Design and Analysis of Decoy Systems for Computer Security

Brian M. Bowen

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2011

ABSTRACT

Design and Analysis of Decoy Systems for Computer Security

Brian M. Bowen

This dissertation is aimed at defending against a range of internal threats, including eavesdropping on network taps, placement of malware to capture sensitive information, and general insider threats to exfiltrate sensitive information. Although the threats and adversaries may vary, in each context where a system is threatened, decoys can be used to deny critical information to adversaries making it harder for them to achieve their target goal. The approach leverages deception and the use of decoy technologies to deceive adversaries and trap nefarious acts. This dissertation proposes a novel set of properties for decoys to serve as design goals in the development of decoy-based infrastructures. To demonstrate their applicability, we designed and prototyped network and host-based decoy systems. These systems are used to evaluate the hypothesis that network and host decoys can be used to detect inside attackers and malware.

We introduce a novel, large-scale automated creation and management system for deploying decoys. Decoys may be created in various forms including bogus documents with embedded beacons, credentials for various web and email accounts, and bogus financial information that is monitored for misuse. The decoy management system supplies decoys for the network and host-based decoy systems.

We conjecture that the utility of the decoys depends on the believability of the bogus information; we demonstrate the believability through experimentation with human judges. For the network decoys, we developed a novel trap-based architecture for enterprise networks that detects "silent" attackers who are eavesdropping network traffic. The primary contributions of this system is the ease of injecting, automatically, large amounts of believable bait, and the integration of various detection mechanisms in the back-end. We demonstrate our methodology in a prototype platform that uses our decoy injection API to dynamically create and dispense network traps on a subset of our campus wireless network. We present results of a user study that demonstrates the believability of our automatically generated decoy traffic. We present results from a statistical and information theoretic analysis to show the believability of the traffic when automated tools are used.

For host-based decoys, we introduce BotSwindler, a novel host-based bait injection system designed to delude and detect crimeware by forcing it to reveal itself during the exploitation of monitored information. Our implementation of BotSwindler relies upon an out-of-host software agent to drive user-like interactions in a virtual machine, seeking to convince malware residing within the guest OS that it has captured legitimate credentials. To aid in the accuracy and realism of the simulations, we introduce a novel, low overhead approach, called virtual machine verification, for verifying whether the guest OS is in one of a predefined set of states. We provide empirical evidence to show that BotSwindler can be used to induce malware into performing observable actions and demonstrate how this approach is superior to that used in other tools. We present results from a user to study to illustrate the believability of the simulations and show that financial bait information can be used to effectively detect compromises through experimentation with real credential-collecting malware. We present results from a statistical and information theoretic analysis to show the believability of simulated keystrokes when automated tools are used to distinguish them.

Finally, we introduce and demonstrate an expanded role for decoys in educating users and measuring organizational security through experiments with approximately 4000 university students and staff.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This dissertation was made possible through the guidance and encouragement of my advisors, Angelos Keromytis and Sal Stolfo. I am grateful to them for the years of support they provided and their help in shaping this thesis.

Many people helped with efforts in this thesis, including: Vasileios Kemerlis, Pratap Prabhu, Vailis Pappas, Stelios Sidiroglou-Douskos, and Ramaswamy Devarajan. I would like to thank all of them for their hard work, contributions, and thoughtful insights.

A large part of a PhD program involves the community of people you get to know and work with. I would like to thank everyone in the Network Security Lab and Intrusion Detection Systems Lab for their support and for helping to make the PhD journey an enjoyable one.

I would also like to thank those who served on my PhD committee, including: Angelos Keromytis, Sal Stolfo, Fabian Monrose, Steven Bellovin, and Moti Young. I am especially grateful for the detailed feedback given by Steven Bellovin and Fabian Monrose.

Support for my thesis work was provided by Sandia National Laboratories through its Doctorate Study Program. I would like to offer a special thanks to everyone at Sandia who supported me in getting into and through the program, including: Kim Denton-Hill, Joselyne Gallegos, David Williams, Rob Leland, Ron Detry, Carol Jones, Bob Hutchinson, Carol Harrison, Jerriann Garcia, David White, David Duggan, as well as everyone on the Sandia Education Committee for their support, including: John Moser, Charline Wells, Pat Sena, Keith Bauer, Krystal Kelley, Diane Peebles, Mary Kay Austin, Rebecca Burt, Paul Yourick, Mark Garrett, Brian Damkroger, and Bernadette Montano.

I would like to thank my friends and family, especially my parents, Patricia and Raymond Bowen, for the lifelong support and encouragement they have provided. John Ziegler also provided helpful editorial reviews.

Finally, I am especially thankful to Leslie Ward for her perseverance over the past few years in enduring the pregnancy and delivery of our twins, Nathan and Chloe Bowen. She has overcome many challenges to support this work and our young twins.

Chapter 1

Introduction

The cyber domain provides a fertile environment for attacks aimed at information theft as a consequence of eavesdropping on networks and hosts. This dissertation is aimed at defending against a range of internal threats, including eavesdropping on network taps, placement of malware to capture sensitive information, and general insider threats to exfiltrate sensitive information.

Much research in computer security has focused on the means of preventing unauthorized and illegitimate access to systems and information. Unfortunately, the most damaging malicious activity is the result of internal misuse within an organization, perhaps since far less attention has been focused inward. Despite classic internal operating system security mechanisms and the body of work on formal specification of security and access control policies, including Bell-LaPadula [Bell and Whaley, 1982] and the Clark-Wilson models [Clark and Wilson, 1987], we still have an extensive insider attack problem. Indeed in many cases, formal security policies are incomplete and implicit or they are purposely ignored in order to get business goals accomplished. There seems to be little technology available to address the insider threat problem. As a result, insider abuse contributes significantly to the losses faced by many organizations today. According to the the annual Computer Crime and Security Survey for 2009 which surveyed 443 security personnel members from US corporations and government agencies, insider incidents were cited by 44 percent of respondents [Richardson, 2009]. The state-of-the-art seems to be still driven by forensics analysis after an attack, rather than technologies that prevent, detect, and deter insider attack.

The characteristics that distinguish insider attacks from traditional external attacks are knowledge and access of the attacker. Malicious Insiders possess greater knowledge of the systems and infrastructures they seek to exploit. They may already have access to the same systems systems either physically or through the network for legitimate reasons making defense a challenging task. Some external attackers can acquire insider characteristics by attaining internal network access. Many attacks use spyware and rootkits, which give outsiders internal access. Such software can easily be installed on systems from physical or digital media (*e.g.,* email, downloads) and allow an attacker to gain administrator or "root" access on a machine along with a capability to gather sensitive data. Rootkits have the ability to conceal themselves and elude detection, especially when the rootkit is previously unknown, as is true in zero-day attacks. An external attacker that manages to install rootkits internally in effect becomes an insider, thereby multiplying the ability to inflict harm. The creation and rapid growth of an underground economy that trades in stolen digital credentials has spurred the growth of spyware-driven bots that harvest sensitive data from unsuspecting users.

Traditional detection techniques rely on comparing signatures of known malicious instances to identify unknown samples, or on anomaly-based detection techniques in which host behaviors are monitored for large deviations from a baseline. Unfortunately, these approaches suffer a large number of known weaknesses. Signature-based methods can be useful when a signature is known, but due to the large number of possible variants, learning and searching all possible signatures to identify unknown binaries is intractable [Song *et al.*, 2007]. Anomaly-based methods are susceptible to false positives and negatives, limiting their potential utility. Consequently, a large amount of existing malware now operate undetected by antivirus software. A recent study focused of Zeus[1] (the largest botnet with over 3.6 million PC infections in the US alone [Messmer, 2009]), revealed that the malware bypassed up-to-date antivirus software 55% of the time [zeu, 2009].

Another drawback to conventional host-based antivirus software is that it typically mon-

[1]Zeus uses key-logging techniques to steal sensitive data such as user names, passwords, account numbers. It can be purchased on the black market for $600, complete with support and maintenance [abu, 2009].

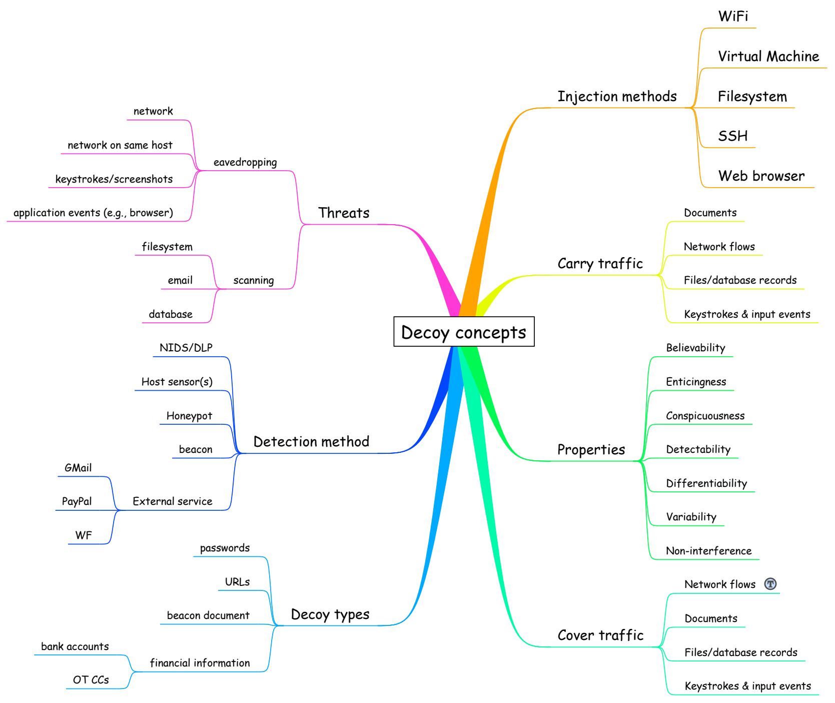


Figure 1.1: The variety of decoy concepts covered in this dissertation.

itors from within the host, making it vulnerable to evasion or subversion by malware. In fact, we see an increasing number of malware attacks that disable defenses such as antivirus software prior to undertaking some malicious activity [Ilett, 2005].

1.1 Contributions

The focus of this dissertation is on a defense system of an offensive nature, intended to confuse and deceive adversaries by leveraging uncertainty, to reduce the knowledge they ordinarily have of target systems, or they may be used to provide false information to an adversary that causes a detectable reaction. This dissertation focuses on the design and analysis of decoys to combat threats by deceiving adversaries with trap-laden misinformation that is detectable upon exploitation. The proposed decoys may be created in various forms including bogus documents with embedded beacons, credentials for various web and email accounts, and bogus financial information that is monitored for misuse.

The cyber landscape provides a vast number of settings in which decoys can be deployed. Naturally, the probability of exposing an attacker with trap-based defense tactics increases with the amount of decoy information that is generated and disseminated. The complexity of their deployment largely depends on the scale of the system and threat model. This dissertation synthesizes a variety of decoy concepts that range from various threat models, to types of decoys, their design, and their deployment. Figure 1.1 provides a synopsis of the variety of decoy concepts covered in this dissertation.

Most of this dissertation is aimed at deceiving attackers by convincing them something is real when in-fact it is not. To show how decoys are generally valuable to security, we inverse the role of decoys and show how they can be used to educate users and provide valuable metrics. Here, we create decoys that mimic attackers' actions as opposed to those of legitimate users. Subsequent chapters will explore each of this concepts in detail.

Thesis Statement: *Network and host decoys can be used to detect malicious actions by inside attackers or malware seeking to extend access, and to educate innocent users on potentially vulnerable actions. Although the threats and adversaries may vary, in each context where a system is threatened, decoys can be used to deny critical information to adversaries making it harder for them to achieve their target goal.*

In summary, this dissertation includes the following contributions:

- A novel set of generally applicable properties are proposed to guide the design and deployment of decoys and maximize the deception they induce for different classes of insiders who vary by their level of knowledge and sophistication.

- A large-scale automated creation and management system for deploying decoys that can indicate malicious insider activity. This provides a means for ordinary users to deploy decoy documents without having to setup sophisticated honeypot systems and sensors.

- The use of the decoy properties to measure the success of the proposed decoy systems. In particular, we focus on the two most important properties of decoys – believability and detectability – for metrics on which the systems are evaluated.

- A novel architecture based on a "record, modify, replay" paradigm to automatically generate large quantities of decoy traffic that are injected into the network. The system continuously regenerates decoys to prevent an adversary from learning how to recognize bait over time. We analyze the believability of the generated traffic with human judges and present results from field experiments. We provide a statistical analysis to show the believability of the traffic when automated tools are used.

- A novel approach for malware detection that relies on the use of decoy injection whereby bogus information is used to bait and delude information stealing malware, forcing it to reveal itself during the exfiltration or exploitation of the monitored information. We demonstrate the believability of the simulations experimentally with human judges and statistical means. We show malware can be detected with various types of web and financial decoys.

- A novel approach to measuring an organization's security posture using decoys that demonstrates an expanded role of decoys for providing utility in measuring security and trapping user mistakes for educational purposes.

1.2 Dissertation Organization

This dissertation lays out a design for host and network deception infrastructure and is organized as follows. In Chapter 2, we survey work related to each of the contribution areas. Chapter 3 introduces a core set of properties which are used as design goals in the systems described in the following chapters. Chapters 5 and 6 describe proposed systems to demonstrate these properties. The systems are designed to enable the automatic generation and injection of network and host decoys. As part of these chapters, results are presented that attest to the believability of the generated decoys and their ability to detect attackers. Chapter 7 discusses an expanded role for decoys for educating users and measuring organizational security. Chapter 8 concludes this dissertation with a summary of results and contributions.

Part I

Related Work and Decoy Properties

Chapter 2

Related Work

The use of deception, or decoys, plays a valuable role in the protection of systems, networks, and information. The first use of decoys (*i.e.,* in the cyber domain) has been credited to Cliff Stoll [Yuill *et al.*, 2004; Spitzner, 2003b] and detailed in [Stoll, 1988], where he provides a thorough account of his crusade to catch German hackers breaking into Lawrence Berkeley Laboratory computer systems. Stoll's methods included the use of bogus networks, systems, and documents to gather intelligence on the German attackers who were apparently seeking state secrets. Among the many techniques waged, he crafted "bait" files, or in his case, bogus classified documents that really contained non-sensitive government information and attached "alarms" to them so that he would know if anyone accessed at them. To Stoll's credit, a German hacker was eventually caught and it was found that he had been selling secrets to the KGB.

2.1 Decoy Properties

In this proposal, a set of generally applicable decoy properties are introduced to guide the design of decoys and maximize the deception they induce for different classes of insiders who vary by their level of knowledge and sophistication. Bell and Whaley [Bell and Whaley, 1982] have described the structure of deception as a process of hiding the real and showing showing the false. They introduce several methods of hiding that include masking, repackaging, and dazzling, along with three methods of showing that include mimicking, inventing, and decoying. Yuill *et al.* [Yuill *et al.*, 2006] expand upon this work and characterize deceptive hiding in terms of how it defeats an adversary's discovery process. They describe an adversary's discovery process as taking three forms: direct observation, investigation based on evidence, and learning from other people or agents. Their work offers a process model for creating deceptive hiding techniques based on how they defeat an adversary's discovery process.

2.2 Decoy Documents

The decoy documents introduced in this dissertation utilize similar deception mechanisms as well as beacons to signal a remote detect and alert in real-time time when a decoy has been opened. Web bugs are a class of silent embedded tokens which have been used to track usage habits of web or email users [McRae and Vaughn, 2007]. Unfortunately, they have been most closely associated with unscrupulous operators, such as spammers, virus writers, and spyware authors who have used them to violate users privacy. Typically they will be embedded in the HTML portion of an email message as a non-visible white on white image, but they have also been demonstrated in other forms such as Microsoft Word, Excel, and PowerPoint documents [Smith, 2000]. When rendered as HTML, a web bug triggers a server update which allows the sender to note when and where the web bug was viewed. Animated images allow the senders to monitor how long the message was displayed. The web bugs operate without alerting the user of the tracking mechanisms. The advantage for legitimate advertisers is that this allows them to monitor advertisement effectiveness, while privacy advocates worry that this technology can be misused to spy on users' habits. Our work leverages the same ideas, but extends them to other document classes and is more sophisticated in the methods used to draw attention. In addition, our targets are insiders who should have no expectation of privacy on a system they violate.

2.3 Decoy Networking

The goal of our work is to design a system for generating network traps as a means of proactive defense against snoopers. *Traffic generation* has long been studied for a variety of tasks that include traffic engineering [Medina *et al.*, 2002] (*e.g.,* load balancing, routing protocols configuration), network simulation, emulation [Vahdat *et al.*, 2002], and many more. To support these applications, many software tools have been created ranging from customizable packet generators, such as *Hping* [Hping,] and *Scapy* [Scapy,], to large-scale network emulators such as *ModelNet* [Vahdat *et al.*, 2002]. Other tools, including *Swing* [Vishwanath and Vahdat, 2009], focus solely on traffic generation, but with the end goal of realistic TCP/IP or UDP values and statistically accurate timing measures. Similarly, *Harpoon* [Sommers and Barford, 2004] is a traffic generation tool for creating packet flows with byte, packet, temporal, and spatial characteristics that match those from existing *netflow* or packet trace data. Although the goals of realistic TCP/IP values overlap with ours, generating believable decoys additionally requires realistic application-layer content. The requirements of which vary from those of the preceding traffic generation efforts, adding to the novelty of our research.

Deception-based information resources that have no production value other than to attract and detect adversaries (like those used by Stoll) are commonly known as *honeypots.* Honeypots serve as effective tools for profiling attacker behavior and to gather intelligence for understanding how attackers operate. Honeypots are considered to have low false positive rates since they are designed to capture only malicious attackers, except for perhaps an occasional mistake by innocent users. Spitzner described how honeypots can be useful for detecting insider attacks [Spitzner, 2003a], in addition to the common external threats for which they are traditionally known. He discusses the use of honeytokens, which he defines as "a honeypot that is not a computer" [Spitzner, 2003b], citing examples that include bogus medical records, credit card numbers, and credentials, with descriptions of how they can be used to detect malicious insiders. Oudot [Oudot, 2004] gave a simple example of how honeypots can be used on wireless networks, but in this case, all of the sessions are the same, making them trivial to avoid. Grundschober [Grundschober and Dacier, 1998] created a sniffer detector for wired networks that relied on simple scripts to create telnet and ftp sessions with bait information, however no attention was given to the believability of the sessions, making them easy to avoid. More importantly, the detector relied on a network intrusion detection system to detect decoy misuse on the network rather than misuse at the application layer, as we do; the benefits of which are discussed in Section 4.2.

Currently, the decoy/honeytoken creation is a laborious and manual process requiring large amounts of administrator intervention. In contrast, we have devised a system that automatically generates and disseminates, continuously, decoy information (of various different types) throughout an operational network to create indistinguishable *honeyflows*. Indeed, it is the *indistinguishability* of our honeyflows, the volume at which they can be produced, and the non-interference with real flows that makes our work novel.

