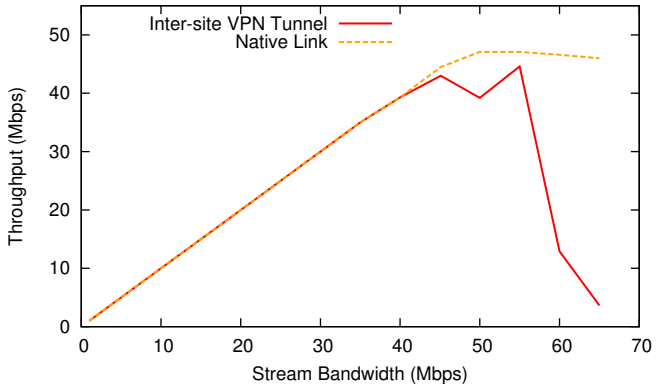

Figure 10: Detectable Throughput Chraracteristics of VPN Tunnel Under *pfifo_fast* Queueing Discipline

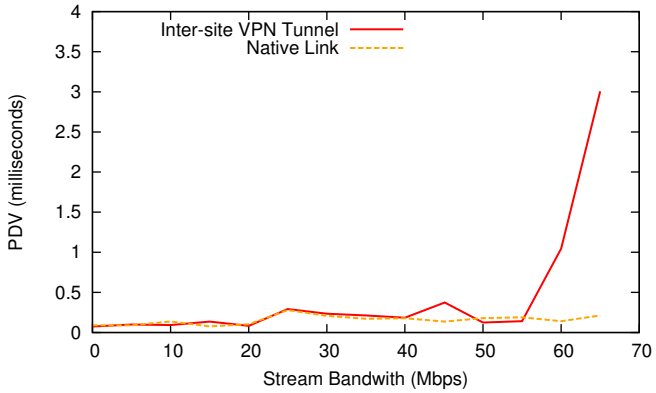

Figure 11: Detectable Packet Delay Characteristics of VPN Tunnel Under *pfifo_fast* Queueing Discipline

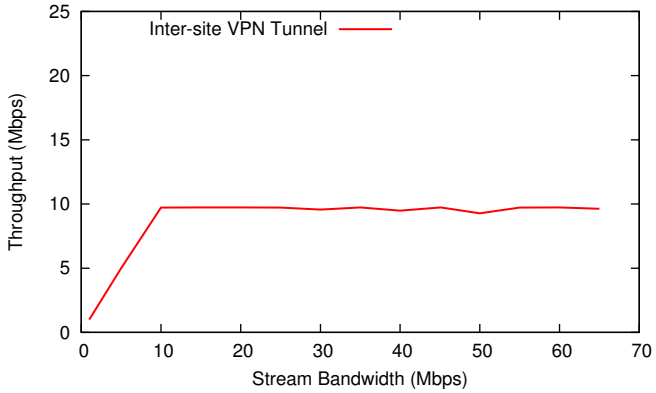

Figure 12: Improved Throughput Characteristics of VPN Tunnel Under HTB Queueing Discipline

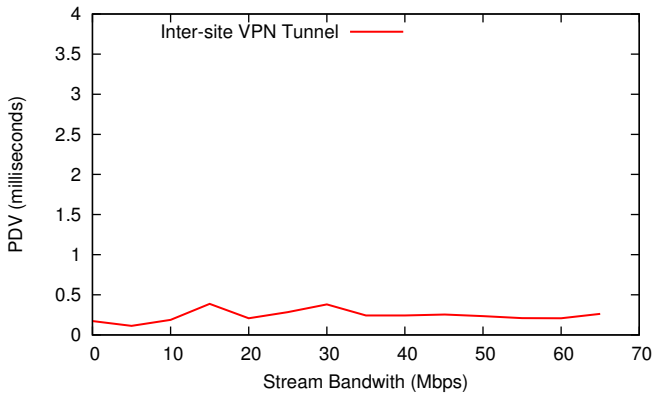

Figure 13: Improved Packet Delay Characteristics of VPN Tunnel Under HTB Queueing Discipline

	Native Link	Inter-site VPN Tunnel
Minimum	1.465	1.903
Maximum	102.392	13.977
Mean	1.766	2.391
Std. Dev.	0.133	0.324

Table 4: ICMP Round Trip Times

native connection's PDV never exceeds 280 μs, the tunneled connection's PDV reaches as high as 3.0 ms, representing an increase of approximately 1070%. An attacker can very easily identify the link as a VPN by observing this significantly higher packet delay variation on high bandwidth streams.

When the HTB queueing discipline is applied, the throughput curve (Figure 12) of the tunneled link no longer exhibits the extreme degradation in throughput for high-bandwidth streams, but rather a negligible (-1.0%) degradation that is in line with that of the native link. The peak throughput of our tunneled link is limited to 10 megabits/second according to the ceiling value we selected for our HTB queueing discipline. While this value differs from the native link's theoretical maximum, the important behavioral characteristic from a fingerprintability standpoint is that the throughput of the tunneled link no longer degrades sharply from its maximum as a function of increased stream bandwidth. Therefore, the attacker is no longer able

to uncover the fact that the underlying connection is a VPN by saturating the link with traffic. Furthermore, the VPN hardware and HTB rate ceiling could be chosen in such a way as to enable a peak throughput in line with the capabilities of the native link, but for our honeypot application, we intentionally limited the amount to 10 megabits/second to reduce the resources an attacker could consume.

Similarly, Figure 13 shows the extreme spike in PDV is eliminated through use of the HTB queueing discipline. While the VPN tunnel was exhibiting up to a 3.0 ms packet delay variation prior to implementing the HTB queueing discipline (Figure 11), afterward the tunneled link's PDV remains under 380 μs. This represents a reduction of 87% relative to the peak PDV of the tunneled link without HTB.

CHAPTER V

CONCLUSION

The turn-key honeynet framework demonstrably provides significant improvements to the logistical challenges of transparency and large-scale integration by way of its cloud-based architecture. It was also shown to have performed very effectively by providing a layer of deception that facilitates its roles as a threat intelligence gathering platform and intrusion prevention solution.