2.4 Host-based Decoys

Deception-based information resources that have no production value other than to attract and detect adversaries are commonly known as honeypots. Honeypots serve as effective tools for profiling attacker behavior and to gather intelligence to understand how attackers operate. They are considered to have low false positive rates since they are designed to capture only malicious attackers, except for perhaps an occasional mistake by innocent users. Spitzner discusses the use of honeytokens [Spitzner, 2003b], which he defines as "a honeypot that is not a computer," citing examples that include bogus medical records, credit card numbers, and credentials. Our work harnesses the honeytoken concept to detect crimeware that may otherwise go undetected.

Injecting human input to detect malware has been shown to be useful by Borders *et al.* [Borders *et al.*, 2006] with their Siren system. The aim of Siren is to thwart malware that attempts to blend in with normal user activity to avoid anomaly detection systems. However, detection is performed by manually injecting human input to generate a sequence of network requests and observing the resulting network traffic to identify differences from the known sequences of requests; deviations are flagged as malicious. Expanding upon Siren, Chandrasekaran *et al.* [Chandrasekaran *et al.*, 2007], developed a system to randomize generated human input to foil potential analysis techniques that may be employed by malware. The work by Holz *et al.* [Holz *et al.*, 2009] to investigate keyloggers and dropzones, relied on executing maleware in CWSandbox [Willems *et al.*, 2007] and automating user input with AutoIt [1]. However, it was limited to ad hoc scenarios designed for the sole purpose of detecting harvesting channels. Their approach depends on miss-configured and insecure dropzone servers to learn about what sort of information is being stolen. While this effort did reveal interesting details about stolen information, it is limited by law and skill of the attackers (*i.e.,* they can just secure their dropzone servers). In addition, relying on simulator software that resides within the host, such as AutoIt, provides attackers with a simple means to detect and avoid it. In contrast to these systems, BotSwindler is difficult to detect, automatically injects input designed to be believable, relies on monitored decoy credentials for detection, and provides a platform to convince malware that it has captured legitimate credentials.

Taint analysis is another technique that has been used to detect credential stealing malware. Egele *et al.* [Egele *et al.*, 2007] used taint analysis to track information as it is processed by the web browser and loaded in to browser helper objects (BHOs). Their approach allows for a human analyst to observe where information is being sent in offline analysis. Similarly, Yin *et al.* [Yin *et al.*, 2007] built Panorama, a taint tracking system that extends beyond BHOs to handle tracking throughout multiple processes, memory swapping, and disks. These systems may work well to track information in a system, but they do so with large overhead (factor of 10-20 slowdown in the systems described) or contain components that reside on the guest [Yin *et al.*, 2007]; both these features that can be detected by malware and used for evasion purposes.

BotSwindler injects monitored bait into VM-based hosts by simulating user activity that is of interest to crimeware. The simulation is performed on the native OS outside of the VM to minimize artifacts that could be used to tip-off resident malicious software. To keep track of the simulation state within the virtual environment, our approach relies on a form of virtual machine introspection (VMI), a concept proposed by Garfinkel *et al.*[Garfinkel and Rosenblum, 2003] to describe the act of inspecting a virtual machine's software from outside the virtual environment. The challenge of VMI lies in overcoming the semantic gap [Chen and Noble, 2001] between the two levels of abstraction represented by the VM and the underlying service or OS. Garfinkel *et al.* focused on inspecting memory, registers, device

[1]`http://www.autoitscript.com`

state, and other process related information to implement an attack resistant host-based IDS for VMs whereby the IDS is located outside of the guest in the virtual machine monitor (VMM). Other VMI implementations include [Jones *et al.*, 2006; Jiang and Wang, 2007; Srivastava and Giffin, 2008; Payne *et al.*, 2007; Krishnan *et al.*, 2010], but unlike most of these approaches, we circumvent the semantic gap and rely on artifacts found in the VMM graphical framebuffer. To the best of our knowledge, we are the first to focus on the verification of state for user simulations, a challenge with unique requirements.

2.5 Educating Users and Measuring Organizational Security

In Chapter 7, we introduce our system for educating users and measuring organizational security using decoy emails. Traditional security training classes can be beneficial for organizations, but they are not enough and there are more effective methods [Kumaraguru *et al.*, 2007]. Our technique involves testing users' vulnerability using a variety of decoy emails; those that fall victim to our phony phishing attacks are informed so that they may learn and change their behavior. Traditional approaches for training users about the threat posed by phishing rely on classes and informational warnings. Efforts to raise user awareness have focused on testing users to demonstrate their vulnerability [New York State Office of Cyber Security & Critical Infrastructure Coordination, 2005]. Some tools have been created to support the sending of fake phishing emails for purposes of pen testing and training[Core Security, 2010; Phishme.com, 2011], but these rely on an administrator to manually construct and send the emails to targeted individuals. None of these tools focus on the development of formal metrics for measuring organizational security such that they can be used for relative comparisons for comparing one organization against another.

Chapter 3

Design Goals

In this chapter, we introduce our threat model that defines various levels of attackers that differ by their level of knowledge and sophistication. We introduce a novel set of generally applicable properties to guide the design and deployment of decoys and maximize the deception they induce for different classes of attackers.

3.1 Threat Model - Level of Sophistication of the Attacker

The insider seeks to identify and avoid the decoys and abscond with "real" information. We broadly define four monotonically increasing levels of insider sophistication and capability. Some will have tools available to assist in deciding what is a decoy and what is real. Others will only have their own observations and thoughts.

- **Low**: Direct observation is the only tool available. The adversary largely depends on what can be gleaned from a first glance. We strive to defeat this level of adversary with our beacon documents, even though decoys with embedded beacons may be distinguished with more advanced tools.

- **Medium**: A more thorough investigation can be performed by the insider; decisions based on other, possibly outside evidence, can be made. For example, if a decoy document contains a decoy account credential for a particular identity, an adversary may verify that the particular identity is real or not by querying an external system (such as www.whitepages.com). Such adversaries will require stronger decoy information possibly corroborated by other sources of evidence.

- **High**: Access to the most sophisticated tools are available to the attacker (*e.g.,* super computers, other informed people who have organizational information). The notion of the "Perfect Decoy" described in the next section may be the only indiscernible decoy by an adversary of such caliber.

- **Highly Privileged**: Probably the most dangerous of all is the privileged and highly sophisticated user. Such attackers might even be aware that the system is baited and will employ sophisticated tools to try to analyze, disable, and avoid decoys entirely. As an example of how defeating this level of threat might be possible, consider the analogy with someone who knows encryption is used (and which encryption algorithm is used), but still cannot break the system because they do not have knowledge of an easy-to-change operational parameter (the key). Likewise, just because someone knows that decoys are used in the system does not mean they should be able to identify them. This is the principal– coming up with a scheme to satisfy it remains an open problem.

3.2 Decoy Properties

One of the major contributions of this thesis is the identification and formal definition of core properties of a decoy that will successfully bait inside attackers. We enumerate various properties and means of measuring these properties that are associated with decoys to ensure their use will be likely to snare an attacker. These properties serve as goals for decoys and systems described in later chapters. Although we define the properties in the context of our decoy documents, they are directly applicable to the rest of the decoys used throughout this work. We note the differences where they may exist. We introduce the following notation for these definitions.

Believable[1]: Capable of eliciting belief or trust; capable of being believed; appearing true; seeming to be true or authentic.

[1]For clarity, each property is provided with its definition gleaned from online dictionary sources.

A good decoy should make it difficult for an adversary to discern whether they are looking at a legitimate source or if they are indeed looking at a decoy. We conjecture that believability of any particular decoy can be measured by adversary's failure to discern one from the other. We formalize this by defining a decoy believability experiment. The experiment is defined for the document space M with the set of decoys D such that $D \subseteq M$ and $M - D$ is the set of authentic documents or credentials.

The Decoy Believability Experiment: $\mathbf{Exp}_{A,D,M}^{believe}$

- For any $d \in D$, choose two documents $m_0, m_1 \in M$ such that $m_0 = d$ or $m_1 = d$, and $m_0 \neq m_1$; that is, one is a decoy we wish to measure the believability of and the second is chosen at random from the set of authentic documents.
- Adversary A obtains m_0, m_1 and attempts to choose $\hat{m} \in \{m_0, m_1\}$ such that $\hat{m} \neq d$, with an effort bounded by cost c (time, computing power, etc).
- The output of the experiment is 1 if $\hat{m} \neq d$ and 0 otherwise.

For concreteness, we build upon the definition of "Perfect Secrecy" proposed in the cryptography community [Katz and Lindell, 2007] and define a "perfect decoy" when:

$$\Pr[\mathrm{Exp}_{A,D,M}^{believe} = 1] = 1/2$$

The decoy is chosen in a believability experiment with a probability of 1/2 (the outcome that would be achieved if the volunteer decided completely at random). That is, a perfect decoy is one that is completely indistinguishable from one that is not. A benefit of this definition is that the challenge of showing a decoy to be believable, or not, reduces to the problem of creating a "distinguisher" that can decide with probability better than 1/2.

In practice, the construction of a "perfect decoy" might be unachievable, especially through automatic means, but the notion remains important as it provides a goal to strive for in our design and implementation of systems. For many threat models, it might suffice to have less than perfectly believable decoys.

We note that the believability property of a decoy may be less important than other properties defined below since the attacker may have to open the decoy in order to decide whether the document is real or not. The act of opening a document or testing a decoy may be all that we need to trap an adversary, irrespective of the believability of its content. Hence, enticing an attacker to open a decoy, say one with a very interesting name, may be a more effective strategy to detect an attacker than producing a decoy document with believable content.

This definition introduced the notion of cost as a means of bounding an attackers ability. Cost may be taken to mean the level of effort, knowledge, or financial means required by the attacker to achieve success.

Chapters 5.3 and 5.4 demonstrate how believability can be evaluated for decoy network traffic. Chapters 6.2 and 6.3 demonstrate the evaluation of believability for decoys injected into hosts.

Enticing: highly attractive and able to arouse hope or desire; "an alluring prospect"; lure.

Herein lies the issue of how does one measure the extent to which a decoy arouses desires, how well is it a lure? One obvious way is to create decoys containing information with monetary value, such as passwords or credit card numbers that have black market value [Symantec, 2008]. This is the case for some of the decoys used in this work; they are known to have value in the underground economy making them enticing targets for cyber criminals.

Enticement also depends upon the attacker's intent or preference. We define enticing documents in terms of the likelihood of an adversary's preference; enticing decoys are those decoys that are chosen with the same likelihood. More formally, for the document space M, let P be the set of documents of an adversary's A preference, where $P \subseteq M$. For some value ϵ such that $\epsilon > 1/|M|$, an enticing document is defined by the probability

$$\Pr[m \to M | m \in P] > \epsilon$$

where $m \to M$ denotes m is chosen from M. An enticing decoy is then defined for the set of decoys D, where $D \subseteq M$, such that

$$\Pr[m \to M | m \in P] = \Pr[d \to M | d \in D]$$

We posit that by defining several general categories of "things" that are of "attacker interest", one may compose decoys using terms or words that correspond to desires of the attacker that are overwhelmingly enticing. For example, if the attacker desires money, any document that mentions or describes information that provides access to money should be highly enticing. We believe we can measure frequently occurring (search) terms associated with major categories of interest (*e.g.*, words or terms drawn from finance, medical information, intellectual property) and use these as the constituent words in decoy documents. To measure the effectiveness of this generative strategy, it should be possible to execute content searches and count the number of times decoys appear in the top 10 list of displayed documents. This is a reasonable approach also, to measuring how conspicuous, defined below, the decoys become based upon the attacker's searches associated with their interest and intent.

Conspicuous: easily visible; easily or clearly visible; obvious to the eye or mind; Attracting attention.

A *conspicuous* decoy should be easily found or observed. Conspicuous is defined similar to enticing, but conspicuous documents are found because they are easily observed, whereas enticing documents are chosen because they are of interest to an attacker. For the document space M, let V be the set of documents defined by the minimum number of user actions required to enable their view. We use a subscript to denote the number of user actions required to view some set of documents. For example, documents that are in view at logon or on the desktop (requiring zero user actions) are labeled V_0, those requiring one user action are V_1, etc. We define a "view", V_i of a set of documents as a function of a number of user actions applied to a prior view, V_{i-1}, hence

$$V_i = \text{Action}(V_{i-1}) \text{ where } V_j \neq V_i, j < i$$

An "Action" may be any command or function that displays files and documents, such as 'ls', 'dir', 'search.' For some value ϵ such that $\epsilon > 0$, a conspicuous document, d, is defined by the probability

$$\prod_{i=0}^{n} \Pr[V_i] > \epsilon$$

where n is the minimum value where $d \in V_n$. Note if d is on the desktop, V_0, $\Pr[V_0] = 1$ (*i.e.*, the documents in full view are highly conspicuous).

When a user first logs in, a conspicuous decoy should either be in full view on the desktop, or viewable after one (targeted) search action. One simple user action is optimal for a highly conspicuous decoy. Thus, a measure of conspicuousness may be a count of the number of search actions needed, on average, for a decoy to appear in full view. The decoy may be stored in the file system anywhere if a simple content-based search locates it in one step. But, this search act depends upon the query executed by the user. The query can either be a location (*e.g.*, search for a directory named "TAX" in which the decoy appears) or a content query (*e.g.*, using Google Desktop Search for documents containing the word "TAX.") In either case, if a decoy document appears after one such search, it is conspicuous. Hence, we may define the set P as all such files that can be found in some number of steps. But, this depends upon what search terms the attacker uses to query! If the decoy never appears because the attacker used the wrong search terms, the decoy is not conspicuous. We posit that the property of *enticing* is likely the most important property, and a formal measure to evaluate enticement will generate better decoys. In summary, an enticing decoy should be conspicuous to be an effective decoy trap.

Detectable; to discover or catch (a person) in the performance of some act: to detect someone cheating.

Decoys must ensure an alert is generated if they are exploited. Formally, this is defined for adversary A, document space M, and the set of decoys D such that $D \subseteq M$. We use $Alert_{A,d} = 1$ to denote an alert for $d \in D$. We say d is detectable with probability ϵ when

$$\Pr[d \to M : Alert_{A,d} = 1] \geq \epsilon$$

Ideally, ϵ should be 1. We seek to maximize ϵ with a strategy of "detection in depth."

We designed the decoy documents with several techniques to provide a good chance of detecting the malfeasance of an inside attack in real-time.

Chapters 5.2 demonstrate detectability for networking-based decoys. Chapters 6.5 and 4.4 demonstrate the detectability of host-based decoys.

Variability: The range of possible outcomes of a given situation; the quality of being subject to variation.

Attackers are humans with insider knowledge, even possibly with the knowledge that decoys are liberally spread throughout an enterprise. Their task is to identify the real documents from the potentially large cache of decoys. One important property of the set of decoys is that they are not easily identifiable due to some common invariant information they all share. A single search or test function would thus easily distinguish the real from the fake. The decoys thus must be highly varied. We define variable in terms of the likelihood of being able to decide the believability of a decoy given *any* known decoy. Formally, we define *perfectly variable* for document space M with the set of decoys D such that $D \subseteq M$ where

$$\Pr[d' \to D : \mathrm{Exp}^{believe}_{A,D,M,d'} = 1] = 1/2$$

Observe that, under this definition, an adversary may have access to *all* N previously generated decoys with the knowledge they are bogus, but still lack the ability to discern the N+1st. From a statistical perspective, each decoy is independent and identically distributed. For the case that an adversary can determine the N+1st decoy only after observing the N prior decoys, we define this as an *N-strong Variant.*

Non-interference: Something that does not hinder, obstructs, or impede.

Introducing decoys to an operational system has the potential to *interfere* with normal operations in multiple ways. Of primary concern is that decoys may pollute authentic data so that their legitimate usage becomes hindered by corruption or as a result of confusion by legitimate users (*i.e.,* they cannot differentiate real from fake). We define non-interference in terms of the likelihood of legitimate users successfully accessing normal documents after decoys are introduced. We use $\text{Access}_{U,m} = 1$ to denote the success of a legitimate user U accessing a normal document m. More formally, for some value ϵ, the document space M, $\forall m \in M$ we define

$$\Pr[Access_{U,m} = 1] \geq \epsilon$$

on a system without decoys. Non-interference is then defined for the set of decoys D such that $D \subseteq M$ and $\forall m \in M$ we have

$$\Pr[Access_{U,m} = 1] = \Pr[Access_{U,m} = 1 | D]$$

Although we seek to create decoys to ensnare an inside attacker, a legitimate user whose data is the subject of an attacker must still be able to identify their own real documents from the planted decoys. The more enticing or believable a decoy document may be, the more likely it would be to lead the user to confuse it with a legitimate document they were looking for. Our goal is to increase believability, conspicuousness, and enticingness while keeping interference low; Ideally a decoy should be completely non-interfering.

Chapter 5.5 demonstrates the use of this property by measuring the amount of non-interference in networking-based decoys.

Differentiable: to mark or show a difference in; constitute a difference that distinguishes; to develop differential characteristics in; to cause differentiation of in the course of development.

It is important that decoys be "obvious" to the *legitimate user* to avoid interference, but "unobvious" to the attacker stealing information. We define this in terms of an inverted believability experiment, in which the adversary is replaced by a legitimate user. We say a decoy is differentiable if the legitimate user always succeeds. Formally, we state this for the document space M with the set of decoys D such that $D \subseteq M$ where

$$\Pr[\text{Exp}_{U,D,M}^{believe} = 1] = 1$$

How might we easily differentiate a decoy for the legitimate user so that we maintain "non-interference" with the user's own actions and legitimate work? This depends on the particular type of decoy. The remote thief who exfiltrates all of a user's files onto a remote hard drive may be perplexed by having hundreds of decoys amidst a few real documents; the thief should not be able to easily differentiate between the two cases. If we store a hundred decoys for each real document, the thief's task is daunting; they would need to test embedded information in the documents to decide what is real and what is not, which should complicate their end goals. For clarity, decoys should be easily *differentiable* to the legitimate user, but not to the attacker without significant effort.

Expiration: the ending of the period of time for which a decoy is valid.

Decoys may become invalid as a result of an attacker identifying it as a decoy, causing the decoy to fail the believability test. In certain cases, the expiration of decoy may be represented as a *half-life*, or the time it takes half of the potential attackers to identify it has being bogus. In other cases, an expiration of a decoy might be defined at its creation time by its creator. For example, a bogus account might be created that it is automatically deactivated after a fixed period of time. Such would be the case for decoy credit cards that automatically become invalid at some predefined date. Formally, we state this in respect to the believability and detectability expirations. After the expiration, a decoy either becomes distinguishable or fails at detection. Formally, for the document space M with the set of decoys D such that $D \subseteq M$ for a given time t and expiration E as:

$$\Pr[\mathrm{Exp}^{believe}_{A,D,M} = 1] = 1/2 \wedge \Pr[d \rightarrow M : Alert_{A,d} = 1] \geq \epsilon \text{ when } t < E$$

and

$$\Pr[\mathrm{Exp}^{believe}_{A,D,M} = 1] = 1 \vee \Pr[d \rightarrow M : Alert_{A,d} = 1] = 0 \text{ when } t \geq E$$

Some of the decoys introduced in Chapter 4 are considered expired after an alert is triggered. The alert indicates that an attacker has learned the username and password for the decoy account and that the account has a zero balance. These decoys can be refreshed by changing the password. In some cases, the refreshing of a decoy may not be possible if the attacker manages to change the password and block our access.