5.1 Efficacy of Turn-key Honeynet Framework

The results show that our technique is successful at using deceptive infrastructure not only to lure attackers, but to monitor their behavior and use it to classify them as malicious. Our proof-of-concept prototype, which we implemented in Python, deployed in LXC with Juju, and exposed to live internet traffic for the month of December 2013, achieved a 97.5% reduction of malicious traffic bound for production hosts by detecting and redirecting attack campaigns well before they concluded. Furthermore, our sensors maintains enhanced network-level transparency when deployed to an enterprise subnet over VPN tunneling. Specifically, the large degradation in throughput and rise in peak PDV are all but eliminated, making the attacker's task of fingerprinting the honeypots based on their network characteristic far less probable. More importantly, the automation around the framework demonstrates it is able to adapt to live threats in near-real-time and bolster the security of the network on which it is deployed without incurring additional maintenance requirements on the part of the enterprise. In its versatility to being deployed either remotely over Layer 2 VPN tunnel or within the cloud, the enterprise also need not devote significant computational resources toward this defense strategy.

5.2 Future Work

The contribution presented in this research is a framework whose efficacy is successfully demonstrated via a prototype in an academic environment. Therefore, many opportunities exist for building upon it in the future. The framework's ability to integrate with physical enterprise deployments via the traffic forwarder and public cloud deployments make it highly adaptable to real production use. However, the turn-key honeynet framework will need to be improved in the following ways:

1. Assure production readiness by load testing and making improvements to visualization/reporting
2. Expansion of the decoy services it provides as lure for attackers
3. Enhancements to the way threat indices are arrived at
4. Enhancements to the feature correlation techniques used to group attacking hosts by similarity.

5.2.1 Evolution from Prototype to Production-Ready System

While the framework served as an effective proof-of-concept for using the automated deployment of honeynets as response to novel attack patterns, the scale at which services are able to be hosted is limited by the amount of hardware available, and it was not tested against the very high loads that would be expected if used by many different organizations. Before the framework is to be deployed on real enterprise networks, it will need to be deployed to a large cluster of servers and undergo rigorous load testing to ensure it will respond to the considerable load under which it will be placed in production. To ease this process, deploying to one or two real enterprise partners will initiate the transition from prototype to production readiness.

In addition to the technical aspects of production readiness, the usability aspects will also need to be considered. The existing web front-end is adequate for visualizing

data from a research perspective. However, visualizations and reporting functionality will need to be tailored to the end users of the systems, who will most likely be executives of the organizations consuming data the framework generates.

5.2.2 Expansion of Decoy Types

We relegated our prototype's low-interaction honeypot library to Kippo (SSH), Glastopf (Web) and Dionaea (numerous protocols), again to serve as proof of concept. As shown in Appendix A by the extensible schema and simplicity with which new decoy types are integrated, our framework was designed with extensibility to new sensor types as one of the primary goals. In the future, the framework's library will include an extended set of decoy services for many additional protocols, thus increasing the attack surface available for use in deceiving attackers and entrapping them. In particular, we noted a large number of connection attempts against the Microsoft RDP and Radmin screensharing protocols, which indicates that these protocols are likely to be ripe vectors for enhanced feature detection.

We also encountered several probing attempts on port 44818/tcp, indicating active reconnaissance against the industrial automation protocol known as EtherNet/IP [74], or more specifically, the adaptation of the Common Industrial Protocol (CIP) to ethernet and TCP/IP. While we are not aware of an existing low-interaction honeypot that emulates this industrial automation protocol, the protocol specification is made public [22] so implementing one is technically feasible. The presence of these seemingly targeted attacks even on our academic network indicates this also would provide a high-value set of threat intelligence vs. our current vantage point of simply rejecting the packets and wondering what the attacker may have been trying to target. In our view, developing an EtherNet/IP honeypot to shed light on this activity is a high priority on the to-do list.

5.2.3 Enhanced Calculation of Threat Indices

We calculated our composite threat index for a given host based upon the sum of individual attack features that (mostly) had threat indices based upon the class of feature involved. For example, our sensors attributed the same threat index contribution of 0.1 to an http request for "/" as an http request for "//phpmyadmin/scripts/setup.php". The latter should contribute a higher value to the overall threat index, and in future works, we will adjust the threat index contribution based on the content as well as the class of feature.

5.2.4 Enhancements to Attack Feature Correlation

In future iterations that have both a higher traffic volume and a significantly larger corpus of attack features, we anticipate much higher computational complexity involved with fully searching the attack features as we have done in this work. We plan to make enhancements to our correlation approach and rely on it to make probabilistic decisions that fit in a real-time computational envelope, while buffering the higher fidelity calculations for completion in the background. We will also work to remove data dependencies wherever possible to make the computations more parallelizable and dispatchable to a large-scale HPC environment, such as Georgia Tech's PACE cluster [40].

APPENDIX A

AGGREGATION SCHEMA/SOURCE CODE

This section provides the schema used for attack aggregation, as well as listings of source code that integrates the API to existing honeypot technologies.

A.1 Schema

Listing A.1: Feature Schema

```
1  {
2    "title" : "Feature_Schema",
3    "type" : "object",
4    "required" : [
5          "_id", "threat_index", "sensor_type", "feature_type",
6    ],
7    "properties" : {
8      "_id" : { "type" : "BinData" },
9      "threat_index" : { "type" : "float" },
10     "sensor_type" : {
11       "type" : "string",
12       "oneOf" : [
13         "dionaea", "glastopf", "kippo"
14       ],
15     },
16     "feature_type" : {
17       "type" : "string",
18       "oneOf" :[
19         // glastopf features
20         "request_raw", "request",
21
```

```
22          // dionaea features
23          "accept", "reject", "connect", "service", "listen",
24          "profile", "offer", "login", "downloads",
25          "dcerpcrequest", "dcerpcbind", "mysql_command",
26          "mssql_fingerprint", "mssql_command",
27          "sip_command::cmd", "sip_command::addr",
28          "sip_command::via"
29
30          // kippo features
31          "client_version", "login_failed", "login_succeeded",
32          "terminal_size", "lost_connection"
33        ],
34      },
35      "feature_data" : {
36        "type" : "object",
37        "oneOf" : [
38          { "$ref" : "#/definitions/requestRawFeature" },
39          { "$ref" : "#/definitions/requestFeature" },
40          { "$ref" : "#/definitions/netProtocolFeature" },
41          { "$ref" : "#/definitions/urlFeature" },
42          { "$ref" : "#/definitions/profileFeature" },
43          { "$ref" : "#/definitions/loginFeature" },
44          { "$ref" : "#/definitions/downloadFeature" },
45          { "$ref" : "#/definitions/dcerpcrequestFeature" },
46          { "$ref" : "#/definitions/dcerpcbindFeature" },
47          { "$ref" : "#/definitions/mysqlCommandFeature" },
48          { "$ref" : "#/definitions/mssqlFingerprintFeature" },
49          { "$ref" : "#/definitions/mssqlCommandFeature" },
50          { "$ref" : "#/definitions/sipCmdFeature" },
51          { "$ref" : "#/definitions/sipAddrFeature" },
52          { "$ref" : "#/definitions/sipViaFeature" },
53          { "$ref" : "#/definitions/clientVersionFeature" },
```

```
54          { "$ref" : "#/definitions/terminalSizeFeature" },
55          { "$ref" : "#/definitions/lostConnectionFeature" },
56        ],
57      },
58    },
59    "definitions" : {
60      "requestRawFeature" : {
61        "request_raw" : { "type" : "string" },
62      },
63      "requestFeature" : {
64        "pattern" : { "type" : "string" },
65        "request_url" : { "type" : "string" },
66      },
67      "netProtocolFeature" : {
68        "port" : { "type" : "integer" },
69        "transport" : { "type" : "string" },
70        "protocol" : { "type" : "string" },
71      },
72      "urlFeature" : {
73        "url" : { "type" : "string" },
74      },
75      "profileFeature" : {
76        "profile_json" : { "type" : "string" },
77      },
78      "loginFeature" : {
79        "username" : { "type" : "string" },
80        "password" : { "type" : "string" },
81      },
82      "downloadFeature" : {
83        "md5" : { "type" : "string" },
84        "url" : { "type" : "string" },
85      },
```

```
86      "dcerpcrequestFeature" : {
87          "name" : { "type" : "string" },
88          "opname" : { "type" : "string" },
89          "opnum" : { "type" : "integer" },
90          "opvuln" : { "type" : "string" },
91          "uuid" : { "type" : "uuid" },
92      },
93      "dcerpcbindFeature" : {
94          "name" : { "type" : "string" },
95          "transfersyntax" : { "type" : "uuid" },
96          "uuid" : { "type" : "uuid" },
97      },
98      "mysqlCommandFeature" : {
99        "args" : [
100         { "mysql_command_arg_data" : { "type" : "string" }, },
101       ],
102       "cmd" : { "type" : "string" },
103       "op_name" : { "type" : "string" },
104     },
105     "mssqlFingerprintFeature" : {
106       "appname" : { "type" : "string" },
107       "cltintname" : { "type" : "string" },
108       "hostname" : { "type" : "string" },
109     },
110     "mssqlCommandFeature" : {
111       "cmd" : { "type" : "string" },
112       "status" : { "type" : "string" },
113     },
114     "sipCmdFeature" : {
115       "allow" : { "type" : "integer" },
116       "call_id" : { "type" : "string" },
117       "method" : { "type" : "string" },
```

```
118         "user_agent" : { "type" : "string" },
119       },
120       "sipAddrFeature" : {
121         "display_name" : { "type" : "string" },
122         "type" : { "type" : "string" },
123         "uri_host" : { "type" : "string" },
124         "uri_port" : { "type" : "integer" },
125         "uri_scheme" : { "type" : "string" },
126         "uri_user" : { "type" : "string" },
127       },
128       "sipViaFeature" : {
129         "address" : { "type" : "string" },
130         "port" : { "type" : "integer" },
131         "protocol" : { "type" : "string" },
132       },
133       "clientVersionFeature" : {
134         "version" : { "type" : "string" },
135       },
136       "terminalSizeFeature" : {
137         "height" : { "type" : "integer" },
138         "width" : { "type" : "integer" },
139       },
140       "connectionLostFeature" : {
141         "ttylog" : { "type" : "BinData" },
142       },
143     },
144 }
```

A.2 Kippo Integration Source

Integration with Kippo required writing a binding to allow it to use MongoDB as its backing event storage (Listing A.2). In addition, logic was written (Listing A.3) to

extract the events into features that conform to the feature schema in Listing A.1.

Listing A.2: MongoDB Integration

```
1  from kippo.core import dblog
2  from twisted.enterprise import adbapi
3  from twisted.internet import defer
4  from twisted.python import log
5
6  import bson
7  import pymongo
8
9  import datetime
10 import uuid
11
12 class DBLogger(dblog.DBLogger):
13     def start(self, cfg):
14         host = cfg.get('database_mongodb', 'host')
15         port = cfg.getint('database_mongodb', 'port')
16         client = pymongo.connection.Connection(host, port)
17         dbname = cfg.get(
18             'database_mongodb', 'dbname', 'kippo')
19         self.db = client[dbname]
20
21     def write(self, session_id, event):
22         event['timestamp'] = datetime.datetime.utcnow()
23         _id = bson.ObjectId(session_id)
24         session = self.db.session.find_one({'_id' : _id })
25
26         if session is None:
27             log('Found no session with _id={}'.format(
28                 session_id))
29             return
30
```

```
31          session['events'].append(event)
32          self.db.session.save(session)
33
34      def createSession(
35          self, peerIP, peerPort, hostIP, hostPort):
36          sensorname = self.getSensor() or hostIP
37          sid = self.db.session.insert(
38              {
39                  'sensor_name' : sensorname,
40                  'peer_ip' : peerIP,
41                  'peer_port' : peerPort,
42                  'events' : []
43              }
44          )
45          return str(sid)
46
47      def handleConnectionLost(self, session, args):
48          self.write(session, {
49              'type' : 'lost_connection',
50              'ttylog' : bson.binary.Binary(
51                  self.ttylog(session))
52          })
53
54      def handleLoginFailed(self, session, args):
55          self.write(session, {
56              'type' : 'login_failed',
57              'username' : args['username'],
58              'password' : args['password']
59          })
60
61      def handleLoginSucceeded(self, session, args):
62          self.write(session, {
```

```
63              'type' : 'login_succeeded',
64              'username' : args['username'],
65              'password' : args['password']
66          })
67
68      def handleCommand(self, session, args):
69          self.write(session, {
70              'type' : 'command',
71              'input' : args['input'],
72              'success' : True,
73          })
74
75      def handleUnknownCommand(self, session, args):
76          self.write(session, {
77              'type' : 'command',
78              'input' : args['input'],
79              'success' : False,
80          })
81
82      def handleInput(self, session, args):
83          self.write(session, {
84              'type' : 'input',
85              'input' : args['input'],
86              'realm' : args['realm'],
87          })
88
89      def handleTerminalSize(self, session, args):
90          self.write(session, {
91              'type' : 'terminal_size',
92              'width' : args['width'],
93              'height' : args['height'],
94          })
```

```
95
96      def handleClientVersion(self, session, args):
97          self.write(session, {
98              'type' : 'client_version',
99              'version' : args['version'],
100         })
101
102     def handleFileDownload(self, session, args):
103         self.write(session, {
104             'type' : 'file_download',
105             'url' : args['url'],
106             'outfile' : args['outfile'],
107         })
108
109 # vim: set sw=4 et:
```