Cost: the price or level of effort required to acquire or produce a particular decoy.

The construction and deployment of decoys comes at an expense that can be measured financially or as a level effort required by the legitimate users to create and maintain operational decoys. This is not to be confused with the cost used in the believability definition, which is the cost the attacker must pay to be able to distinguish a decoy. The cost of a decoy system largely depends on its scale and each of the properties.

3.3 Design Goals Summary

We have introduced a novel set of properties to guide in the design of decoy systems. It remains an open problem to prove the completeness or incompleteness of this set. The remainder of this dissertation focuses on the use of these properties in designing decoy-based systems. In particular, we focus on the two most important properties – detectability and believability. Chapters 5.3 and 5.4 focus on evaluating the believability of decoy network traffic. Chapters 6.2 and 6.3 focus on evaluating the believability decoys injected into hosts. Chapters 5.2 demonstrate detectability for networking-based decoys. Chapters 6.5 and 4.4 demonstrate the detectability of host-based decoys. The remaining properties are important, but we do not necessarily demonstrate them through experimental analysis except where it is appropriate. In many cases, their proof is trivial and we focus on their applicability.

Part II

Decoy Systems

Chapter 4

Design and Generation of Decoys

One of the core systems of this dissertation is the Decoy Document Distributor ($\texttt{D}^3$) System, a web-based service for generating, distributing, and monitoring decoys. $\texttt{D}^3$ can be used by registered users to generate decoys for download, or as a decoy data source for the host and network components. To achieve the goal of wide spread distribution of decoys a variety of methods are considered to trap potential attackers with varying levels of sophistication. The contributions of this system include:

- A large-scale automated creation and management system for deploying decoys that can detect the presence (and, in some cases, "identity") of malicious insiders, or at least indicate malicious insider activity. This provides a means for ordinary users to deploy honey documents without having to setup sophisticated honeypot systems and sensors.

- An offensive trap-based defense system is proposed to detect masqueraders and traitors, and to flood attackers with bogus exfiltrated information that they must analyze in order to find real information of value. Hence, our long term goal is to flood the miscreant marketplace with bogus information devaluing their quarry.

- A design of decoy information that combines a number of methods and monitors, both internal and external, to detect insider exploitation using a common and ubiquitous set of baited targets, ordinary looking documents.

1. A watermark is embedded in the binary format of the document file to detect when the decoy is loaded in memory, or egressed in the open over a network.
2. A "beacon" is embedded in the decoy document that signals a remote web site upon opening of the document indicating the malfeasance of an insider illicitly reading bait information.
3. If these methods fail to detect an insider attack or an exfiltration of baited documents, the content of the documents contain bait and decoy information that is monitored as well. Bogus logins at multiple organizations as well as bogus and realistic bank information is monitored by external means.

- An easy to use system to broadly deploy decoys to ordinary users who are alerted by email when a decoy has been touched on their laptops and personal computers; no such system presently exists.
- Various monitors for detecting the (mis)use of Gmail, Payal, Bank, and university bait credentials.

4.1 Decoy Documents

The primary goal of the trap based defense is to detect malfeasance. Since no system is foolproof, we propose that multiple overlapping signals be embedded in the decoy documents to ensure *detectability.* Any alert generated by the multiple decoys is an indicator that some insider activity has occurred. Since the attacker may have varying levels of sophistication, a combination of traps are used in decoy documents to increase the likelihood one will succeed in generating an alert. A sophisticated attacker may, for example, disable the internal beacon, or cut off network connections avoiding communication, disable or kill local host monitoring processes, or they may exfiltrate documents via a web-browser without opening them locally. The documents are designed with several means of detecting their misuse:

- embedded honeytokens, computer login accounts created that provide no access to valuable resources, and that are monitored when (mis)used;

- embedded honeytoken banking login accounts specifically created and monitored for this trap-based technology demonstration specifically to entice financially motivated attackers;
- a network-level egress monitor that alerts whenever a marker, specially planted in the decoy document, is detected. Snort may be used as simple signature detector;
- a host-based monitor that alerts whenever a decoy document is "touched" in the file system such as a copy operation;
- an embedded "beacon" alerts a remote server at a site at Columbia and recorded on $\mathtt{D}^3$. The web site emits an email to the registered user who created and downloaded the decoy document.

The implementation of features are described below.

4.1.1 Honeytokens

This layer of defense is made up of "bait" information such as online banking logins provided by a collaborating financial institution, credit card numbers, login accounts for online servers, and web based email accounts. The primary requirement for bait is that it be detectable when (mis)used. The various types of information used are discussed in a Chapter 4.2.

4.1.2 Beacon Implementation

The highly sophisticated attacker will likely attempt to differentiate between a real document and a decoy by analyzing the binary file format prior to opening a file. This necessitates a design where beacon code and watermarks in decoy documents are hidden to avoid their easy identification. The attacker would surely avoid the decoys if they could easily identify them by a simple static test for an embedded beacon. The beacon code can be embedded in documents in a number of ways and made to appear statistically equivalent to its surrounding data using a blending technique called "spectrum shaping" (see [Song *et al.*, 2007; Detristan *et al.*, 2003]). Such obfuscation techniques are very hard to defeat [Li *et al.*, 2007].

Using common techniques developed for malware, beacons attempt to silently contact a centralized server with a unique token embedded within the document at creation time. The token is used to identify the decoy and document, the IP address of the host accessing the decoy document. Depending on the particular document type and the rendering environment used during viewing of the beacon document, some additional data may be collected.

The first proof-of-concept beacons have been implemented in MS Word and PDF and deployed through the $\texttt{D}^3$ system. In the case of the MS Word document beacons, the examples rely on a stealthily embedded remote image that is rendered when the document is opened. The request for the remote image is a positive indication the document has been opened. In the case of PDF document beacons, the signaling mechanism relies on the execution of Javascript within the document. The $\texttt{D}^3$ site includes a tutorial guiding the user on how to generate, download, and enable the decoys' silent communication on hosts. It is important to point out that there are methods for disabling the beacon mechanism. In Section 4.2.3, we provide an evaluation of beacon robustness.

4.1.3 Embedded Marker implementation

Beacon documents contain embedded markers that a host or network sensor may detect either when documents are loaded in memory or transmitted in the clear. The markers are constructed as a unique pattern of word tokens uniquely tied to the document creator. The sequence of word tokens is embedded within the beacon document's meta-data area or reformatted as comments within the document format structure. Both locations are ideal for embedding markers since most rendering programs ignore these parts of the document. The embedded markers can be used in Snort signatures for detecting exfiltration.

4.2 Trap-based Decoys

Our trap-based decoys are detectable outside of a host by external monitors, so they do not require host monitoring nor do they suffer the performance burden characteristic of decoys that require constant internal monitoring (such as those used for taint analysis). They are made up of *bait information* including online banking logins provided by a collaborating financial institution, login accounts for online servers, and web based email accounts. For the experiments in this thesis, we focused on the use of decoy Gmail credentials, PayPal credentials, and banking credentials. These were chosen because they are widely used and known to have underground economy value [Symantec, 2008; Holz *et al.*, 2009], making them alluring targets for crimeware, yet inexpensive for us to create. The banking logins are provided to us by a collaborating financial institution. As part of the collaboration, we receive daily reports showing the IP addresses and timestamps for all accesses to the accounts at any time.

The decoy PayPal and bank accounts have an added bonus that allows us to expose the credentials without having to be concerned about an attacker changing their password. PayPal requires multi-factor authentication to change the passwords on an account. Yet, we do not reveal all of the attributes of an account making it difficult for an attacker to change the authentication credentials. For the banking logins, we have the ability to manage the usernames and passwords.

Custom monitors for PayPal and Gmail accounts were developed to leverage internal features of the services that provide the time of last login, and in the case of Gmail accounts, the IP address of the last login. In the case of PayPal, the monitor logs into the decoy accounts every hour to check the PayPal recorded last login. If the delta between actual and expected times is greater than 75 seconds, the monitor triggers an alert for the account and notifies us by email. The 75 second threshold was chosen because PayPal reports the time to a resolution of minutes rather than seconds. The choice as to what time interval to use and how frequently to poll presents significant tradeoffs that we analyze in Chapter 4.2.1.

In the case of the Gmail accounts, custom scripts access `mail.google.com` to parse the bait account pages, gathering account activity information. The information includes the IP addresses for the previous 5 account accesses and the time. If there is any activity from IP addresses other than the monitor's host IP, an alert is triggered with the time and IP of the offending host. Alerts are also triggered when the monitor cannot login to the bait account. In this case, we conclude that the account password was stolen (unless monitoring resumes) and maliciously changed unless other corroborating information (like a network outage) can be used to convince otherwise.

We also introduce a type of decoy that we refer to as a *one-time decoy.* One-time decoys function by revealing themselves as a side-effect of revealing an attacker and become immediately expired. An example of a "one-time decoy" is a *bogus* and *invalid* username and password combination that is indistinguishable from one that is real, except when it is used. An attacker is forced to test the credential in order to distinguish and validate it. Upon testing the decoy credential and learning the password is bogus, the decoy reveals itself as being fake; however, the act of testing, results in the attacker revealing himself.

We also employ beaconed decoy documents as an another type of decoy. Beacon decoys are implemented to silently contact a centralized server when a document is opened, passing to the server a unique token that was embedded within the document at creation time. The token is used to uniquely identify the decoy document and its association to the network location of the host accessing the decoy document. In addition to passing the token and IP address, extra data are collected that depend on the particular document type, and rendering environment, used while viewing the beacon document. In the case of the MS Word document beacons, the examples rely on a stealthily embedded remote image that is rendered when the document is opened. The request for the remote image is a positive indication the document has been opened. Similarly, in the case of PDF document beacons, the signaling mechanism relies on the execution of Javascript within the document.

4.2.1 PayPal Decoy Analysis

The PayPal monitor relies on the time differences recorded by the monitoring server and the PayPal service for a user's last login. The last login time displayed by the PayPal service is presented with a granularity of minutes. This imposes the constraint that we must allow for at least one minute of time between the PayPal monitor, which operates with a granularity of seconds, and the PayPal service times. In addition, we have observed that there are slight deviations between the times that can likely be attributed to time synchronization issues and latency in the PayPal login process. Hence, it is useful to add additional time to the threshold used for triggering alerts (we make it longer than the minimum resolution of one minute).

Table 4.1: PayPal decoy false negative likelihoods.

Polling Frequency	**False Negative Rate**
.5 hour	.0417
1 hour	.0208
24 hour	.0009

Another parameter that influences the detection rate is the frequency at which the monitor polls the PayPal service. Unfortunately, it is only possible to obtain the last login time from the PayPal service, so we are limited to detecting a single attack between polling intervals. Hence, the more frequent the polling, the greater the number of attacks on a single account that we can detect and the quicker an alert can be generated after an account has been exploited. However, the fact that we must allow for a minimum of one minute between the PayPal last login time and the monitor's, implies we must consider a significant tradeoff. The more frequent the polling, the greater the likelihood for false negatives due to the one minute window. In particular, the likelihood of a false negative is:

$$P_{FN} = \frac{\text{Length of window}}{\text{Polling interval}}$$

Table 4.1 provides examples of false negative likelihoods for different polling frequencies using a 75 second threshold. These rates assume only a single attack per polling interval. We rely on a 75 second threshold because it was experimentally determined that it exhibits no false positives. This was determined by monitoring the accounts for five days prior to additional experiments. For the experiments described in Chapter 6.5, we use the 1 hour polling frequency because we believe it provides an adequate balance (the false negative rate is low enough and the alerts are generated quickly enough).

4.2.2 Gmail Decoy Analysis

Table 4.2 lists the counts for login errors obtained by the decoy account monitors over a 5-month period for a selection of the Gmail decoy identities we have. This study was

Table 4.2: False alerts for Gmail Decoys over a 5-month period (based on 36,000 login attempts).

ID	Login Errors	Error Rate
1	136	.00377
2	140	.00388
3	133	.00369
4	132	.00367
5	136	.00377

conducted on decoys before the bait injection system was constructed to measure false positives associated with the monitoring infrastructure alone. For these measurements, we count login errors as false positives because sometimes we cannot discern whether they are due to account exploit in which the password has been changed, or if they were caused by something benign, like a network failure. Most of the time the login errors occur for all of the decoys simultaneously and we can be reasonably assured that the problem is due to an infrastructure hiccup. However, on rare occasions we get errors for accounts on an individual basis (hence, the different numbers). When alerts are generated individually, we cannot immediately make the determination as to whether it is a true or false positive. Resolving the true source of these errors requires waiting to see if account access resumes, which typically happens within minutes. If it does not, one can be reasonably certain that the account has been compromised.

4.2.3 Beacon Implementation Tests

To test the robustness of the beacon implementations we tested them with the most common configurations of operating systems and document viewers. To this end, we contacted a random group of users across the Internet and sent them each two types of beacon documents along with a request that they open them as part of a benign experiment. The results of tests conducted on PDF and Word beacons are presented in Table 4.3 and 4.4 below. These results are a representative sample of real users across multiple hosts accessing the

Table 4.3: PDF Beacon Test Results

OS	Application	#Tests	#Pings
Windows XP	Adobe	6	6
Windows Vista	Adobe	4	4
Mac OS	Preview	1	0
Mac OS	Adobe	1	1
Ubuntu	Evince	1	0

Table 4.4: Word Beacon Test Results

OS	Application	#Tests	#Pings
Windows XP	Word	5	4
Windows XP	GoogleDocs	1	0
Windows Vista	Adobe	4	4
Mac OS	Word	2	2
Linux	OpenOffice	1	0

beacon documents. For the most part the beacon technology works well on the windows platform while not as well on Mac and Linux operating systems. The reason is that the default PDF reader is not Adobe's and does not execute Javascript embedded within the documents. Similarly, Word document beacons do not work when applications other than Microsoft Word (*e.g.,* OpenOffice or Google Docs) are used to open them. We are currently researching ways to address these limitations and will focus on them in future work.

4.3 Perfectly Believable Decoys

As described in Chapter 3, a good decoy should make it difficult for an adversary to discern whether they are looking at an authentic document from a legitimate source or if they are looking at a decoy. We define a "perfect decoy" to be a decoy that is completely indistinguishable from one that is not. One approach we use in creating decoys relies on a document marking scheme in which all documents contain embedded markings such that decoys are tagged with HMACs (*i.e.,* a keyed cryptographic hash function) and non-decoys are tagged with indistinguishable randomness. Here, the challenge of distinguishing decoys reduces to the problem of distinguishing between pseudorandom and random numbers, a task proven to be computationally infeasible under certain assumptions about the pseudorandom generation process. Hence, we claim these to be examples of perfect decoys and the only attacker capable of distinguishing them is one with the key, perhaps the highly privileged insider.

As a prototype perfect decoy implementation, we built a component into D^3 for adding HMAC markers into PDF documents. Markers are added automatically using the iText API, and inserted into the OCProperties section of the document. The OCProperties section was chosen because it can be modified on any PDF without impact on how the document is rendered, and without introduction of visual artifacts. The HMAC value itself is created using a vector of words extracted from the content of the PDF. The HMAC key is kept secret and managed by D^3, where it is also associated with a particular registered host. Since the system depends on all documents being tagged, another component inserts random decoy markers in non-decoy documents, making them indistinguishable from decoys without knowledge of the secret key.

4.3.1 Detecting Perfectly Believable Decoys

The second host sensor also detects malicious activity by monitoring user actions directed at HMAC-embedded decoy documents. Any action directed toward a decoy is suggestive of malicious activity. When a decoy document is accessed by any application or process, the host sensor initiates a verification function. The verification function is responsible for differentiating between decoys and normal documents by computing a decoy HMAC for the particular document in question and comparing it to the one embedded in the OCProperties section of the document. If there is a match, the document is deemed a decoy and an alert is triggered; otherwise, the document is deemed normal and no action is taken.

The host sensor performs tasks similar to antivirus programs. In evaluating the performance of the sensor, we use overhead comparisons of antivirus programs as a benchmark, since the task of comparing an HMAC code is not substantially different from testing for an embedded virus signature. Hence, accuracy performance is not relevant for this particular detector. However, there is a fundamental difference between the task of detecting malware and that of detecting decoy activity. Antivirus programs are designed to prevent the execution of and quarantine malicious software whenever any process is initiated. In decoy detection the objective is merely to trigger an alert when a decoy file is loaded into memory. Thus, the decoy detection need not serialize execution; for example, it may be executed asynchronously (and in parallel by running on multiple cores).

We have tested the decoy host sensor on a Windows XP machine. A total of 108 decoy PDF documents generated through D^3 were embedded in the local file system. Markers containing randomness in place of HMACs were embedded in another 2,000 normal PDF files on the local system. Any attempt to load a decoy file in memory was recorded by the sensor including content or metadata modification, as well as any attempt to print, zip, or unzip the file.

The sensor detects the loading of decoy files in memory with 100% accuracy by validating the HMAC value in the PDF files. However, as we discovered during our validation tests, decoy tests can be susceptible to non-negligible false positive rates. The problem encountered in our testing was created by antivirus scans of the filesystem. The file accesses of the scanning process that touched a large number of files, resulted in the generation of spurious decoy alerts. Although we are engineering a solution to this particular problem by ignoring automatic antivirus scans, our test does highlight the challenges faced by such monitoring systems. There are many applications on a system that access files indiscriminately for legitimate reasons. Care must be taken to ensure that only (illicit) human activity triggers alerts. As a future improvement to the sensor, file touches not triggered by user-initiated actions, but rather caused by routine processes, such as antivirus scanners or backup processes may be filtered. Nevertheless, this demonstrates a fundamental design challenge to architect a security system with potentially interfering competing monitors.

With regard to the resource consumption of the sensor, the components of the sensor used an average 20 KB of memory during our testing, a negligible amount. When performing tests such as the zipping or copying of 50 files, the file access time overhead averaged 1.3 sec on a series of 10 tests, using files with an average size of 33 KB. Based on these numbers, we assert that our system has minimal performance impact to the system and user experience.

4.4 Masquerade detection using Decoy Documents as Bait

We have defined the general properties that decoys should have and discussed how we may measure these properties, but here we focus on the most important property: *detectability*. Under ideal testing conditions, decoy efficacy could be shown through deployment on true operational systems either within an enterprise environment, or on personal computers, by the number of attacks they are able to detect or thwart (they have a deterrence effect). However, given reasonable time limits, the infrequency of attacks within the insider threat model makes this approach impractical within a university environment. As we mentioned we are now seeking a larger user population to study and measure decoy generation over time.