Listing A.3: Kippo Feature Extraction

```
1  import hashlib
2
3  import bson
4  import bson.json_util
5  import pymongo
6
7  conn = pymongo.connection.Connection('10.0.3.6')
8
9  kippo_db = conn['kippo']
10 correlation_db = conn['correlation']
11
12 def record_feature(remote_host, ts, feature):
13     feature_id = bson.Binary(hashlib.sha256(
14         bson.json_util.dumps(feature)).digest())
15     feature['_id'] = feature_id
```

```
16
17      correlation_db.features.update(
18          {'_id' : feature_id},
19          feature,
20          upsert=True,
21      )
22
23      correlation_db.host_features.update(
24          {'_id' : remote_host},
25          {'$push' : {'features' : {
26              'id' : feature_id, 'ts' : ts}}
27          },
28          upsert=True,
29      )
30
31  transform_functions = {
32      'client_version' : lambda event : {
33          'version' : event['version'],
34      },
35      'login_failed' : lambda event : {
36          'username' : event['username'],
37          'password' : event['password'],
38      },
39  }
40
41  def convert_kippo_event_to_feature(remote_host, event):
42      timestamp = event.pop('timestamp')
43      feature_type = event.pop('type')
44      event_feature = {
45          'sensor_type' : 'kippo',
46          'feature_type' : feature_type,
47       'feature_data' : event,
```

```
48      }
49      record_feature(remote_host, timestamp, event_feature)
50
51  def feature_extraction_core(query=None):
52      matches = kippo_db.session.find(query)
53
54      for match in matches:
55          _id = match.pop('_id')
56          remote_host = match.pop('peer_ip')
57
58          for event in match['events']:
59              convert_kippo_event_to_feature(remote_host, event)
60
61  if __name__ == '__main__':
62      # Perform batch feature extraction if run from commandline
63      feature_extraction_core()
```

A.3 Glastopf Integration Source

Glastopf already includes a binding that allows it to store data in MongoDB, so no additional code was needed to implement this functionality. Extraction of Glastopf events into features that conform to the schema is shown in Listing A.4.

Listing A.4: Glastopf Feature Extraction

```
1  import datetime
2  import hashlib
3  import json
4
5  import bson
6  import pymongo
7
8  conn = pymongo.connection.Connection('10.0.3.6')
9
```

```
10 glastopf_db = conn['glastopf']
11 correlation_db = conn['correlation']
12
13 def record_feature(remote_host, ts, feature):
14     feature_id = bson.Binary(
15         hashlib.sha256(json.dumps(feature)).digest())
16     feature['_id'] = feature_id
17
18     correlation_db.features.update(
19         {'_id' : feature_id},
20         feature,
21         upsert=True,
22     )
23
24     correlation_db.host_features.update(
25         {'_id' : remote_host},
26         {'$push' : {'features' : {
27             'id' : feature_id, 'ts' : ts}}
28         },
29         upsert=True,
30     )
31
32 def feature_extraction_core(query=None):
33     matches = glastopf_db.events.find(query)
34
35     for match in matches:
36         _id = match.pop('_id')
37         timestamp = datetime.datetime.strptime(
38             match.pop('time'), '%Y-%m-%d %H:%M:%S')
39         remote_host = match.pop('source')[0]
40
41         record_feature(remote_host, timestamp, {
```

```
42              'sensor_type' : 'glastopf',
43              'feature_type' : 'request',
44              'feature_data' : {
45                  'pattern' : match['pattern'],
46                  'request_url' : match['request_url'],
47              },
48          })
49
50          record_feature(remote_host, timestamp, {
51              'sensor_type' : 'glastopf',
52              'feature_type' : 'request_raw',
53              'feature_data' : {
54                  'request_raw' : match['request_raw'],
55              },
56          })
57
58  feature_extraction_core()
```

A.4 Dionaea Integration Source

As Dionaea uses a complex mix of files and persistence to a sqlite database to store its events, the migration of this functionality to MongoDB was not performed at this time. Rather, the extraction of Dionaea features from the sqlite database is done in batch fashion on a periodic schedule by the code shown in Listing A.5.