Another approach to evaluation is a user study in which users are organized and asked to evaluate decoys based on each of the key decoy properties mentioned earlier. We take human evaluation to be the gold standard of evaluation since the human mind is the ultimate target of our decoys. That is, we wish to show how well our decoys can induce deception on human test subjects. One of the challenges of conducting a traditional user study lies in the logistics of obtaining volunteers. In our methodology, we attempt to reduce this challenge by leveraging external attackers to serve as participants in our study on masquerade detection. To do so, we "invite" attackers (or more accurately, bamboozle them) into our study by attracting them with a set of vulnerable systems on the university network, which also serve as our testing platform.

4.4.1 Experimental Setup

Our test platform was setup within a honeynet [The Honeynet Project, 2010]. It consisted of several virtual machines running Linux and configured with Sebek [The Honeynet Project, 2003] to capture attacker activities including commands and file references. In order to limit potential damage from system compromise and still allow for testing, we configured the honeynet to allow all incoming connections while restricting the number of outgoing connections.

The virtual machine hosts within the honeynet were configured with accounts and home directories for three decoy usernames. To make the environment as real as possible, genuine data from personal accounts on other systems were loaded into each of the home directories. We changed name references within the data to reflect those of the appropriate decoy users. In total, our phony user accounts contained 15 or more directories and 50-100 files. The hosts were then seeded with several of D^3's decoy files using the decoy distributor utility. The decoy files were generated to have conspicuous names such as "stolen passwords", "credit card", "private data", and "Gmail AccountInfo", but were distributed within the polluted home directories of the decoy accounts, making the environment as real as possible.

To lure test subjects into the study, our initial approach was to use attackers that attempt to gain internal access via password scanning. Password scanning attacks are common on the university network, where attempts on a typical machine are in the range of thousands per day. To enable attacker access, we conducted a short study to first determine the most common usernames and passwords (excluding those for root and actual users) used in these attempts. We created accounts with several of these usernames and passwords, to quickly learn that this breed of attacker was not going to suffice for our user study; their sole purpose seemed confined to creating zombies for botnets. While this may be a valid threat to study while evaluating decoys, allowing bots to operate on the university network poses too much risk.

In our second and more aggressive approach, we narrowed our recruitment effort to web forums and IRC channels with the expectation and hope that we would get fewer attacks involving botnets. In this approach, we selected several high volume forums to solicit volunteers and posted variations of invitations with messages that included hostnames, usernames, and passwords. The idea was to provide just enough innocent-looking information from a novice to lure people into our machines without providing direct evidence that we were conducting a deception-based experiment.

While our methodology could, in theory, provide anyone with access to our test platform, by selectively choosing the location of postings and contents postings, we expected to recruit two primary classes of individuals:

- Legitimate and generally curious computer-savvy individuals. These users have no interest in extending privileges in an unauthorized way, but participate in the study out of curiosity, as there is no other incentive.

- Unscrupulous opportunistic hackers who attempt to extend their network access by whatever means afforded to them. These individuals are enticed by our posting as they see our machines as low "hanging fruit" in their targeting campaign.

In either case, we believe these individuals to be suitable candidates for our study (with one caveat mentioned later). Both classes of individuals can be used in measuring the enticement property of decoys. We measure this by examining the behavior exhibited in file access, both with respect to the particular files a user attempts to read and in the order in which the files are read. For example, if all users consistently read the same file first, we know the file must indeed be enticing.

In regards to indistinguishability of the decoys, we note that the content of these decoys contains bait information in the form of monitored credentials on real systems. Certainly, if our attackers take the time to use the decoy credentials, there is an implication that they must also be believable. More importantly though, if they use the credentials and we detect their use, we have also answered the most important question of – can we *detect* the attacker? Note that the first class of the individuals is by definition, not useful for this part of the study. That is, attempting to use credentials found on our machines is clearly an illicit activity, which they would not partake in. Unfortunately, we do not have a good way of distinguishing the two classes, so we get slightly skewed results.

4.4.2 Experimental Results

Over the span of the first week[1], our hosts netted 20 unique users, determined anecdotally by source IP address and reinforced by the unique behaviors exhibited by each. The length of user sessions ranged from minutes to hours. The users that spent hours were focused on activities that included writing code and attempting to install other software, presumably with malicious intent. The most common activities exhibited for users upon login included activities such as: determining who else was on the machine, checking and deleting the command history, and exploring the system (*i.e.,* reading the most *conspicuous* files). We

[1]Most attacks occurred within the first 8-hour period after posting.

note that in almost all cases, our decoys were among the first few files viewed by users upon initial login (as noted, viewing the history file was also popular). The most significant observations were made for 6 of the 20 unique users:

1. There were three attempts to use Gmail credentials that were contained in a decoy document, which triggered an alert on $\mathtt{D}^3$.
2. One attacker changed the password on a bogus Gmail account, which also triggered a $\mathtt{D}^3$ alert.
3. There were at least two attempts to exfiltrate decoy files (with *scp* and *sftp*; one file, named "stolenpasswords", contained credentials to the university systems.
4. There was one attempt to use the university credentials contained in the "stolen-password" file, which we were alerted to by the monitor that signaled an alert to $\mathtt{D}^3$.

We take these results as evidence that D^3 indeed has value as a defense against masqueraders. While only 5 of the 20 users sounded an alarm on $\mathtt{D}^3$, we emphasize that our methodology did include an unknown proportion of benign users. Furthermore, the focus of study was on masquerade detection; admittedly, we do not yet have a good way of evaluating our system on traitors, but this will be the focus of future work.

One flaw in our evaluation methodology that was revealed during testing was that we allowed users to make changes to the file system. We did this deliberately to increase the realism of the environment in the experiments. The problem this created was that it made decoy defense vulnerable to deletion (*e.g.,* several of our visitors executed wholesale deletion of files with "rm -rf *") . This poses a problem in our testing methodology, but not necessarily in practice. That is, the act of deleting files is in itself a detectable behavior that would alert monitors of suspicious behavior.

In this study, we omitted testing decoy documents with embedded beacons. The honeypots set up to attract remote attackers were stripped down Linux machines that had no installed applications necessary to open and render the decoy documents. In Chapter 4.2.3, we describe tests of the beacon implementation on multiple hosts.

4.5 Design and Generation of Decoys Summary

In summary, although the use of bait information and similar trap-based defenses is well known, most of those efforts have focused either on artifacts that are logically separate from the operational systems (*e.g.,* honeypots [Spitzner, 2003a]) or on low-level snippets of information created manually (*e.g.,* fake database records [Spitzner, 2003b]). The $\mathtt{D}^3$ system is a scalable and automated trap-based defensive system that requires attackers to expend considerable effort to identify realistic useful information from purposely planted bogus information intended to deceive. Naturally, the probability of exposing a malicious insider with trap-based defense tactics increases with the amount of decoy information that is generated and disseminated. $\mathtt{D}^3$ offers the novel service of automatically creating and managing decoy documents, enabling the throttling of bait based on the desired protection level or cost (*e.g., interference*) one is willing to pay.

Chapter 5

Decoy Networking

In general, there is little that can be done to detect passive eavesdropping on networks. Some techniques that have been applied to wired networks for detecting snoopers—although unreliably—are based on DNS behavior or network and machine latency [AntiSniff, 2009]. The problem is only exacerbated with WiFi due to the range of signals and the absence of physical access barriers. To demonstrate the efficacy of network decoys, this chapter takes a particular focus to use of decoys on WIFI networks, although the approach is broadly applicable and can be applied to wired networks.

The ubiquity of wireless networking exposes information to threats that are difficult to detect and defend against. Even with the latest advances aimed at protecting wireless networks, compromises still occur that allow sensitive information to be recorded and absconded. Secure protocols such as WiFi Protected Access 2 (WPA2) can help in preventing network compromise, but in many cases they are not used for reasons that may include cost, complexity, or overhead. In fact, the 2008 RSA Wireless Security Survey reported that only 49% of corporate access points in New York City (NYC) and 48% in London used advanced security [Cracknell *et al.*, 2008]. To make things worse, only 24% and 19% of the NYC and London total APs respectively, used a WPA2 variant.

The nature of radio communication makes the problem far more challenging; generally speaking, these methods are not applicable. We address the problem of eavesdropping and offer a proactive defense that makes it difficult for snoopers to avoid detection by targeting the *semantic information* sought by the attackers rather than *network-level observables* that has been the focus of previous efforts. We broadly target two types of attackers:

- Insiders, who legitimately have access to a network, but attempt to use it for attaining illegitimate goals. In the case of shared-key encrypted wireless networks, (*e.g.,* WEP, and some instances of WPA) malicious insiders may eavesdrop with little difficulty since they are already within the protective security perimeter. In other cases, there may simply be no data encryption (*e.g.,* as in many enterprise networks), where the only barriers to separate the outside are firewalls or some form of physical security, or with wireless hotspots (whether commercial or free).

- Those that successfully infiltrate the network through attacks at the protocol level [Beck and Tews, 2009; Bittau *et al.*, 2006], password guessing, router hijacking [Akritidis *et al.*, 2007; Tsow *et al.*, 2006], or some vulnerability in WiFi security. As a concrete example, consider the case of the massive credit card heist that occurred at TJX [Pereira, 2007] in which attackers exploited the vulnerable WEP protocol to gain internal network access. Once inside, attackers eavesdropped undetected, acquired additional credentials, and eventually stole over 45 million credit cards [McGlasson, 2007].

Our intuition is to confuse, deceive, and detect attackers by leveraging uncertainty. We achieve this by introducing decoy traffic with enticing information that will, eventually, cause the eavesdropper to undertake some *observable action*, such as access a decoy account using sniffed credentials. Our methodology for building a trap-based network that is designed to maximize the realism of decoy traffic. We propose and demonstrate the utility of a novel architecture based on a "record, modify, replay" paradigm to automatically generate large quantities of decoy traffic that are injected into the network. The system continuously regenerates decoys to prevent an adversary from learning how to recognize bait over time. Our contribution lies in the *automation* of decoy generation and injection, which allows the use of decoys in large volumes.

Our prototype implementation demonstrates the feasibility of this approach on WiFi networks. However, the methodology is broadly applicable and can be adopted to conventional wired networks. Our proactive defense, which offers a controllable level of protection, is based on the amount of "bait" traffic one is willing to inject. This amount can be throttled based on a tolerated level of interference, as indicated in Chapter 5.5.

Demonstrating decoy efficacy and accuracy against snoopers requires an indeterminate amount of time; in Chapter 5.2 we simulate attacks to show that the monitoring works well and would capture snoopers if they misuse the stolen credentials. This assurance depends on whether the snooping adversary captures the decoys that are believed to be real. Hence it is the *believability* of decoys that is the most important property evaluated in this work. We posit that the believability of decoy network flows can be measured by their indistinguishability from what is real and we demonstrate decoy flow believability by conducting a user study that is analogous to the Turing Test [Turing, 1950]; results are presented in Chapter 5.3 that testify to decoy realism.

5.1 Platform Implementation

Synthetic network traffic is typically generated to support simulations, or emulations, that require traffic to be structurally and syntactically correct with respect to protocols. In contrast, decoy traffic is designed with a fundamentally different goal—to induce deception on the human viewer. In Chapter 3, we formally defined a core set of properties, including *believability, non interference, detectability, variability,* and *enticement* to guide the creation of decoys. We used some of these properties to aid the design of our platform and its evaluation. We posit that achieving the deception goal requires traffic to be believable, a quality ultimately measured by humans, in addition to the more general requirements of syntactical and structural correctness. Our system addresses these objectives with an architecture comprised of several hardware and software components that have been designed to support the "record, modify, replay" paradigm for producing honeyflows. This model produces believable decoys by leveraging human- generated content from recorded flows, as opposed to relying solely on machine intelligence. The resulting honeyflows contain both *cover* and *carry* traffic; carry traffic contains the decoys, whereas cover traffic includes everything else to support the believability of carry traffic. The architectural components, shown in Figure 5.1, include a decoy traffic generator, a distribution platform built on commodity hardware, and a set of broadcasters for performing the injection of the various types

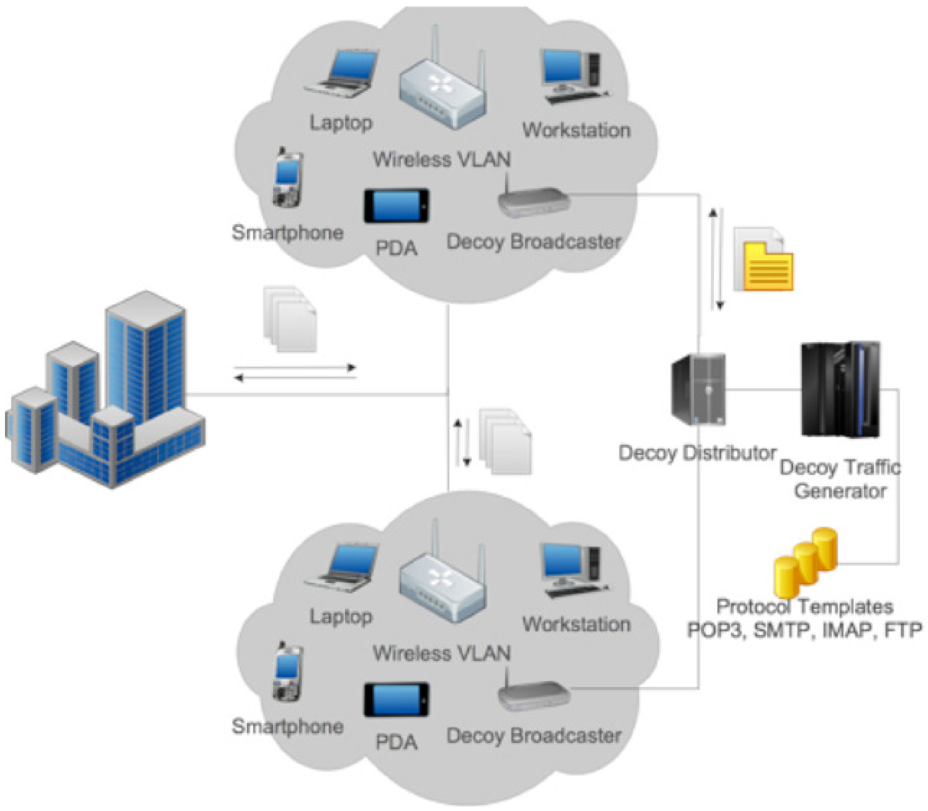


Figure 5.1: Injection Platform.

of decoys. The implementation details are discussed in the following subsections.

5.1.1 Automated Decoy Traffic Generator

The decoy traffic generator uses the software API that we developed to produce honeyflows through a multi-step process, as shown in Figure 5.2. The automated process begins by loading recorded network data, which might either be a template containing anonymous trace data, or ideally, a complete network trace containing authentic traffic—we have specifically designed the API to handle both types of input due to the ethical and legal issues concerning the recording of network traffic (see Section 5.6). Within the university environment, we use the template approach in which sets of protocol-specific templates are manually created and passed to the API as input. The templates contain traffic of various network protocols including TCP session samples for protocols used by our decoys. The obvious drawback of

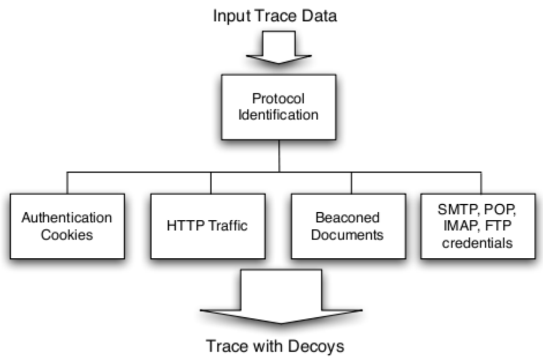
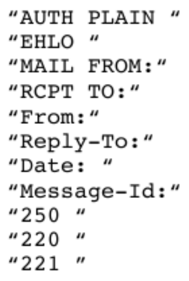


Figure 5.2: Honeyflow creation process.



```
"AUTH PLAIN "
"EHLO "
"MAIL FROM:"
"RCPT TO:"
"From:"
"Reply-To:"
"Date: "
"Message-Id:"
"250 "
"220 "
"221 "
```

Figure 5.3: SMTP Identifiers.

Table 5.1: Rules used to match protocols.

Protocol	**No. of Identifiers**	**% Required**
FTP	14	65%
GMail	7	70%
IMAP	10	40%
POP3	5	80%
SMTP	10	50%

templates is that the diversity and volume of the content is limited, which may subtract from the realism of the overall generated traffic. However, it is important to note that there are other environments in which it is legal and common to record traffic (*e.g.,* enterprise environments). In these environments, it would be advantageous to use live network traces as a basis for decoy traffic within which decoys can be added.

Once the API obtains an input trace, a new trace is automatically created with decoy information by following these steps:

1. Each input trace consists of multiple protocols and TCP sessions. We demultiplex each session/protocol into individual trace files for simpler processing.

2. Configuration information (*e.g.,* decoy information, IP/MAC addresses of emulated networks) is read from a user specified configuration file.

3. Each demultiplexed trace file is passed through protocol-specific traffic *identifier* functions for the protocols we support (currently Gmail, SMTP, POP, IMAP, FTP, HTTP) to find the best match. The best match is found using predefined rules that examine network trace data to determine protocols based on the content of application-layer headers and protocol status messages. The approach relies on the presence of identifiers specific for a given protocol. The API can handle identifiers which are both simple literal strings or complex regular expressions. For example, Figure 5.3 shows the identifiers we use for the SMTP protocol. To accommodate varying application-layer protocol implementations, we rely on a percentage of identifiers being present for each protocol as shown in Table 5.1, rather than all of them. We determined the percentage by manually observing real traffic from various implementations on a per-protocol basis. Specifically, for the SMTP identifiers we rely on 80% of the identifiers being present. If protocol determination does not succeed, the trace is marked as *unknown* and the API proceeds to step 6.

4. Identified traces are passed through a protocol *modifier* function to insert decoy information. Our API supports rules for adding bait to protocol headers, such as Gmail cookies and SMTP passwords, and protocol payloads (*i.e.,* email body, web page content). Additionally, our implementation provides rules for creating several types of decoys including: Gmail authentication cookies, URLs, passwords for unencrypted protocols (*e.g.,* SMTP, POP, IMAP), and beaconed documents as email attachments (see Section 4.2). Moreover, the API can also be used to introduce bait HTTP flows that contain monitored URLs or handle protocol complexities such as:

    (a) *Multi-packet editing.* If multi-packet editing is required (*e.g.,* insert a decoy file as attachment into a POP3 trace), we buffer the data in memory. When a boundary is found (*i.e.,* a protocol status code indicating an end of file), the modifier function stops buffering and inserts the decoy object. This data is then written back to the output trace file as multiple packets.