Listing A.5: Dionaea Feature Extraction

```
1  import hashlib
2  import json
3
4  import bson
5  import pymongo
6
```

```
7  conn = pymongo.connection.Connection('10.0.3.6')
8
9  dionaea_db = conn['dionaea']
10 correlation_db = conn['correlation']
11
12 def record_feature(remote_host, ts, feature):
13     feature_id = bson.Binary(hashlib.sha256(
14         json.dumps(feature)).digest())
15     feature['_id'] = feature_id
16
17     correlation_db.features.update(
18         {'_id' : feature_id},
19         feature,
20         upsert=True,
21     )
22
23     correlation_db.host_features.update(
24         {'_id' : remote_host},
25         {'$push' : {'features' : {
26             'id' : feature_id, 'ts' : ts}}
27         },
28         upsert=True,
29     )
30
31 def transform_connection(connection):
32     feature = {
33         'sensor_type' : 'dionaea',
34         'port' : connection['local_port'],
35     }
36
37     if(connection['connection_type'] == 'reject'):
38         feature['feature_type'] = 'reject'
```

```
39          feature['transport'] = connection[
40              'connection_transport'].strip()
41      elif(connection['connection_type'] == 'accept'):
42          feature['feature_type'] = 'accept'
43          feature['protocol'] = connection[
44              'connection_protocol']
45      elif(connection['connection_type'] == 'listen'):
46          feature['feature_type'] = 'listen'
47          feature['protocol'] = connection[
48              'connection_protocol']
49      elif(connection['connection_type'] == 'connect'):
50          feature['feature_type'] = 'connect'
51          feature['protocol'] = connection[
52              'connection_protocol']
53
54      return feature
55
56  transform_functions = {
57      'logins' : lambda login : {
58          'sensor_type' : 'dionaea',
59          'feature_type' : 'login',
60          'feature_data' : {
61              'username' : login['login_username'],
62              'password' : login['login_password'],
63          },
64      },
65      'mssql_fingerprints' : lambda mssql_fingerprint : {
66          'sensor_type' : 'dionaea',
67          'feature_type' : 'mssql_fingerprint',
68          'feature_data' : {
69              'cltintname' : mssql_fingerprint[
70                  'mssql_fingerprint_cltintname'],
```

```
71              'hostname' : mssql_fingerprint[
72                  'mssql_fingerprint_hostname'],
73              'appname' : mssql_fingerprint[
74                  'mssql_fingerprint_appname'],
75          },
76      },
77      'mssql_commands' : lambda mssql_command : {
78          'sensor_type' : 'dionaea',
79          'feature_type' : 'mssql_command',
80          'feature_data' : {
81              'cmd' : mssql_command['mssql_command_cmd'],
82              'status' : mssql_command['mssql_command_status'],
83          },
84      },
85      'mysql_commands' : lambda mysql_command : {
86          'sensor_type' : 'dionaea',
87          'feature_type' : 'mysql_command',
88          'feature_data' : {
89              'cmd': mysql_command['cmd']['mysql_command_cmd'],
90              'op_name': mysql_command['cmd'][
91                  'mysql_command_op_name'],
92              'args': mysql_command['args'],
93          },
94      },
95      'dcerpcrequests' : lambda dcerpcrequest : {
96          'sensor_type' : 'dionaea',
97          'feature_type' : 'dcerpcrequest',
98          'feature_data' : {
99              'uuid' : dcerpcrequest['dcerpcrequest_uuid'],
100             'opnum' : dcerpcrequest['dcerpcrequest_opnum'],
101             'name' : dcerpcrequest['dcerpcservice_name'],
102             'opname' : dcerpcrequest['dcerpcserviceop_name'],
```

```
103             'opvuln' : dcerpcrequest['dcerpcserviceop_vuln'],
104         },
105     },
106     'dcerpcbinds' : lambda dcerpcbind : {
107         'sensor_type' : 'dionaea',
108         'feature_type' : 'dcerpcbind',
109         'feature_data' : {
110             'name' : dcerpcbind['dcerpcservice_name'],
111             'uuid' : dcerpcbind['dcerpcbind_uuid'],
112             'transfersyntax' : dcerpcbind[
113                 'dcerpcbind_transfersyntax'],
114         },
115     },
116     'downloads' : lambda download : {
117         'sensor_type' : 'dionaea',
118         'feature_type' : 'downloads',
119         'feature_data' : {
120             'md5' : download['download']['download_md5_hash'],
121             'url' : download['download']['download_url'],
122         },
123     },
124     'profiles' : lambda profile : {
125         'sensor_type' : 'dionaea',
126         'feature_type' : 'profile',
127         'feature_data' : {
128             'profile_json' : json.dumps(json.loads(
129                 profile['emu_profile_json'])),
130         },
131     },
132     'offers' : lambda offer : {
133         'sensor_type' : 'dionaea',
134         'feature_type' : 'offer',
```

```
135         'feature_data' : {
136             'url': offer['offer_url'],
137         },
138     },
139     'services' : lambda service : {
140         'sensor_type' : 'dionaea',
141         'feature_type' : 'service',
142         'feature_data' : {
143             'url': service['emu_service_url'],
144         },
145     },
146     'sip_commands::cmd' : lambda cmd : {
147         'sensor_type' : 'dionaea',
148         'feature_type' : 'sip_command::cmd',
149         'feature_data' : {
150             'method' : cmd['sip_command_method'],
151             'user_agent' : cmd['sip_command_user_agent'],
152             'call_id' : cmd['sip_command_call_id'],
153             'allow' : cmd['sip_command_allow'],
154         },
155     },
156     'sip_commands::addrs' : lambda addr : {
157         'sensor_type' : 'dionaea',
158         'feature_type' : 'sip_command::addr',
159         'feature_data' : {
160             'type' : addr['sip_addr_type'],
161             'uri_scheme' : addr['sip_addr_uri_scheme'],
162             'uri_port' : addr['sip_addr_uri_port'],
163             'uri_user' : addr['sip_addr_uri_user'],
164             'display_name' : addr['sip_addr_display_name'],
165             'uri_host' : addr['sip_addr_uri_host'],
166         },
```

```
167         },
168         'sip_commands::vias' : lambda via : {
169             'sensor_type' : 'dionaea',
170             'feature_type' : 'sip_command::via',
171             'feature_data' : {
172                 'protocol' : via['sip_via_protocol'],
173                 'port' : via['sip_via_port'],
174                 'address' : via['sip_via_address'],
175             },
176         },
177         'children' : lambda child : transform_connection(
178             child['connection']),
179 }
180
181 def record_sip_commands(remote_host, timestamp, sip_commands):
182     for command in sip_commands:
183         record_feature(
184             remote_host,
185             timestamp,
186             transform_functions['sip_commands::cmd'](
187                 command.pop('cmd'))
188         )
189
190         for field, value in command.iteritems():
191             for feature in value:
192                 feature_type = 'sip_commands::{}'.format(
193                     field)
194                 record_feature(
195                     remote_host,
196                     timestamp,
197                     transform_functions[feature_type](feature)
198                 )
```

```
199
200 def feature_extraction_core(query):
201     matches = dionaea_db.connections.find(query)
202
203     for match in matches:
204         _id = match.pop('_id')
205         timestamp = match.pop('timestamp')
206         connection = match['connection']
207         remote_host = connection['remote_host']
208
209         record_feature(
210             remote_host,
211             timestamp,
212             transform_connection(match.pop('connection')),
213         )
214
215         record_sip_commands(
216             remote_host, timestamp,
217             match.pop('sip_commands', []))
218
219         for field, value in match.iteritems():
220             for feature in value:
221                 record_feature(
222                     remote_host,
223                     timestamp,
224                     transform_functions[field](feature)
225                 )
226
227         dionaea_db.connections.remove({'_id' : _id})
228
229 def extract_rejects():
230     query = {
```

```
231         'connection.connection_type' : 'reject',
232     }
233     feature_extraction_core(query)
234
235 def extract_httpd_accept():
236     query = {
237         'connection.connection_type' : 'accept',
238         'connection.connection_protocol' : 'httpd',
239     }
240
241     feature_extraction_core(query)
242
243 def extract_mssqld_accept():
244     query = {
245         'connection.connection_type' : 'accept',
246         'connection.connection_protocol' : 'mssqld',
247     }
248
249     feature_extraction_core(query)
250
251 def extract_mysqld_accept():
252     query = {
253         'connection.connection_type' : 'accept',
254         'connection.connection_protocol' : 'mysqld',
255     }
256
257     feature_extraction_core(query)
258
259 def extract_smbd_accept():
260     query = {
261         'connection.connection_type' : 'accept',
262         'connection.connection_protocol' : 'smbd',
```

```
263     }
264
265     feature_extraction_core(query)
266
267 def extract_ftpd_accept():
268     query = {
269         'connection.connection_type' : 'accept',
270         'connection.connection_protocol' : 'ftpd',
271     }
272
273     feature_extraction_core(query)
274
275 def extract_epmapper_accept():
276     query = {
277         'connection.connection_type' : 'accept',
278         'connection.connection_protocol' : 'epmapper',
279     }
280
281     feature_extraction_core(query)
282
283 def extract_SipSession_accept():
284     query = {
285         'connection.connection_type' : 'connect',
286         'connection.connection_protocol' : 'SipSession',
287     }
288
289     feature_extraction_core(query)
290
291
292 extract_rejects()
293 extract_httpd_accept()
294 extract_mssqld_accept()
```

```
295 extract_smbd_accept()
296 extract_mysqld_accept()
297 extract_ftpd_accept()
298 extract_epmapper_accept()
299 extract_SipSession_accept()
```