    (b) *Protocol encoding.* The API formats the decoy information appropriately for the given protocol (*e.g.,* Base64 for POP3 attachments).

5. Rules are used for the replacement of MACs and IPs to those from a predefined set to suit the environment. For example, we select bogus IP addresses that are consistent with those used inside a wireless cell, so as to avoid breaking the semantics of the corresponding network topology. Similarly, the IP/MAC pairing is carefully selected to be persistent throughout multiple bogus sessions.

6. Additional variability and randomness are introduced to the honeyflows using these techniques:

   (a) For identified TCP server protocols the client port is randomly generated. However, since different clients have different *ephemeral port* ranges (*e.g.,* FreeBSD follows the IANA dynamic/private port range, Linux uses the range 32768 to 61000, Solaris uses 32768 through 65535 and so forth), we generate the client port either based on the bogus host that we simulate (in case the client OS is important), or by following the IANA dynamic/private port range (when the client OS is irrelevant).

   (b) TCP sequence numbers are modified to be consistent with the size of the newly generated packets, whereas heuristics are used to modify aspects of content like names, addresses, and dates so that they match those of the decoy identities.

   (c) Parameterization of temporal features (*e.g.,* total flow time, inter-packet time) that can be extracted from Netflow or packet trace data [Sommers and Barford, 2004] enable the creation of honeyflows that are statistically similar to normal traffic.

7. OS fingerprint models of *p0f* [Zalewski, 2006] are used to generate honeyflows that resemble the host OS. For example, to generate traffic that appears to emanate from a Linux host, we avoid generating traffic that appears to have come from the MS Outlook email client.

8. The demultiplexed traces are combined into a single trace, which is then broadcasted to the environment.

5.1.2 Statistically Similar Temporal Features

In cases where real traffic is available, it can be used to build generative models of temporal features. Modeling real traffic as opposed to Netflow data is beneficial because it allows for more precise timing models. One approach we have used relies on analyzing TCP sessions to gather per-session metrics including the total session length, average time between packets, minimum time between packets, and maximum time between packets. We then use this information to bound timing models when synthesizing session temporal features. For example, for a given session, we record the packet times, calculate mean and standard deviations between packets. We then sample the session to get a distribution of times relative to the standard deviation. This distribution is used for binning the data. The size of the bin is parameterized to control its granularity. In Chapter 5.4, we provide an analysis on how the granularity of the bins influence performance. The distribution of times relative to the standard deviation and how they are binned become the timing model for a particular session. In practice, we have found our binning strategy to be sufficient. Other approaches that rely on the creation of equiprobable bins [Gianvecchio and Wang, 2007] may also be used [1].

We generate new random packet times using this model as a control for the minimum and maximum times. This approach allows for an unbounded amount of variability in the temporal features for synthesized flows. The variability can be controlled by parameters that determine how closely the generated timing features conform to the session's timing distribution. This technique can also be used to control the performance. We provide an evaluation statistical similarities of the synthesized flows to real flows using classification techniques in Chapter 5.4.

5.1.3 Decoy Broadcaster

Decoy Broadcaster is the architectural component of our system that is responsible for spreading the bait content inside a network segment. It is comprised of both hardware and software entities. Figure 5.1 illustrates a decoy broadcaster inside the context of our

[1] This is a substantial thesis topic on its own being pursued by Yingbo Song [Song, 2011]

campus-wide wireless network. The underlying hardware consists of a low-cost, general-purpose, wireless router with the ability to inject traffic. The device is *strategically* placed in the vicinity of a legitimate access point (AP) so as to maximize the coverage of the replayed traffic. Ideally, the bait content should be sniffable by all wireless clients inside the same cell. An additional requirement of the decoy broadcaster is the support of *monitor mode*[2] operation by its wireless NIC. Our preliminary experimentation revealed that monitor mode is the only one that provides the flexibility to inject packets that meet the needs of our architecture. In all other modes, injection either failed or it was limited. For example, in *managed mode* we found that it was not possible to modify frame fields such as `FromDS`, `ToDS`, or the MAC address, which may be important for creating realistic traffic. Furthermore, it was not possible to inject anything other than data frames (*e.g.,* ACKs, RTS/CTS). The problem is that such limitations may create artifacts in the honeyflows that allow sophisticated adversaries to identify and avoid the bogus traffic.

For our prototype implementation we used `Accton MR3201A` [Mini router, 2009] a mesh router based on `Atheros AR2315` chipset, with 32 MB DRAM and 8 MB flash. The device comes pre-flashed with a modified version of *OpenWRT* [OpenWRT, 2009] —a Linux-based firmware for embedded devices. However, in order to fully utilize the capabilities of the device, we installed a custom OpenWRT image. Our configuration aims at free space maximization and negligible CPU usage due to leftover services. The root filesystem of the device is about 1.8 MB, leaving us with 5.2 MB of free space in the flash disk. Because of the relatively large portion of free RAM (*i.e.,* almost 24 MB of free memory) we can use a fraction of it as a ramdrive in order to increase the decoy storage capacity. Therefore, an additional 15 MB were put aside, using the *tmpfs* filesystem, giving us in total almost 20 MB of space for decoys. Accton's wireless NIC uses the *MadWifi* [the madwifi project, 2009] driver that supports a wide set of features such as:

- Different operation modes: Station, Master, Ad-Hoc and so on, including the monitor mode.

[2]Monitor mode (`RFMON`), is one of the six operational modes of an IEEE 802.11 compatible card. The remaining five are: *Master* (AP), *Managed* (client associated to an AP, also known as *Station*), *Ad-hoc*, *Mesh*, and *Repeater*.

- Multiple *Base-Station IDs* (BSSIDs) via different virtual interfaces on top of the same NIC. That is, the *Virtual Access Points* (VAPs) feature, which supports virtual interfaces that can even be in different modes.

- 4-address header support, dynamic frequency selection, background scanning.

The most important features are the VAPs and monitor mode support. As far as monitor mode is concerned, we tweaked the MadWifi driver in order to suppress 802.11 ACK frames (only in VAPs being in `RFMON` mode), since we have our own ACK frames recorded as part of the decoy traffic, and ignore ACK timeouts in injected frames[3]. To inject the honeyflows we ported *Tcpreplay* [Tcpreplay, 2009], a suite for replaying previously captured traffic for network testing purposes. The typical injection workflow is specified as follows:

1. Create a new VAP in the Decoy Broadcaster and set it in Monitor mode.

2. Upload bait traffic into the Decoy Broadcaster.[4]

3. Use Tcpreplay to distribute the decoy traffic inside the wireless cell.

It is critical that the decoy repository on broadcasters be refreshed regularly. In some cases, this is required to support the broadcasting of valid bait. For example, we use authentication cookies (see Section 4.2) as one type of decoy. Since these are valid for only a finite amount of time, they need to be routinely regenerated. Most importantly, however, is that decoy traffic must be routinely updated so that it remains believable to attackers. If the same traffic was continuously replayed, it would be easily distinguishable based on the retransmissions of protocol header parts (*e.g.,* TCP sequence numbers, IP TTL, TCP/UDP source port numbers, IP ID), which should be unique for every session.

We considered various approaches for resolving this issue. At one extreme, we may have a fully centralized solution, which involves preparing new honeyflows in the `Decoy Traffic`

[3]We inject whole sessions: traffic from all communicating parties including ACK frames and retransmissions.

[4]This can be done either by having another VAP in managed mode and establish a communication channel between the Decoy Broadcaster and the Decoy Distributor, or by directly utilizing the Ethernet interface of the mini-router.

`Generator` (see Figure 5.1) and disseminating them to the proper `Decoy Broadcasters` (*i.e.,* certain MAC/IP addresses for certain cells to avoid having spatial inconsistencies). At the other extreme, a decentralized approach can be employed for "on-the-fly" honeyflow creation within the decoy broadcasters. Each option offers different tradeoffs. For example, a benefit of the centralized approach is that it requires no intelligence at the decoy broadcasters; they are only dummy bait traffic repeaters. Drawbacks of the centralized approach include the imposition of additional overhead on the decoy traffic generator, scalability limitations, and the lack of fine-grained control over injection (*i.e.,* the delay between the time that the generator decides to send a decoy for injection and the time the actual injection takes place). The decentralized approach provides more flexibility since it leverages continuous bait generation with agile decoy broadcasters. Nonetheless, the packet processing required to create honeyflows, demands devices with considerable capabilities. This tradeoff, though identified, has not been evaluated in this study and it will be the focus of future research.

5.2 Detecting Snoopers

Our system injects a variety of different types of "bait" traffic into Wi-Fi channels in order to entice, deceive, and alert us to the presence of malicious eavesdroppers. Enticing and detecting attackers largely depends on attackers' goals, whether they pilfer sensitive information to sell on the black market, or perhaps, some form of espionage. The capacity to expose otherwise elusive attackers on wireless networks is one of the primary contributions of this work. Unfortunately, the ability to evaluate this contribution is constrained by the infrequency of attacks in our university environment. Waiting for such an attack requires an indeterminate amount of time and may not be practical. Therefore, in order to assess the effectiveness of our system in realistic environments, we performed two studies. The first experiment was performed at the `Defcon` '09 hacking conference in Las Vegas, to test whether the decoy injection framework would succeed in transmitting decoy credentials. Additionally, we developed a program to simulate threats known to exist in the wild (*e.g.,* massive cookie harvesting) and tested it in our campus network. The results from both studies are presented and discussed below.

5.2.1 Defcon Experiment

Defcon's yearly meeting includes the infamous *wall of sheep* [Wall of Sheep, 2009], which is an interactive demonstration of what can happen when network users do not use the protection of encryption. Defcon staff eavesdrop the network traffic for unencrypted credentials, which they later post on a publicly accessible wall as a good-natured reminder of what a malicious person could do.

Throughout the conference we repeatedly injected decoy traffic and waited for some decoy credentials to appear on the wall. One of our decoy credentials did indeed appear on the wall of sheep 5.4, which is an indication of a successful decoy injection. Surprisingly, a Gmail decoy alert was triggered after someone logged into one of our Gmail accounts from an IP address in New Jersey, shortly after the account was used in Las Vegas. In that case, we believe the decoy was the victim of a cookie hijacking attack, but we do not have strong evidence for this. The Defcon staff post the collected information (although passwords are only partially shown), but they do not use any credential. However, this does not exclude other participants that were passively monitoring the wireless channel during the conference from being malicious.

This experiment provides evidence that our system may detect when a snooper is using automated tools for harvesting and exploiting credentials in the wild. Though we have performed a detailed evaluation regarding the quality of our decoy traffic in believability terms (see Section 5.3), we expect that a typical adversary will probably utilize automated tools that massively hunt credentials or other interesting information (*e.g.,* identity data, credit card numbers). Unfortunately, the Wi-Fi bait traffic we broadcasted was not adequately sniffed. We later learned that Defcon staff were monitoring the switch mirroring ports as opposed to Wi-Fi radio channels. However, this is orthogonal to our experiment.

5.2.2 Massive Cookie Harvesting

As a practical and relevant alternative to evaluate real attacks, we have developed a program to simulate threats known to exist in the wild [Pereira, 2007]. In particular, we model

Figure 5.4: Defcon '09 Wall of Sheep.

attackers that attempt to harvest login cookies in mass (also known as *SideJacking* [Graham, 2007]), which are broadcasted in the network unencrypted and can be exploited by an attacker to provide full access to users' personal accounts and information. Any web site that allows cookies in clear text(*e.g.,* Yahoo Mail, MSN Hotmail) is potentially vulnerable to this type of attack; without loss of generality we focus on Gmail. Our selection of Gmail is partially due to the size of the Gmail user base (113 million registered users [Morse, 2009]), but also because Gmail provides the means to allow us monitoring access.

Our model attack program is called `Gsnoop` and it works by sniffing a specified network connection to identify and record Gmail login cookies. Once a cookie is obtained, Gsnoop uses the cookie to log into the account and read the author, subject, and date—the selection of which is arbitrary and for demonstration purposes only—of the first 3 emails as proof of account compromise. While this is fairly benign, the code could be easily extended to do more malicious acts such as searching the inbox for other valuable credentials (as often found), sending spam, deleting all contacts, and so forth.

To validate our decoys, we run Gsnoop on the university network, but in a restricted mode so that it would not login to accounts indiscriminately (that would be unethical). The host running Gsnoop was placed in monitor mode and physically located within twenty feet of one of the Decoy Broadcasters to ensure sufficient decoy exposure. To conduct the experiment, ten unique honeyflow sessions were injected by the broadcaster. The honeyflows were generated to contain the authentication cookies for three different decoy identities.

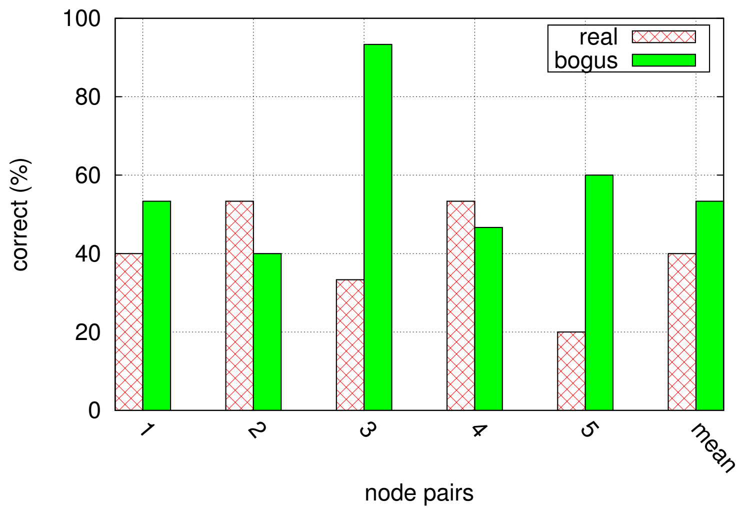


Figure 5.5: DTT 1 Results: Real vs. Decoy.

Results from the attack simulation included exactly one alert for each of the decoys. The alerts were triggered within ten minutes of the Gsnoop automated attack, validating that the system worked as intended. The latency between the exploit time and detection time was an artifact of how frequently the monitoring system was configured to poll for account activity on the Gmail decoys. In addition to ensuring the validity of the bait cookies, this also provided support for the structural correctness of the fabricated frames, as well as the operational success of the decoy monitors. Although there were no false positives recorded in this testing scenario, we have found that false positives can pose a problem for decoys in general. We discuss this in Section 4.2

5.3 Believability of Bogus Traffic: A Decoy Turing Test

Alan Turing proposed [Turing, 1950] a method to demonstrate "artificial intelligence" through the failure of human judges to distinguish between human and machine conversational simulators. The *imitation game* as it was named, was conducted over a text-only communication channel whereby the judge engaged in conversation with both a human and machine. The machine was said to have passed the test if the judge could not reliably distinguish between it and the human. Following the notion of the original imitation game, we designed a *Decoy Turing Test* (DTT) that relies upon human judges to distinguish between

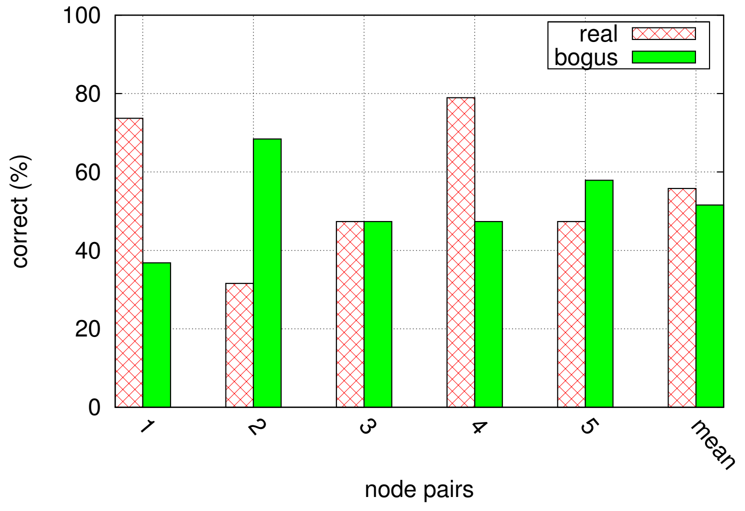


Figure 5.6: DTT 2 Results: Real vs. Decoy.

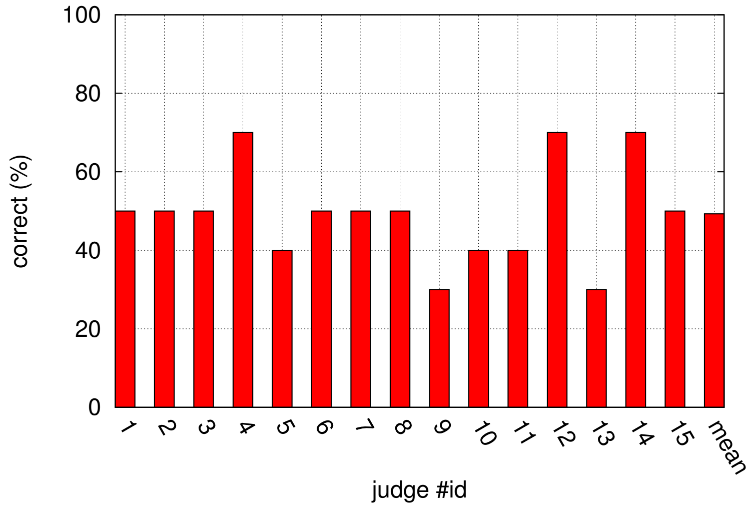


Figure 5.7: DTT 1 Results: Users' Correctness.

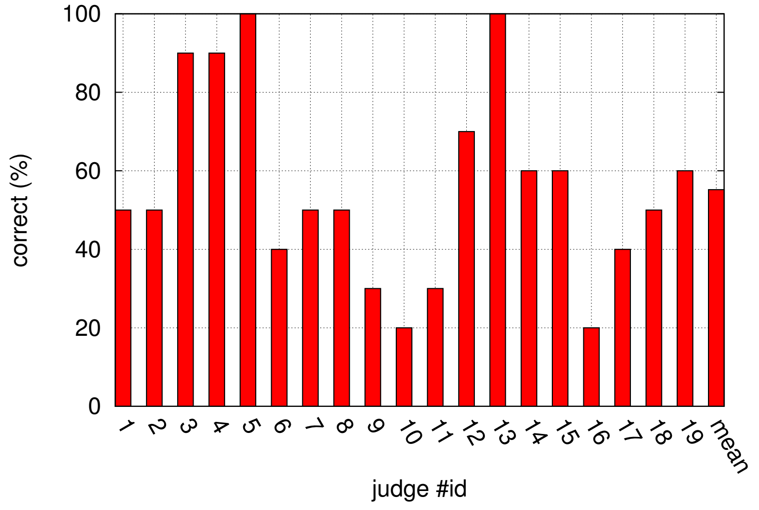


Figure 5.8: DTT 2 Results: Users' Correctness.

authentic and machine generated decoy network traffic. Their inability to reliably discern one traffic source from the other attests to decoy believability.