APPENDIX B

SYSTEM APIS

This appendix provides a high level overview of the enabling methods for the HoneyCluster, datasource, and automated firewall components of the framework.

B.1 HoneyCluster

The interface to the HoneyCluster, which manages the lifecycle of honeypot assets, is provided by the following methods:

- provision
  - arguments: *service_name, ip_address, net_interface*
  - returns: *host_report_data*
  - description: Creates a new honeypot instance according to a template described by *service_name*. Once created, automatically invokes the necessary firewall methods to expose the honeypot instance as *ip_address*, via *net_interface*.
- deprovision
  - arguments: *decoy_name*
  - returns: *command_success*
  - description: Destroys the honeypot instance associated with *decoy_name* and automatically invokes the necessary firewall methods to remove it from the network.

B.2 DataSource

The interface to the threat intelligence data is enabled by the following methods:

- get_geoip
  - arguments: *host_ip*
  - returns: *geoip_data*
  - description: Retrieves and returns IP Geolocation Data for *host_ip* from freegeoip.net [32].
- get_whois
  - arguments: *host_ip*
  - returns: *geoip_data*
  - description: Retrieves and returns IP WHOIS Data for *host_ip* from whois.arin.net [4].
- get_host_report
  - arguments: *host_ip*
  - returns: *host_report_data*
  - description: Returns document containing all attack features observed as originating from *host_ip*.
- get_known_hosts
  - arguments: None
  - returns: *known_hosts_list*
  - description: Returns list containing IP addresses of all hosts that have interacted with the framework.

- get_decoy_list
  - arguments: None
  - returns: List containing all active decoys within the system.
- get_decoy_report
  - arguments: *decoy_name*
  - returns: decoy_report.
  - description: If decoy named *decoy_name* exists, returns document with vital information including the service(s) hosted, uptime, and all observed attack features.
- get_threat_trends
  - arguments: *period_of_interest*
  - returns: *threat_report*
  - description: Returns document with aggregated attack statistics for the period specified in *period_of_interest*.

B.3 Automated Firewall

The interface to control the firewall in automated fashion is enabled by methods that fall into the following categories:

- System State Listings
  - list_rules
    - arguments: None
    - returns: List containing all exigent iptables rules
  - list_ip_aliases

    - arguments: None
    - returns: List containing all exigent IP aliases on all interfaces

- Direct Rule Manipulation
  - insert_rule
    - arguments: *rule_to_insert*
    - returns: *command_success*
    - description: Inserts rule described by *rule_to_insert* if it does not exist. Does nothing otherwise.
  - remove_rule
    - arguments: *rule_to_remove*
    - returns: *command_success*
    - description: Removes rule matching *rule_to_remove* if one exists. Does nothing otherwise.

- High-Level Rule Manipulation
  - expose_port
    - arguments: *net_interface, external_ip_address, internal_ip_address, external_port, internal_port, transport_protocol*
    - returns: *command_success*
    - description: Exposes service at *internal_ip_address*:*internal_port* as *external_ip_address*:*internal_port* via *net_interface*.
  - setup_one_to_one_nat
    - arguments: *net_interface, external_ip_address, internal_ip_address*
    - returns: *command_success*

        * description: Exposes internal host at *internal_ip_address* on *external_ip_address* via *net_interface*.

    - blacklist
        * arguments: *ip_address*
        * returns: *command_success*
        * description: Adds rule to place packets bound from remote host at *ip_address* onto HOSTILE chain.

    - deblacklist
        * arguments: *ip_address*
        * returns: *command_success*
        * description: Removes rule (if exists) that places packets bound from remote host at *ip_address* onto HOSTILE chain.

- IP Alias Manipulation

    - establish_ip_alias
        * arguments: *net_interface, ip_address, netmask*
        * returns: *command_success*
        * description: Adds a new ip alias for *ip_address*/*netmask* on *net_interface* if one does not exist. Does nothing otherwise.

    - remove_ip_alias
        * arguments: *ip_address*
        * returns: *command_success*
        * description: Removes the ip alias for *ip_address* if it exists. Does nothing otherwise.

REFERENCES

[1] Akerman, N., Anderson, R., Arora, A., de Barros, A. P., Blum, R. O., Carter, L. R., Edwards, L., Farnsely, G. F., Gercke, M., Kannan, K., Krishnan, S., Lee, H., Longstaff, T., Rees, J., Shimeall, T. J., Spafford, E. H., Takefuji, Y., Uchida, K., Versace, M., and Yuqing, S., "Unsecured economies: Protecting vital information," tech. rep., McAfee, January 2009.

[2] Amazon, "Aws security center." `http://aws.amazon.com/security/`.

[3] Amies, A., Sluiman, H., Tong, Q. G., and Liu, G. N., *Developing and Hosting Applications on the Cloud (IBM Press)*. IBM Press, 2012.

[4] "Arin whois," 2014. `http://whois.arin.net`.

[5] AyeThu, A., "Integrated intrusion detection and prevention system with honeypot on cloud computing environment," *International Journal of Computer Applications*, vol. 67, no. 4, pp. 9–13, 2013.

[6] Bächer, P., Holz, T., Kötter, M., and Wicherski, G., "Know your enemy: Tracking botnets.," *The Honeynet Project: Know Your Enemy Series*, August 2008.