We conducted two separate experiments that differed in the pool of judges and the amount of time they were asked to spend with the experiment. In our first experiment, human judges were solicited and selected based on their prior knowledge of networking protocols and experience in examining network traces [5]. The first experiment relied upon a pool of 15 judges consisting of PhD's and graduate students in the network security field, a staff member from the department computing research facility, and a security professional from an antivirus company. They were asked to spend at least 15 minutes – a minimal amount of time on the task. For the second experiment, we relied on 19 students from a network security class to participate as part of a homework assignment and we motivated them with grades. We did not specify the amount of time they should spend, but it was clear from the length of the responses that the second group put in considerable more time. In both cases, the task for the judges required the analysis of network trace data, created specifically for this experiment using the injection API. The test trace was created through the process outlined in Chapter 5.1, but with slight modifications to enable a structured study. We constructed our test data set including traffic from only 10 hosts, assuming the

[5]The IRB for this study is included in Chapter 9.

judges would dedicate limited time, and tolerate only a small volume of data.

To create the test data, we began by recording traffic from 5 hosts on a private network. The private network was used so that we would not accidentally record other users' traffic and skirt legal or ethical boundaries. Due to the fact that the network data were ultimately going to be distributed to the judges (and perhaps elsewhere), we had users' on the private network assume "test" identities that were created for local email, FTP servers, and Gmail accounts. The users were asked to engage one another in email conversations, surf the web as they would normally, and perform FTP transactions. We recorded approximately 15 minutes of traffic in which there were samples of HTTP, Gmail account activity, POP/IMAP, SMTP, and FTP traffic.

This network trace was then scrubbed of all non-TCP traffic to reduce the volume of data we would be asking our judges to examine. The resulting trace was passed to the honeyflow creation process as shown in Figure 5.2 to produce honeyflows for each of the 5 hosts. These honeyflows were loaded with the decoy credentials, given their own MACs and valid university IP addresses, and finally interwoven with the authentic flows to create a file containing all of the network trace data. The choice was made to give honeyflows distinct IP addresses to simplify the task for the judges. For each of the resulting 10 IP addresses, the judges were asked to make the binary decision: *real* or *decoy*. We requested them to conduct their analysis and permitted them to use any tool to aid in making the decision.

5.3.1 Results and Discussion

Figures 5.5 and 5.6 summarize the results for each of 10 hosts in the two rounds of experiments. The hosts are arranged in pairs in which the right bars correspond to decoys and the left bars correspond to the authentic traffic on which decoys are based. The height of the bars reflect the number of judges that correctly decided whether a given host was real or decoy. Although there are some inconstancies in the results between the two graphs with Figure 5.5 suggesting that judges were able to discern decoys more regularly than authentic hosts (as shown by the height of the bars on the right) and Figure 5.6 suggesting the opposite, the mean was approximately the same for both experiments. In addition, it is important to take into consideration the judges' overall correctness. Figure 5.7 and Figure 5.8 show the overall correctness for each of the fifteen judges in the first experiment and nineteen judges in the second experiment. Overall, the first experiment judges were 49% correct, on average, suggesting that we have achieved the goal of indistinguishable decoys. Interviewing the judges we concluded that the bias for decoys in Figure 5.5 stemmed from their *tendency* to guess "decoy" more frequently than not. In other words, decoy was the default decision when a judge was uncertain. Since this tendency led them to tag real traffic as decoys, one can surmise that the use of decoys in a network has an additional deterrent value against knowledgeable adversaries. The judges in the second experiment performed slightly better with 55% correctness. Despite their marginal success, the judges still failed to distinguish the decoys almost half of the time, leaving plenty of opportunity to trick them. Both results are statistically equivalent to 50% for significance $p > .05$.

One of the reasons for the discrepancy between the two results is likely because is the judges in the second experiment spent far more time in their analysis, which was clear from the length and quality of their explanations. As clear from Figure 5.8, four of the second experiment judges successfully identified deficiencies in the decoys that allowed them to distinguish decoys. The first issue had to do with obscure manufacturer names (*e.g.,* Shandong New Beiyang Information Technology Co.) for some MACs used in decoy traffic, which enabled a correct determination to be made about whether traffic was decoy or not. The problem was that we randomly generated MAC addresses rather than using only MACs from common vendors. The second issue uncovered by the judges was an invariant in the TCP header for the decoy traffic. Two of the judges discovered that TCP window size was not being correctly set. The first issue had to do with poor design choice while the second was simply a bug. We have since fixed both of these problems, but the issues do speak to the challenge of getting bogus traffic to look real, especially in the eyes of knowledgeable judges willing to invest time to conduct a deep analysis.

Another challenge was dealing with judges that have insider knowledge. Our study did include judges with knowledge of the department network topology and one who works for the computing facility, but this knowledge did not help in distinguishing decoys. We should also point out that between the two studies, there were actually 4 users that had 7 out of 10 correct, but their justification did not turn out to be a true means for distinguishing decoys.

For example, one of the judges said that the IPs of the destination hosts in the traffic did resolve through reverse DNS; however, these same IPs were found in the real traffic. Hence, this judge was simply lucky since this is not a true flaw to identify the decoys. Regardless, the fact that some, but not all, decoys are correctly identified is promising, since we only need a single bait to be taken for detection to occur.

5.4 Statistical and Information Theoretic Analysis

In this section, we present results of a statistical analysis of the generated network traces. The goal was to see if a machine learning algorithm might be able to accurately classify traffic as real or machine generated. An attacker may resort to such an algorithm if direct observation alone cannot be used to distinguish real from generated. In particular, we focus on the timing information of the generated traffic as an example of what attackers may attempt to analyze.

5.4.1 Evaluation Data

The experiments were performed by first recording traffic from seven IP addresses on a private network. We arranged it so that each of the seven IP addresses would primarily communicate with a single type of protocol. This resulted in seven samples that containing each of the various protocols including IMAP, FTP, SMTP, POP, and HTTP. The number of sessions recorded for each protocol was variable with the HTTP sessions being the most due to the way the browsers are implemented. It is typical for a browser to spawn many sessions to gather the content of a single web page. Even though the API does not inject decoys into encrypted traffic, we included HTTPS as an additional sample because we can still model the timing information. The recorded samples were split so that half of the TCP sessions could be used for training a classifier and half could be used for testing. The sessions used for testing the classifier were also used by the API to build generative timing models for the generated session times. These models were then used by the API to generate new sessions. Once the new sessions were generated, we had a total of three network traces to represent training, testing, and generated data. We then extracted session-level features from each of traces that included total session time, average inter-arrival time for packets, the standard deviations between times, minimum time between packets, and maximum time. For purposes of feature extraction, we defined a session to be the time the first SYN packet is identified between a set of host and port pairs to the time the last packet is identified. We did not bother identifying FIN packets to accommodate sessions that may end prematurely.

5.4.2 Classification Experiments

Our initial evaluation of the generated timing information was performed using classifiers from Weka [Hall *et al.*, 2009]. Many algorithms within the tool failed to be able to classify the real traffic. Our tests indicated that these algorithms also failed to classify the generated traffic. This is desirable because if it were not the case, an attacker may use such an approach to identify the generated traffic. However, we did not feel the results warranted representation beyond this note. Instead, we selected two algorithms based on their ability to accurately classify real sessions for various protocols because many algorithms failed at this task. We selected the J48 decision tree and BayeNet classifiers. In addition to classifying real traffic correctly, we selected the decision tree algorithm because prior work by Early *et al.* [Early *et al.*, 2003] has shown them to be useful in classifying network traffic for HTTP, FTP, Telnet, SMTP, and SSH protocols.

In selecting an algorithm for classification, it is common to test the algorithm using k-fold validation on the training data. The result of which is an indication on how well one expects the algorithm to correctly classify test data (*i.e.,* that which has not been seen). We conducted a 10-fold analysis whereby 10% of the data was tested against a model created from the remaining data. The test was repeated ten times by varying the test data so that all of the data was used for testing. The average number of instances classified correctly under 10-fold analysis for the Decision Tree and BayesNet classifiers were 63% and 51% respectively. For purposes of these experiments, the results are sufficient because our goal is to show the statistical similarities between the generated and real traffic, as opposed to whether they can be classified correctly.

The results of running the Decision Tree and BayesNet classifiers on the generated timing data and real timing data are presented in Figs. 5.9 and 5.10, respectively. The Decision Tree classifier had an average accuracy of 55.6% for real and 56.9% for generated traffic. The BayesNet classifier had an average accuracy of 55.2% for real and 47.3% for generated traffic. The close results for these two classifiers suggest that this particular type of analysis would not be useful for an attacker attempting to distinguish the real from generated traffic.

5.4.3 Kolmogorov-Smirnov Tests

In our second experiment, we relied on the Kolmogorov-Smirnov test, which has been shown to be useful in detecting some types of covert timing channels in network traffic [Gianvecchio and Wang, 2007] [Peng *et al.*, 2006]. The task of detecting covert channels is similar to ours in that it requires identifying statistical differences between samples that are themselves highly variable. The Kolmogorov-Smirnov test is a non-parametric test that can be used to show that two samples come from the same distribution. An advantage of Kolmogorov-Smirnov test over others (e.g., Chi-Square) is that it does not rely on assumptions about the underlying distributions of the sample data and is intended for use when the distributions are continuous.

Similar to tests conducted in [Gianvecchio and Wang, 2007], the aim of our experiments were to see if we could detect statistical differences in the inter-arrival times between real and generated traffic. We used the training and testing data described in 5.4.1 to obtain threshold such that the false positive rate was .05. This was done experimentally by extracting the inter-arrival times for sets of 50 packets in both the training and test data and scoring them with the Kolmogorov-Smirnov test. The lower the resulting score, the closer the test sample is to the training set. We determined the score such that the false positive rate was .05 and used this as the threshold for subsequent tests with generated timing data. Scores for the generated timing data over this threshold were deemed true positives. That is, if an attacker were using this test as a tool to distinguish real timing data from generated timing data, the timing data scoring over this threshold would successfully be identified as generated. Our choice to use the .05 false positive rate was to allow for enough resolution in the data to show how varying bin lengths influence the true positive rate. We conducted three tests with variable bin lengths for the generated traffic ranging from .001 to .1 sec-

Table 5.2: Kolmogorov-Smirnov test results for different binning thresholds.

Threshold	**Bin length (seconds)**	**FP Rate**	**TP Rate**
.41	.001	.05	.06
.41	.01	.05	.11
.41	.1	.05	.27

onds. The results of the tests are presented in Table 5.2. The results indicate that the true positive rate can be controlled by varying the bin lengths of the generated traffic. With a bin length of .001, we obtained a .06 true positive rate meaning that 94% of the generated traffic would go undetected using this test. Even with a specificity of .1 seconds, 73% of the generated traffic would go undetected, making the attackers task difficult. Other tests that we conducted with a false positive rate of .01 resulted in true positive rate under 1% for all of the bin lengths, suggesting the challenge might even be more difficult.

5.4.4 Entropy Tests

Entropy is a common measure for the amount of disorder in a data set[Lee and Xiang, 2001]. It has also been used in the detection of covert timing channels [Peng *et al.*, 2006]. It is commonly used to measure the number of bits required to encode the classification of a data item. For a data set X containing possible classifications $x_1, x_2, ..., x_n$, the entropy of X is defined as:

$$H(X) = \sum_{i=0}^{n} \Pr(x) \log \Pr(x)$$

For the timing data, we calculate the bin for a particular time as the approximate distance from the mean in standard deviations. The classification of a time maps to its bin. Figure 5.11, we present a comparison of entropy values for the actual and generated data for each of the samples. The results suggest that there is no loss of information in our generation process that would be useful by an adversary that is attempting distinguish real from generated sessions. In fact, the results show that there is a slight gain of information in the generated traffic. This is a consequence of the binning strategy used. These particular results were generated using a bin length of .01. Decreasing the bin size has the effect of

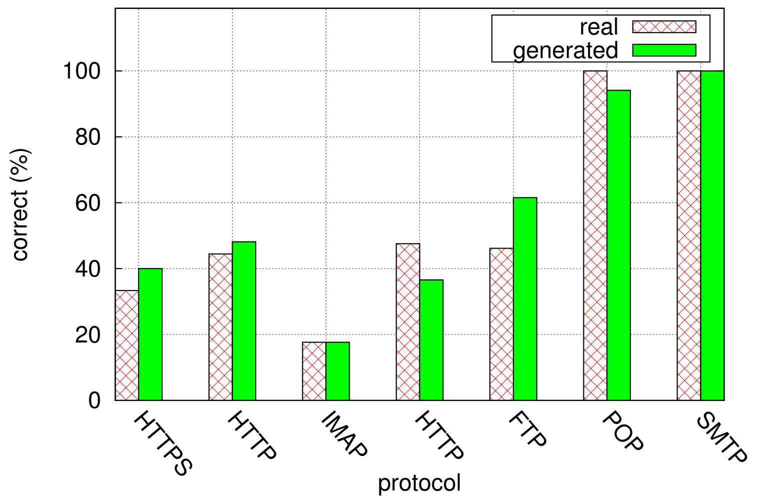


Figure 5.9: Decision Tree classification.

tightening (e.g., using .001) the bounds used for generating timing data, thereby decreasing the amount of information.

5.5 Interference Measurements

Introducing decoy traffic into an operational network has the potential to *interfere* with normal network activities in multiple ways. Our primary concern is that decoys may pollute authentic data so that their legitimate usage becomes hindered by corruption or as a result of confusion by legitimate users (*i.e.,* they cannot differentiate real from fake). We address this concern and minimize the risk of corrupting normal data by injecting frames that are not addressed to legitimate hosts or users. Hence, only a passive eavesdropper will observe them. Of secondary concern is the performance impact due to the increased burden on network resources. Flooding Wi-Fi channels with bogus data comes at a performance cost that can be measured from the side-effects to available bandwidth, packet error rate, and channel contention. We posit that there is a tradeoff between the amount of deceptive data we may inject to maximize the protection level, and the *perceived* performance as measured by the impact on user applications. In this section, we present experimental results that show our approach causes only minimal interference to ordinary users.

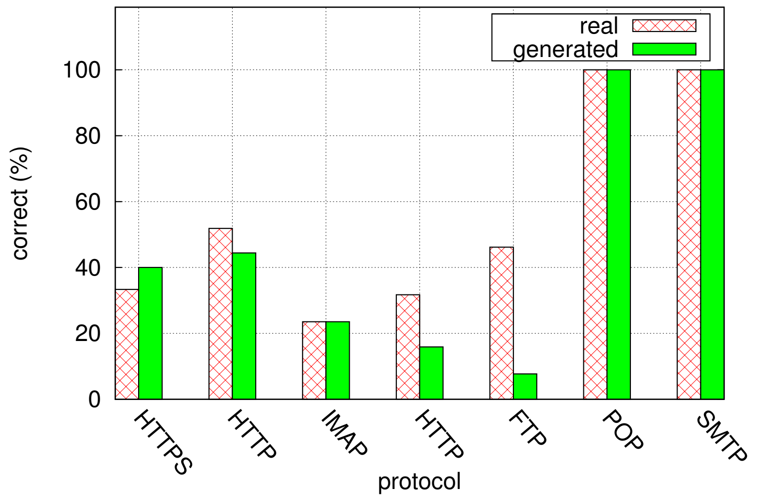


Figure 5.10: BayesNet classification.

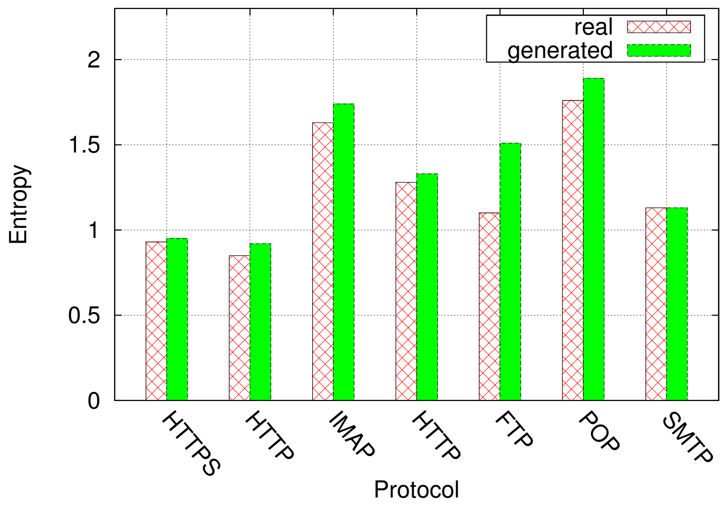


Figure 5.11: Average Entropy per session for *generated* and *actual* timing data.

5.5.1 Experimental Setup

In this section, we describe the setup for the interference testbed and the method used for measuring interference.

5.5.1.1 Interference Testbed

To setup an interference testbed, Five hosts were employed for the needs of our study. `Hermes` and `Hades` are identical Accton mini-routers flashed with our customized OpenWRT firmware. Hermes acted as a legitimate AP for providing connectivity to the Internet and to the rest of the university infrastructure. The selected radio mode was IEEE 802.11g and the operating channel was verified with a monitoring utility to be idle (*i.e.,* no other WLANs operated on the same frequency) during the experiments. Hades was placed in monitor mode and positioned nearby Hermes for broadcasting the decoy traffic (see Section 5.1.3). `Poseidon` is a laptop that was used for measuring its performance degradation during injection and `Zeus` is a workstation used to generate different channel loads. Finally, `compute02` and `compute03` serve as traffic source/sink either for estimating the available bandwidth on Poseidon, or assisting Zeus in channel load generation. Both of them belong to an 8-node cluster (`cluster.cs.columbia.edu`) that is part of the departmental computing facilities. Zeus and Poseidon were associated to Hermes and further connected to `compute` cluster via the campus wired network.

5.5.1.2 Bandwidth estimation

The capacity of Hermes (*i.e.,* Hermes Wi-Fi network) is approximated using the non-intrusive tool `pathrate` [Dovrolis *et al.*, 2004]. Different channel loads where emulated using the invasive `nuttcp` [nuttcp, 2010] tool, whereas the available bandwidth of Poseidon was estimated using `wbest` [Li *et al.*, 2008]. We note that capacity is defined to be the maximum throughput that a network path can provide to an application when there is no competing traffic (*i.e.,* cross traffic). Available bandwidth, on the other hand, is the maximum throughput a path can provide to an application, given the path's current cross traffic load.