[7] Barford, P. and Yagneswaran, V., "An inside look at botnets," in *Advances in Information Security*, 2006.

[8] Biedermann, S., Mink, M., and Katzenbeisser, S., "Fast dynamic extracted honeypots in cloud computing," in *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pp. 13–18, ACM, 2012.

[9] BlueLock, "Cloud security features and functionality." `http://www.bluelock.com/virtual-datacenters/cloud-security/security-features/`.

[10] Bodeau, D. J., Graubart, R., and Fabius-Greene, J., "Improving cybersecurity and mission assurance via cyber preparedness (cyber prep) levels," *Social Computing / IEEE International Conference on Privacy, Security, Risk and Trust*, pp. 1147–1152, 2010.

[11] Boehme, A., Flores, B., Schweitzer, J., and Islam, J., "Software defined perimeter," tech. rep., Cloud Security Alliance, December 2013.

[12] Boniface, M., Nasser, B., Papay, J., Phillips, S., Servin, A., Yang, X., Zlatev, Z., Gogouvitis, S., Katsaros, G., Konstanteli, K., Kousiouris, G., Menychtas, A., and Kyriazis, D., "Platform-as-a-service architecture for real-time quality of service management in clouds," in *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pp. 155–160, May 2010.

[13] Borland, T., "Attacking kippo," 2013. http://www.alertlogic.com/attacking-kippo/.

[14] Bradley, T., "Epsilon data breach: Expect a surge in spear phishing attacks," *PCWorld*, April 2011.

[15] Brown, S., Lam, R., Prasad, S., Ramasubramanian, S., and Slauson, J., "Honeypots in the cloud," 2012.

[16] "Carnivore news," 2014. http://libemu.carnivore.it.

[17] Carnivore.it, "Dionaea — catches bugs," 2009. http://dionaea.carnivore.it.

[18] CenturyLink, "Cloud security: Cios have more pressing things to worry about." http://www.centurylink.com/business/asset/white-paper/cloud-security-cios-have-more-pressing-things-to-worry-about-cm101361.pdf.

[19] Cha, A. and Nakashima, E., "Google china cyberattack part of vast espionage campaign, experts say," *The Washington Post*, January 2010.

[20] Chen, T., "Stuxnet, the real start of cyber warfare?," *IEEE Network*, vol. 25, no. 2, pp. 2–3, 2011.

[21] Chen, Y., Paxson, V., and Katz, R. H., "What's new about cloud computing security," *University of California, Berkeley Report No. UCB/EECS-2010-5 January*, vol. 20, no. 2010, pp. 2010–5, 2010.

[22] "The cip networks library; volume 1; common industrial protocol (cip)," 2013. ftp://ftp.heapg.com/ARCHIVE/AB-OCS/EthernetIPCD/CIP/Vol1_3.3.pdf.

[23] Cloudflare, "Cloudflare security." https://www.cloudflare.com/features-security.

[24] Damballa, "Advanced persistent threats," tech. rep., Damballa, Inc., 2010.

[25] Damballa, "The command structure of the aurora botnet: History, patterns, and findings," tech. rep., Damballa, Inc., 2010.

[26] Dodge, R., Brown, R., and Ragsdale, D., "Honeynet solutions," *CISR-WECS6*, 2004.

[27] Dornseif, M. and May, S., "Modelling the costs and benefits of honeynets," *arXiv preprint cs/0406057*, 2004.

[28] "Ember charts," 2013. `http://addepar.github.io/#/ember-charts/overview`.

[29] "Ember.js - a framework for creating ambitious web applications," 2013. `http://emberjs.com`.

[30] Evangelinos, C. and Hill, C., "Cloud computing for parallel scientific hpc applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's ec2.," *ratio*, vol. 2, no. 2.40, pp. 2–34, 2008.

[31] Falliere, N., "Stuxnet introduces the first known rootkit for industrial control systems," tech. rep., Symantec, Inc., August, 2010.

[32] "freegeoip.net," 2014. `http://freegeoip.net`.

[33] Fucs, A., de Barros, A. P., and Pereira, V., "New botnets trends and threats," in *Black Hat Europe*, (http://www.blackhat.com/presentations/bh-europe-07/Fucs-Paes-de-Barros-Pereira/Whitepaper/bh-eu-07-barros-WP.pdf), 2007.

[34] Garfinkel, T. and Rosenblum, M., "A virtual machine introspection based architecture for intrusion detection," in *NDSS*, 2003.

[35] Ghosh, A. K., Schwartzbard, A., and Schatz, M., "Learning program behavior profiles for intrusion detection," in *Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring*, vol. 1, p. 6, 1999.

[36] Giani, A., Berk, V. H., and Cybenko, G. V., "Data exfiltration and covert channels," in *Proceedings of SPIE Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense V*, 2006.

[37] "Glastopf," 2010. `http://glastopf.org`.

[38] Grizzard, J., Sharma, V., Nunnery, C., Kang, B. B., and Dagon, D., "Peer-to-peer botnets: Overview and case study," in *First Workshop on Hot Topics in Understanding Botnets*, 2007.

[39] Grow, B. and Epstein, K., "The new e-spionage threat," *Business Week*, 2008.

[40] "Pace: A partnership for an advanced computing environment," 2014. `http://pace.gatech.edu`.

[41] Gu, G., Perdisci, R., Zhang, J., and Lee, W., "Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection," in *Proceedings of the 17th Conference on Security Symposium*, SS'08, (Berkeley, CA, USA), pp. 139–154, USENIX Association, 2008.

[42] Hoffman, S., "Rsa: Aurora hackers targeted google source code," *CRN*, March 2010.

[43] Honda, O., Ohsaki, H., Imase, M., Ishizuka, M., and Murayama, J., "Understanding tcp over tcp: effects of tcp tunneling on end-to-end throughput and latency," in *Optics East 2005*, pp. 60110H–60110H, International Society for Optics and Photonics, 2005.

[44] HoneyMole, "Honeymole project page," 2009. `http://www.honeynet.org.pt/index.php/HoneyMole`.

[45] HoneynetProject, "About the honeynet project," 2008. `http://www.honeynet.org/about`.

[46] "Iperf sourceforge project page," 2011. `http://iperf.sourceforge.net`.

[47] Jiang, X., Xu, D., and Wang, Y.-M., "Collapsar: A vm-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention," *J. Parallel Distrib. Comput.*, vol. 66, pp. 1165–1180, Sept. 2006.