5.5.2 Results and Discussion

The testbed created for our experiment, the bandwidth estimation tools employed, as well as our evaluation methodology are all described in great detail in Sections 5.5.1.1 and 5.5.1.2. The two embedded devices (*i.e.,* `Hermes` and `Hades`) that we utilized in our experiments run our customized OpenWRT (v2.6.26 kernel) image, whereas everything else run the *GNU/Linux* OS (v2.6.24 kernel). All our hosts were in multiuser mode, but no other user tasks were running throughout the experiments. We performed our tests late at night so as to ensure that the wired Ethernet would be unloaded and the wireless utilization would be minimal.[6]

Initially, we used `pathrate` to estimate *channel capacity* (from `Zeus` to `compute03`). This would give us an upper-bound of the wireless channel capacity, since pathrate estimates the minimum link capacity among all links on a path. The measurement was repeated 15 times in order to estimate the variance. The reported capacity was 20–22 Mbps. We performed the same test again, but this time Zeus was directly connected to the wired Ethernet (*i.e.,* bypassing the wireless link of Hermes). The reported capacity was 94–97 Mbps, which confirms that the limiting factor was, indeed, the wireless link.

Following that we used `wbest` from `compute02` to `Poseidon`, so as to estimate the *available bandwidth* of the latter. This was done with and without having decoy injection and the resulting degradation in the available bandwidth of Poseidon is the actual performance cost of decoy broadcasting, as observed from an application running on a single host. However, since the available bandwidth is highly coupled to the underlying channel load, and in order to have more pragmatic results, we used the `nuttcp` tool and created *contending traffic* so as to emulate different channel loads. Nuttcp created traffic between Zeus and compute03 at various rates, using 1470-byte UDP datagrams. The actual rates were: 2.2 Mbps, 7.26 Mbps, 11 Mbps, 14.52 Mbps, 16.5 Mbps, and 19.8 Mbps, which correspond to 10%, 33%, 50%, 66%, 75%, and 90%, respectively, of the previously measured wireless channel capacity. Our choice of upper-bound (and not average) capacity results in overes-

[6]We performed our experiments in an idle frequency. However side channels were preoccupied and the DSSS nature of the IEEE 802.11b/g standard imposes interference from operating devices in nearby frequencies.

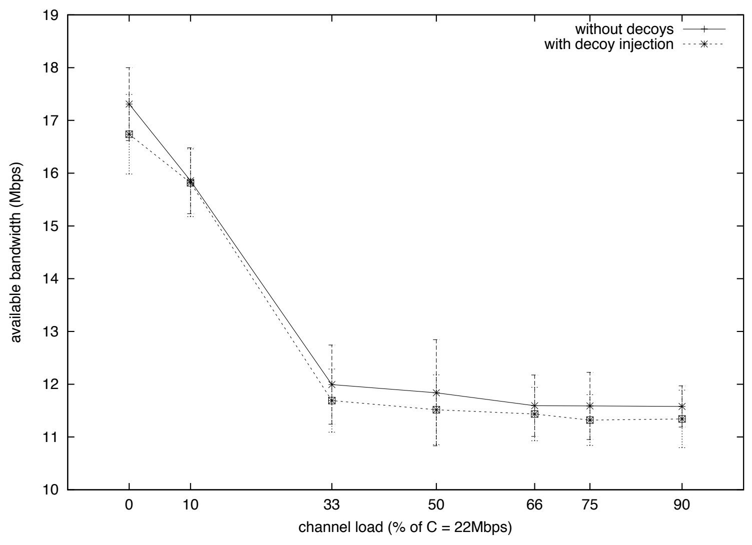


Figure 5.12: Interference cost.

timating the bandwidth degradation due to decoy broadcasting; in other words, our results are pessimistic. Decoy broadcasting was performed from Hades using Tcpreplay and all experiments were repeated 15 times.

The results are illustrated in Figure 5.12. In general terms, there is a decrease in the bandwidth as the channel load increases. However, generating traffic at rates greater than 66% of the channel capacity results in packet loss on Zeus (*i.e.,* Zeus generated traffic at 14.52 Mbps but the actual rate that compute03 reported was 7 Mbps). The MAC layer of IEEE 802.11 gives a fair share to Zeus and Poseidon—the available bandwidth fluctuates between 11 and 12 Mbps. Apparently, the performance degradation due to beacon broadcasting is negligible and the confidence intervals indicate that the difference between the two scenarios is statistically insignificant.

The regulating factor in the whole process is the actual rate at which injection is performed. Notice that we evaluate our proposal under a realistic scenario. The decoy traffic was comprised by 828 packets in total and included an HTTP login into a Gmail account, an FTP login, and an IMAP login. Tcpreplay reported 0.04 Mbps (12pps) since it replays

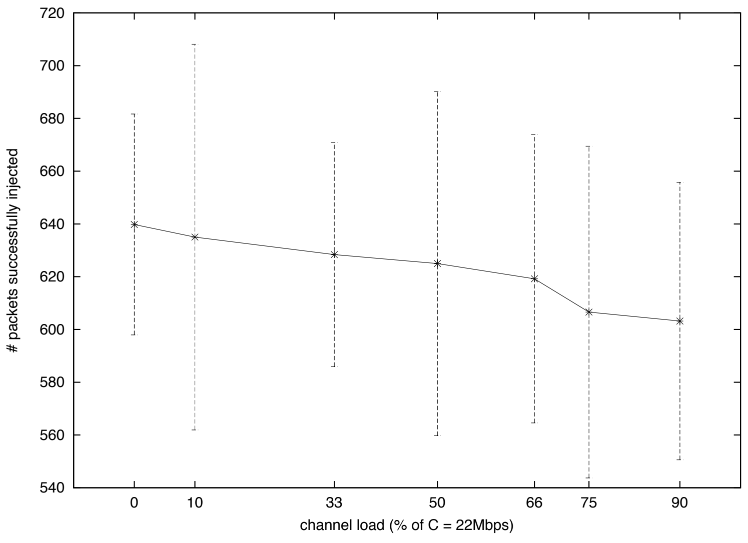


Figure 5.13: Packets successfully injected.

the traffic by *maintaining* the corresponding timing information. Replaying at higher rates will not give us any benefit as Figure 5.13 suggests: number of packets that were injected by Hades and successfully intercepted by Zeus. We used one VAP (see Section 5.1.3) for monitoring the injected packets and a different one for connecting to Hermes and generating traffic. Even at moderate loads we cannot successfully inject the whole set of decoys due to the fact that we suppressed retransmissions and ACKs. The number of packets successfully injected however are a considerable portion of the total 828 set of packets, and hence they demonstrate that there is a relatively high probability of successfully conveying them to a potential snooper. The confidence intervals indicate that the channel load does not significantly affect the number of packets that we can successfully inject. This is mostly due to the slow rate that we used during the replay process.

The experimental testbed, though simple, indicated that deceptive traffic can be broadcasted at negligible cost. Yet, the implications of large-scale deployments on campus-wide Wi-Fi networks cannot be asserted without further experimentation. Mobility, multi-rate support, diversity in traffic patterns (*i.e.,* different packet sizes, burst *vs.* bulk transfers), and the dynamic nature of the wireless medium in general would affect the absolute cost of decoy broadcasting. However, our results are encouraging in that we can perform deceptive traffic injection successfully at a small cost. This will allow us to further investigate larger-scale deployments.

5.6 Legal Considerations

As we already noted in the previous sections, our study relied on protocol specific templates that were manually created and used as input to the decoy API. To increase the diversity of the bogus content, live network traces could be used instead. While employing this approach, numerous legal considerations must be made. More specifically, several US federal privacy laws[7] limit access to network data. The Wiretap Act [18 U.S.C §2511 P1, 2010] regulates the collection of electronic communications contents and in general, this law prohibits third parties from intercepting and recording traffic. The Pen Register and Trap and Trace Act is similar [18 U.S.C §§3121-3127, 2010], but regulates the monitoring of non-content header information for electronic communications. In both of these statutes, there exist several provider exceptions that permit employees of a network operator to record communications to the extent necessary to protect the operator. We believe that this exception would enable our approach of using live network traces as a basis for decoy traffic to be legally deployed in a corporate environment. A more thorough treatment of legal issues surrounding the recording of traffic can be found in [Ohm *et al.*, 2007] and [Matwyshyn *et al.*, 2010].

5.7 Limitations and Open Problems

The focus of this study was limited to TCP traffic and was conducted offline. It is important to point out that this excluded aspects of the 802.11 protocol and broadcast traffic. In our case, it was prudent to exclude these because their inclusion may have overwhelmed the volunteer judges to the point of not participating. However, we believe that our results for the TCP traffic can be extended to the 802.11 protocol transmissions (*e.g.,* management

[7] There are also state laws that vary by state.

frames, control frames, beacons), but it remains open an area for future work. We should also note that in conducting the study offline, as we did, we may have limited the information that might otherwise be available under real- world conditions. It might be possible for an adversary to snoop multiple access points to try and correlate traffic in order to distinguish real traffic from decoys. This scenario was outside the scope of our evaluation. Future work may address this via a large-scale user study and through clustering analysis of captured traces. We also note that an adversary could possibly determine visually that a particular AP is not in use and use this knowledge to distinguish decoy network traffic. Although we did not implement this, the problem can be easily fixed by only broadcasting decoy traffic when there is real traffic.

The believability of our honeyflows stems from the "record, modify, replay" model. Replaying recorded flows can potentially expose sensitive information, but it is information that has already been exposed on the network (although a compromise may have occurred after initial exposure). In employing this strategy, one must consider the tradeoffs (*i.e.,* the replay risk) against the benefit of being able to detect an intruder when you may not have been able to otherwise.

Much of the realism and believability of our generated network traffic was derived original recorded traffic. A significant challenge remains in finding a way to generate traffic without relying on templates and recorded traffic.

5.8 Decoy Networking Summary

Decoy trap-based security defenses, and deception in general, are powerful tools against a wide range of threats in wireless environments. In this chapter, we have demonstrated a system that shows the feasibility of automatically generating large amounts of believable decoy information and showed that it was possible without interfering with normal operations. We used human subjects to evaluate the believability of the generated decoys and showed that is difficult to distinguish from the real thing; our experienced judges achieved only 49% accuracy on average, equivalent to random guessing. We also demonstrated decoy efficacy against automated tools, designed to harvest and exploit credentials in mass by sniffing network transmissions. Moreover, we evaluated our system in a real wireless network that someone was monitoring and successfully detected eavesdropping and exploitation attempts. Considerable work remains to address the potential challenges that active adversaries may pose, such as those that may snoop multiple access points to try and correlate traffic, or those that may use additional sources (like an administrator) to discern decoys without testing them.

Chapter 6

Decoy Host System

In this chapter, we consider attacks against a host system in which an attacker lacks long-term physical access, but has the capability to install malicious software which may be used for long term reconnaissance or to steal information of value. The creation and rapid growth of an underground economy that trades in stolen digital credentials has spurred the growth of crime-driven bots that harvest sensitive data from unsuspecting users. This form of malevolent software employs a variety of techniques ranging from web-based form grabbing and key stroke logging, to screenshots and video capture for the purposes of pilfering data on remote hosts to automate financial crime [Holz *et al.*, 2009; Sthlberg, 2007]. The targets of such malware range from individual users and small companies to the most wealthiest organizations [top, 2009] and recent studies indicate that bot infections are on the rise and up to 9% of the machines in an enterprise are now bot-infected [Higgins, 2009].

Traditional crimeware detection techniques rely on comparing signatures of known malicious instances to identify unknown samples, or on anomaly-based detection techniques in which host behaviors are monitored for large deviations from a baseline. Unfortunately, these approaches suffer a large number of known weaknesses. Signature-based methods can be useful when a signature is known, but due to the large number of possible variants, learning and searching all possible signatures to identify unknown binaries is intractable [Song *et al.*, 2007]. On the other hand, anomaly-based methods are susceptible to false positives and negatives, limiting their potential utility. Consequently, a large amount of existing crimeware now operate undetected by antivirus software. A recent study focused of Zeus[1] (the largest botnet with over 3.6 million PC infections in the US alone [Messmer, 2009]), revealed that the malware bypassed up-to-date antivirus software 55% of the time [zeu, 2009].

Another drawback of conventional host-based antivirus software is that it typically monitors from within the host it is trying to protect, making it vulnerable to evasion or subversion by malware; we see an increasing number of malware attacks that disable defenses such as antivirus software prior to undertaking some malicious activity [Ilett, 2005].

In this work, we introduce BotSwindler, a novel system designed for the proactive detection of credential stealing malware on VM-based hosts. BotSwindler relies upon an out-of-host software agent to drive user simulations that are meant to convince malware residing within the guest OS that it has captured legitimate credentials. By the nature of its out-of-host operating position, the simulator is tamper resistant and difficult to detect by malware residing within the host environment. We posit that malware that detects BotSwindler would need to analyze the behavior of its host and decide whether it is observing a human or not. In other words, the crimeware would need to solve a Turing Test [Turing, 1950]. We assert that if attackers are forced to spend their time looking at the actions on each infected host one by one to determine if they are real or not in order to steal information, BotSwindler would be a success; the attackers' task does not scale. To generate simulations, BotSwindler relies on a formal language that is used to specify a simulation of human user's sequence of actions. The language provides a flexible way to generate variable simulation behaviors that appear realistic. Simulations can be tuned to mimic particular users by using various models for keystroke speed, mouse speed and the frequency of errors made during typing.

One of the challenges in designing an out-of-host simulator lies in the ability to detect the underlying state of the OS. That is, to verify the success or failure of mouse and keyboard events that are passed to the guest OS. For example, if the command is given to open a browser and navigate to a particular URL, the simulator must validate that the URL was successfully opened before proceeding with the next command. To aid in the accuracy and

[1]Zeus uses key-logging techniques to steal sensitive data such as user names, passwords, account numbers. It can be purchased on the black market for $600, complete with support and maintenance [abu, 2009].

realism of the simulations, we developed a low overhead approach, called virtual machine verification (VMV), for verifying whether the state of the guest OS is in one of a predefined set of states.

BotSwindler aims to detect crimeware by deceptively inducing it into an observable action during the exploitation of monitored information injected into the guest OS. To entice attackers with information of value, the system supports a variety of different types of bait credentials including decoy Gmail and PayPal authentication credentials, as well as those from a large financial institution [2]. Our system automatically monitors the decoy accounts for misuse to signal exploitation and thus detect the host infection by credential stealing malware.

BotSwindler presents an instance of a system and approach that can be used to deal with information-level attacks, regardless of their origin. In our prototype, we rely on credentials for financial institutions because they are good examples for which we can easily evaluate, but the approach is aimed at any kind of large-scale automated harvesting of "interesting" data — where "interesting" depends on both the environment and the malware. Although we demonstrate our system with three types of credentials, the system can be extended to support any type of credential that can be monitored for misuse. As one of the contributions of this work, we consider different applications of BotSwindler including how it could be applied practically in an enterprise environment with simulations and decoys adapted to the specific deployment setting. In part of doing so, we discuss how BotSwindler can be deployed to service hosts that include those which are not VM-based, making this approach broadly applicable.

We have implemented a prototype version of BotSwindler using a modified version of QEMU [Bellard, 2005] running on a Linux host. User simulation is implemented using X11 libraries and interaction with the graphical frame buffer. We demonstrate our prototype through experiments with crimeware on a Windows guest, but BotSwindler can operate on any guest operating system supported by the underlying hypervisor or virtual machine monitor (VMM).

[2]By agreement, the institution requested that its name be withheld.

6.0.1 Overview of Results

To demonstrate the effectiveness of BotSwindler, we tested our prototype against real crimeware samples obtained from the wild. Our results from two separate experiments with different types of decoy credentials show that BotSwindler succeeds in detecting malware through attackers' exploitation of the monitored bait. In our first experiment, through experiments with 116 Zeus samples, we received 14 distinct alerts using PayPal and Gmail decoys. In a second experiment with 59 different Zeus samples, we received 3 alerts from our banking decoys.

The long-term viability of BotSwindler defense largely depends on the believability of the bait-injecting simulations by the attackers. We performed a computational analysis to see if attackers could employ machine learning algorithms on keystrokes to distinguish simulations. We present results from experiments running Naive Bayes and Support Vector Machine (SVM) classifiers on real and generated timing data to show that they produce nearly identical classification results making this kind of analysis ineffectual for an adversary. To show that adversaries resorting to manual inspection of the user activities would be sufficiently challenged, we evaluated the believability of user simulations via a decoy Turing Test in which human judges were tasked with trying to distinguish BotSwindler's actions from those of a real human. The failure of the judges to distinguish suggests BotSwindler's simulations are convincingly human-like. In our study with 25 human judges evaluating 10 videos of BotSwindler actions and of a human, the judges' average success rate was 46%, indicating the simulations provide a good approximation of human actions.

Finally, recognizing that attackers may try to distinguish simulated behavior via performance metrics, we evaluated the overhead of our approach by measuring the cost imposed by the virtual machine verification (VMV) technique. Our results indicate that VMV imposes no measurable overhead, making the technique difficult to detect by malware using performance analysis.

In summary, the primary contributions of this chapter include:

- **BotSwindler architecture:** It introduces BotSwindler, a novel, accurate, efficient, and tamper-resistant zero-day crimeware detection system. BotSwindler relies on the use of decoy injection whereby bogus information is used to bait and delude crimeware, causing it to reveal itself during the exploitation of the monitored information.

- **VMSim language:** It introduces VMSim, a new language for expressing simulated user behavior. VMSim facilitates the construction and reproduction of complex user activity, including specifying aggregate statistical behavior.

- **Virtual Machine Verification (VMV):** It introduces virtual machine verification, a low overhead approach for verifying simulation state. VMV enables robust out-of-host user action simulation through graphical state verification.

- **Real malware detection results:** It presents results to show the effectiveness of BotSwindler in detecting real malware when decoy PayPal, Gmail, and banking credentials are injected, stolen, and exploited by the attackers.

- **Statistical and information theoretic analysis:** It presents the results of a computational analysis on generated keystroke timing data to show it would be difficult to detect simulations through analysis with machine learning algorithms or entropy measurements.

- **Believability user study results:** It presents user study results that show the believability of simulations created with BotSwindler's VMSim language.

- **Performance Overhead results:** It shows that BotSwindler imposes no measurable overhead, hence making itself undetectable via timing measurement methods.

6.1 BotSwindler Components

The BotSwindler architecture, as shown in Fig. 6.1, consists of two primary components including a simulator engine, VMSim, and a virtual machine verification component. Another aspect of BotSwindler (although not shown in the figure) are the monitored decoys that we employ for detecting malware. These components are described in the next three sections.

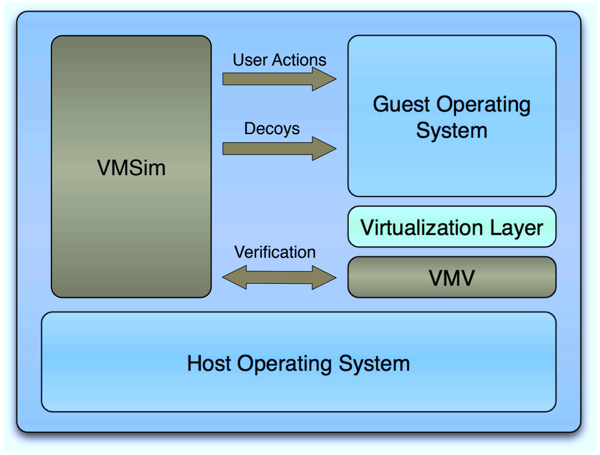


Figure 6.1: BotSwindler architecture.