[48] Joyent, "Securing joyent cloud." `http://www.joyent.com/developers/security`.

[49] "Ubuntu juju," 2013. `https://juju.ubuntu.com`.

[50] Kanich, C., Kreibich, C., Levchenko, K., Enright, B., Voelker, G. M., Paxson, V., and Savage, S., "Spamalytics: An empirical analysis of spam marketing conversion.," *ACM Computer and Communications Security Conference*, August 2008.

[51] Keizer, G., "Russian 'cybermilitia' knocks kyrgyzstan offline.," *Computerworld*, 2009.

[52] KeyFocus, "Kfsensor product page." `http://www.keyfocus.net/kfsensor/`.

[53] Kibirkstis, A., "ntrusion detection faq: What is geolocation and how does it apply to network detection?," tech. rep., SANS Security Resources, November 2009.

[54] "Kippo," 2011. `https://code.google.com/p/kippo/`.

[55] Lakkaraju, K., Yurcik, W., and Lee, A. J., "Nvisionip: netflow visualizations of system state for security situational awareness," in *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security (VizSEC/DMSEC '04)*, pp. 65–72, ACM, 2004.

[56] "Linux advanced routing and traffic control," 2013. http://lartc.org.

[57] Li, Z., Goyal, A., and Chen, Y., "Honeynet-based botnet scan traffic analysis," in *Botnet Detection*, pp. 25–44, Springer, 2008.

[58] "libemu x86 shellcode emulation," 2010. http://libemu.carnivore.it.

[59] McAfee, I., "Operation aurora," tech. rep., McAfee Threat Center, 2010.

[60] Meyer, R., "Secure authentication on the internet," tech. rep., SANS Reading Room, http://www.sans.org/reading_room/whitepapers/securecode/secure-authentication-internet_2084, 2007.

[61] Microsoft, "Technical overview of the security features in the windows azure platform." http://www.windowsazure.com/en-us/support/legal/security-overview/.

[62] "Mongodb," 2010. http://www.mongodb.org.

[63] Nazario, J., "Botnet tracking: Tools, techniques, and lessons learned," tech. rep., Arbor Networks, 2007.

[64] "nepenthes - the finest collection," 2008. http://www.honeynet.org/project/nepenthes.

[65] NSA, "Defense in depth: A practical strategy for achieving information assurance in today's highly networked environments," tech. rep., National Security Agency, http://www.nsa.gov/ia/_files/support/defenseindepth.pdf, April 2011.

[66] Pinguelo, F. and Muller, B., "Virtual crimes – real damages: A primer on cybercrimes in the united states and efforts to combat cybercriminals," *Virginia Journal of Law and Technology, Forthcoming*, 2011.

[67] Provos, N., "A virtual honeypot framework," in *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13 (SSYM'04)*, vol. 13, p. 1, USENIX Association, 2004.

[68] Rackspace, "Security solutions and services." http://www.rackspace.com/security/domains/.

[69] Real, R. and Vargas, J. M., "The probabilistic basis of jaccard's index of similarity," *Systematic biology*, vol. 45, no. 3, pp. 380–385, 1996.

[70] Rimal, B. P., Choi, E., and Lumb, I., "A taxonomy and survey of cloud computing systems," in *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pp. 44–51, Ieee, 2009.

[71] Rist, L., "Glastopf," 2010. http://honeynet.org/sites/default/files/files/KYT-Glastopf-Final_v1.pdf.

[72] Rivera, J. and Van Der Meulen, R., "Gartner says the road to increased enterprise cloud usage will largely run through tactical business solutions addressing specific issues." http://www.gartner.com/newsroom/id/2581315.

[73] Robertson, J. and Svensson, P., "Targeted nature of email breach worries experts," *Associated Press*, April 2011.

[74] Rockwell, "Ethernet/ip: Industrial protocol white paper," 2013. http://literature.rockwellautomation.com/idc/groups/literature/documents/wp/enet-wp001_-en-p.pdf.

[75] Roesch, M., "Snort – lightweight intrusion detection for networks," in *13th USENIX Systems Administration Conference (LISA '99)*, 1999.

[76] Rosenbaum, R., "Secrets of the little blue box," *Esquire*, October 1971.

[77] Salesforce, "Security overview." https://trust.salesforce.com/trust/security/.

[78] Sampson, S., "Prettypark virus slowly gathering steam," *ZDNet*, June 1999.

[79] Specter, "Specter product page," 2009. http://www.specter.com/.

[80] Spitzner, L., "The honeynet project: Trapping the hackers," *Security & Privacy, IEEE*, vol. 1, no. 2, pp. 15–23, 2003.

[81] Spitzner, L., *Honeypots: tracking hackers*, vol. 1. Addison-Wesley Reading, 2003.

[82] Spitzner, L., "Honeynets in universities: Deploying a honeynet at an academic institution," *The Honeynet Project: Know Your Enemy Series*, 2004.

[83] Spitzner, L., "Know your enemy: Genii honeynets," *The Honeynet Project: Know Your Enemy Series*, 2005.

[84] Spitzner, L., "Know your enemy: Honeywall cdrom roo," *The Honeynet Project: Know Your Enemy Series*, 2005.

[85] Spitzner, L., "Know your enemy: Honeynets," *The Honeynet Project: Know Your Enemy Series*, 2006.

[86] Thonnard, O. and Dacier, M., "A framework for attack patterns' discovery in honeynet data," *digital investigation*, vol. 5, pp. S128–S139, 2008.

[87] "Virustotal," 2014. https://www.virustotal.com.

[88] Vokorokos, L. and Balaz, A., "Host-based intrusion detection system," in *14th International Conference on Intelligent Engineering Systems (INES)*, pp. 43–47, May 2010.

[89] Wang, K. and Stolfo, S., "Anomalous payload-based network intrusion detection," in *Recent Advances in Intrusion Detection, Lecture Notes in Computer Science*, pp. 203–222, Springer Berlin, 2004.

[90] Wikipedia, "Wikileaks," *Wikipedia*, 2011.

VITA

Albert Walter Brzeczko, Jr. received his Bachelors of Science in Electrical and Computer Engineering from Johns Hopkins University in December of 2004. After graduation, he worked for Raytheon Solipsys. In Fall of 2007, he enrolled at Georgia Tech, where he completed his Master of Science degree in Electrical and Computer Engineering in August of 2009 and concluded his studies in May of 2014 upon completion of his Ph.D.