6.1.1 VMSim

BotSwindler's user simulator component, VMSim, performs simulations that are designed to convince malware residing inside the VM that command sequences are genuine. We posit that successfully creating a sequence of actions that tricks the malware into stealing and uploading a decoy credential can be achieved only if two essential requirements are met:

1. the simulator process remains undetected by the malware
2. the actions of the simulator appear to be generated by a human

We approach the first requirement by decoupling the location of where the simulation process is executed and where its actions are received. To do this, we run the simulator outside of a virtual machine and pass its actions to the guest host by utilizing the X-Window subsystem on the native host. The second requirement is addressed through a simulation creation process that entails recording, modifying, and replaying mouse and keyboard events captured from real users. To support this process, we leverage the Xorg Record and XTest extension libraries for recording and replaying X-Window events. The product is a simulator that runs on the native host producing human-like events without introducing technical artifacts that could be used to alert malware of the BotSwindler facade.

```
<ActionType> ::= <WinLogin> <ActionType>
        | <CoverType> <ActionType> | <CarryType> <ActionType>
        | <WinLogout> | <VerifyAction> <ActionType> | e
<CoverAction> ::= <BrowserAction> <CoverAction>
        | <WordAction> <CoverAction> | <SysAction> <CoverAction>
<BrowserAction> ::= <URLRequest> <BrowserAction>
        | <OpenLink> <BrowserAction> | <Close>
<WordAction> ::= <NewDoc> <WordAction>
        | <EditDoc> <WordAction> | <Close>
<SysAction> ::= <OpenWindow> | <MaxWindow>
        | <MinWindow> | <CloseWindow>
<VerifyAction> ::= Img1 | Img2 | ... | ImgN | Unknown
<CarryAction> ::= <PayPalInject> | <GmailInject>
        | <CCInject> | <UnivInject> | <BankInject>
```

Figure 6.2: VMSim language.

VMSim relies on formal language to specify the sequence of actions in the simulations. Representative details of the formal language are provided in Fig. 6.2 (many details are omitted due to space limitations). The language provides a flexible way to generate variable simulation behaviors and workflows, but more importantly it supports the use of *cover* and *carry* actions; carry actions result in the injection of decoys (described in Sect. 4.2), whereas cover actions include everything else to support the believability of carry traffic. For example, cover actions may include the opening and editing of a text document (`WordActions`) or the opening and closing of particular windows (`SysActions`). The `VerifyAction` allows VMSim to interact with VMV (described in Sect. 6.1.2) and provides support for conditional operations, synchronization, and error checking. Interaction with the VMV is crucial for the accuracy of simulations because a particular action may cause random delays for which the simulation must block on before proceeding to the next action.

The simulation creation process involves the capturing of mouse and keyboard events of a real user as distinct actions. The actions that are recorded map to the constructs of the VMSim language. Once the actions are implemented, the simulator is tuned to mimic a particular user by using various biometric models for keystroke speed, mouse speed, mouse distance, and the frequency of errors made during typing. These parameters function as controls over the language shown in Fig. 6.2 and aid in creating variability in the simulations. Depending on the particular simulation, other parameters such as URLs or other text that must be typed are then entered to adapt each action. VMSim translates the language's actions into lower level constructs consisting of keyboard and mouse functions, which are then output as X protocol level data that can be replayed via the XTest extensions.

To construct biometric models for individuals, we have extended QEMU's VMM to support the recording of several features including keycodes (the ASCII code representing a key), the duration for which they are pressed, keystroke error rates, mouse movement speed, and mouse movement distance. Generative models for keystroke timing are created by first dividing the recorded data for each keycode pair into separate classes where each class is determined by the distance in standard deviations from the mean. We then calculate the distribution for each keycode sequence as the number of instances of each class. We adapt simulation keystroke timing to profiles of individual users by generating random times that are bounded by the class distribution. Similarly, for mouse movements we calculate user specific profiles for speed and distance. Recorded mouse movements are broken down into variable length vectors that represent periods of mouse activity. We then calculate distributions for each user using these vectors. The mouse movement distributions are used as parameters for tuning the simulator actions. We note that identifying the complete set of features to model an individual is an open problem. Our selection of these features is to illustrate a feasible approach to generating statistically similar actions and represent just a small subset of the sources for human variability. In addition, these features have been useful for verifying the identify of individuals in keystroke and mouse dynamics studies [Monrose and Rubin, 1997; Ahmed and Traore, 2007]. More sophisticated models could be created by considering aspects such as the particular task a simulation executing and types of mouse movement, which have been shown to be a sources of variability [Shneiderman, 1984;

Jay *et al.*, 2007].

In Sect. 6.2 we provide a statistical and information theoretic analysis of the generated simulated times.

One of the advantages of using a language for the generation of simulation workflows is that it produces a specification that can be ported across different platforms. This allows the cost of producing various simulation workflows to be amortized over time. In the prototype version of BotSwindler, the task of mapping mouse and keyboard events to language actions is performed manually. The mappings of actions to lower level mouse and keyboard events are tied to particular host configurations. Although we have not implemented this for the prototype version of BotSwindler, the process of porting these mappings across hosts can be automated using techniques that rely on graphical artifacts like those used in the VMV implementation and applying geometric transformations to them.

Once the simulations are created, playing them back requires VMSim to have access to the display of the guest OS. During playback, VMSim automatically detects the position of the virtual machine window and adjusts the coordinates to reflect the changes. Although the prototype version of BotSwindler relies on the display to be open, it is possible to mitigate this requirement by using the X virtual frame buffer (Xvfb) [xvf, 2009]. By doing so, there would be no requirement to have a screen or input device.

6.1.1.1 Statistically Similar Temporal Features

In cases where real key-stroke timing data is available, it can be used to build generative models for VMSim's temporal features. One approach we have used for building these models relies on analyzing key-stroke timing data to gather metrics. In particular, we models using keystroke bi-grams (e.g., in the word "the", we model times between t-h and h-e.) that take in account how long a key is pressed and the time between key strokes. This can be done for all combination of keys. For a given bi-gram, we use the recorded times to calculate the mean and standard deviations for key-press times and the time between presses for a given individual. We then calculate a distribution of times relative to the standard deviation. This becomes the timing model for a particular individual. We then generate new random key-stroke times using this model as a control for the minimum and maximum times. This approach allows for an unbounded amount of variability in the temporal features for synthesized key-strokes. The variability can be controlled by parameters that determine how closely the generated timing features conform to the session's timing distribution. We provide an evaluation statistical similarities of the synthesized key-stroke times to recorded times using classification techniques in 6.2.

6.1.2 Virtual Machine Verification

The primary challenge in creating an of out-of-host user simulator is to generate human-like events in the face of variable host responses. This task is essential for being able to tolerate and recover from unpredictable events caused by things like the fluctuations in network latency, OS performance issues, and changes to web content. Conventional in-host simulators have access to OS APIs that allow them to easily to determine such things. For example, simulations created with the popular tool AutoIt can call its `WinWait` function, which can use the Win32 API to obtain information on whether a window was successfully opened. In contrast, an out-of-host simulator has no such API readily available. Although the Xorg Record extensions do support synchronization to solve this sort of problem, they are not sufficient for this particular case. The Record extensions require synchronization on an X11 window as opposed to a window of the guest OS inside of an X11 window, which is the case for guest OS windows of a VM[3].

We address this requirement by casting it as a verification problem to decide whether the current VM state is in one of a predefined set of states. In this case, the states are defined from select regions of the VM graphical output, allowing states to consist of any visual artifact present in a simulation workflow. To support non-deterministic simulations, we note that each transition may end in one of several possible next states. We formalize the VMV process over the set of transitions T, and set of states S, where each $t_0, t_1, ..., t_n \in T$ can result in the the set of states $s_{t1}, s_{t2}, ..., s_{tn} \subseteq S$. The VMV decides a state verified for a current state c, when $c \in s_{ti}$.

The choice for relying on the graphical output allows the simulator to depend on the same

[3]This was also a challenge when we tested under VMware Unity, which exports guest OS windows as what appear to be ordinary windows on the native host.

```
//initialize xleft, xright, ybottom, ytop to be a
//bounding box around the verification image
for(x = xleft; x < xright;x++) {
   for(y = ytop; y < ybottom; y++)  {
         // Load the pixels from the guest and verification image
         cg = getpixel(monitor_screen, x, y);
         cm = getpixel(guest_screen, x, y);

         // If this is not a match, see if the limit is reached
         if (((cg & fmtg->Rmask) != (cm & fmtg->Rmask)) ||
         ((cg & fmtg->Gmask) != (cm & fmtg->Gmask)) ||
         ((cg & fmtg->Bmask) != (cm & fmtg->Bmask)))
                   if (bad_pixel_count > threshold)
                            return no_match;
                   else
                            bad_pixel_count++;
   }
}
```

Figure 6.3: VMV pseudocode of the monitor function.

graphical features a user would see and respond to, enabling more accurate simulations. In addition, information specific to a VM's graphical output can be obtained from outside of the guest without having to solve the semantic gap problem [Chen and Noble, 2001], which requires detailed knowledge of the underlying architecture. A benefit of our approach is that it can be ported across multiple VM platforms and guest OS's. In addition, we do not have to be concerned with side effects of hostile code exploiting a system and interfering with the Win32 API like traditional in-host simulators do, because we do not rely on it. In experiments with AutoIt scripts and in-host simulations, we encountered cases where scripts would fail as a result of the host being infected with malware.

The VMV was implemented by extending the Simple DirectMedia Layer (SDL) component of QEMU's [Bellard, 2005] VMM. Specifically, we added a hook to the `sdl_update` function to call a VMV `monitor` function. Figure 6.3 shows pseudocode for the verification procedure that is invoked. The code performs a pixel comparison between the guest screen and a verification image. Instead of failing after a single mismatch, the procedure uses an established threshold of bad pixels to account for minor discrepancies that may occure. For example, the presence of a mouse pointer on the screen would cause a verification failure if we did not allow for it. Hooking this code into into the `sdl_update` function results in the VMV being invoked every time the VM's screen is refreshed. The choice of invoking the VMV only during `sdl_update` was both to reduce the performance costs and because it is precisely when there are updates to the screen that we seek to verify states (it is a good indicator of user activity).

States are defined during a simulation creation process using a pixel selection tool (activated by hotkeys) that we built into the VMM. The pixel selection tool allows the simulation creator to select any portion of a guest OS's screen for use as a state. In practice, the states should be defined for any event that may cause a simulation to delay (*e.g.,* network login, opening an application, navigating to a web page). The size of the screen selection is left up to the discretion of the simulation creator, but typically should be minimized as it may impact performance. In Sect. 6.4 we provide a performance analysis to aid in this consideration.

6.2 Statistical and Information Theoretic Analysis

In this section we present results from the statistical analysis of generated keystroke timing information. The goal of these experiments was to see if a machine learning algorithm (one that would be available to a malware sample to determine whether keystrokes are real or not) might be able to classify keystrokes accurately into user generated or machine generated. For these experiments, we relied on Killourhy and Maxion's benchmark data set [Killourhy and Maxion, 2009]. The data set was created by having 51 subjects repeatedly type the same 10 character password, 50 times in 8 separate sessions, to create 400 samples

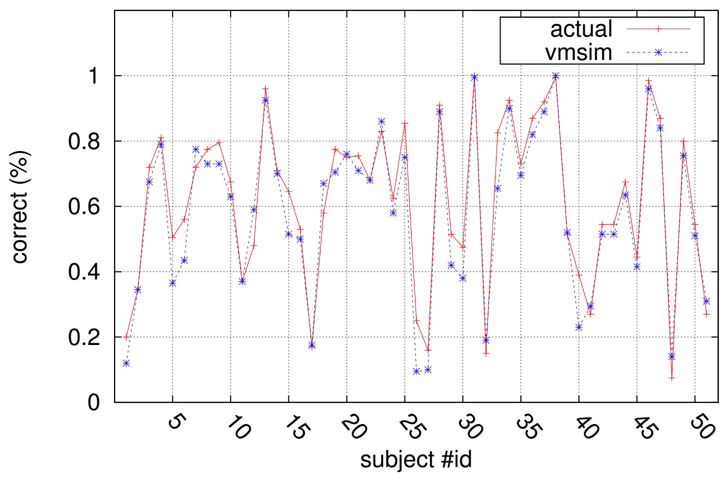


Figure 6.4: SVM classification.

for each user. Accurate timestamps were recorded by using an external clock. Using this publicly available real user data ensures that experiments can be repeated.

6.2.1 Classification Experiments

For our initial evaluation of VMSim's generated timing information, we used Weka [Hall *et al.*, 2009] to conduct classification experiments. We divided the benchmark data set in half and used 200 password timing vectors from each user to train Naive Bayes and Support Vector Machine (SVM) classifiers. We selected these classifiers because they represent opposite ends of the spectrum in terms of their classification abilities. To evaluate the strength of the models, we conducted a 10-fold analysis whereby 10% of the data is tested against a model created from the remaining data. The test is repeated ten times by varying the test data so that all of the data is used for testing. The average number of instances classified correctly under 10-fold analysis for the Naive Bayes and SVM classifiers were 75% and 76% respectively.

To generate the simulation data, we used the remaining 200 timing vectors from each user were used as input to VMSim's generation process to generate 200 new timing vectors for each user. The same 200 samples were used for testing against the generated samples in the classification experiments. Note that we only used fields corresponding to hold times

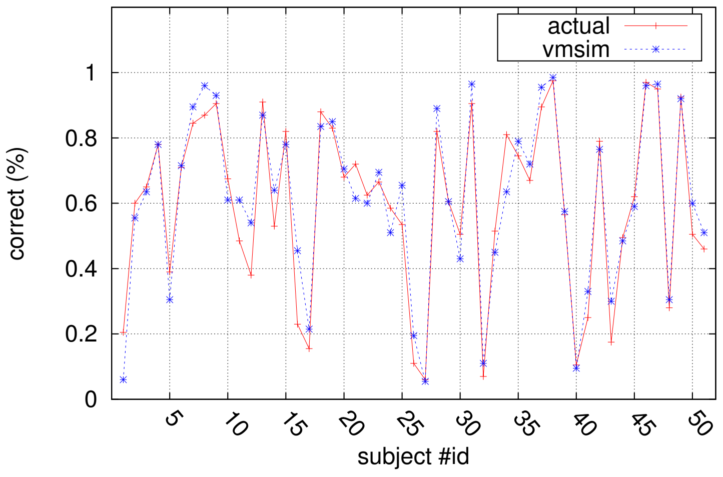


Figure 6.5: Naive Bayes classification.

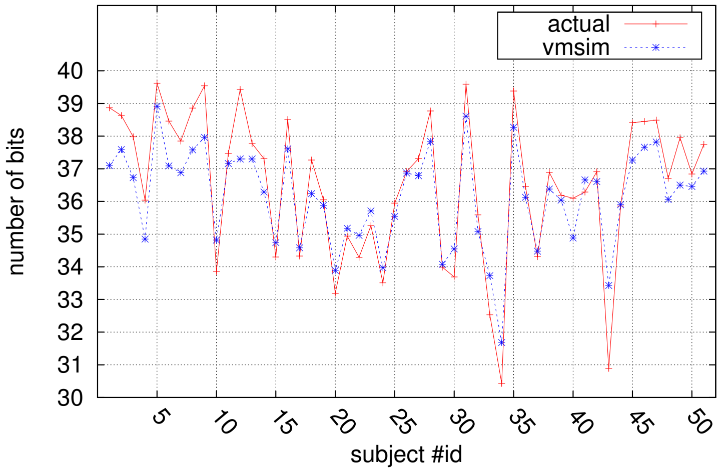


Figure 6.6: Entropy of *generated* and *actual* timing data.

Detector	**FP Rate**	**TP Rate**
Euclidean	.05	.11
Manhattan	.05	.10
Scaled Manhattan	.05	.03

Table 6.1: Average true positive rates for three anomaly detectors tested against the simulated keystroke timing data.

and inter-key latencies because the rest were not applicable to this work (they can also contain negative values). The results of running the SVM and Naive Bayes classifiers on the generated data and real data are presented in Figs. 6.4 and 6.5, respectively. The results are nearly identical for these two classifiers suggesting that this particular type of analysis would probably not be useful for an attacker attempting to distinguish the real from generated actions.

In Fig. 6.6, we present a comparison of entropy values (the amount of information or bits required to represent the data) [Lee and Xiang, 2001]for the actual and generated data for each of the 200 timing vectors of the 51 test subjects. The results suggest that there is no loss of information in our generation process that would be useful by an adversary that is attempting distinguish real from generated actions.

6.2.2 Anomaly Detection Experiments

In a second set of experiments, we relied on several classic anomaly detection techniques including the Euclidean distance [Duda *et al.*, 2000], Manhattan distance, and scaled Manhattan distance [Araújo *et al.*, 2005]. Each of these algorithms (among others) were evaluated against one another in [Killourhy and Maxion, 2009]. The scaled Manhattan distance was shown to have the best equal error rate among all algorithms tested. We selected it for this reason and arbitrarily selected two others for comparison.

Our evaluation was performed by dividing the benchmark data set in half and using 200 password timing vectors from each user as training data. The remaining 200 timing vectors from each user were used as input to VMSim's generation process to generate 200 new

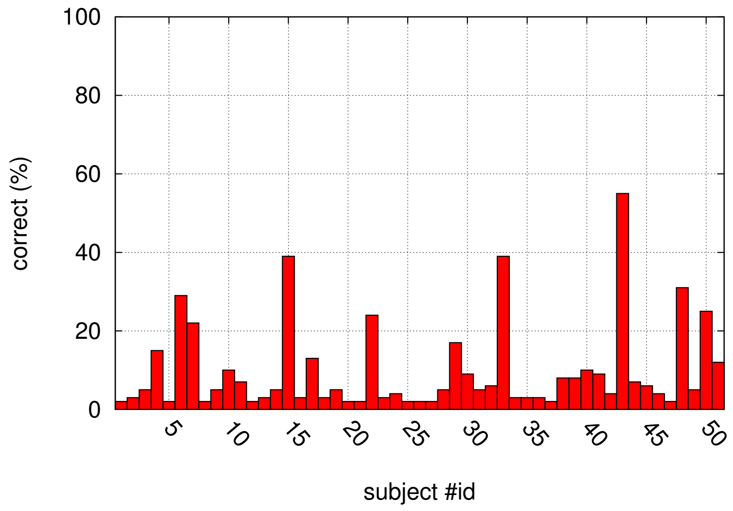


Figure 6.7: True positive rates for anomaly detection using the Euclidean distance and a threshold set with a .05 FP rate.

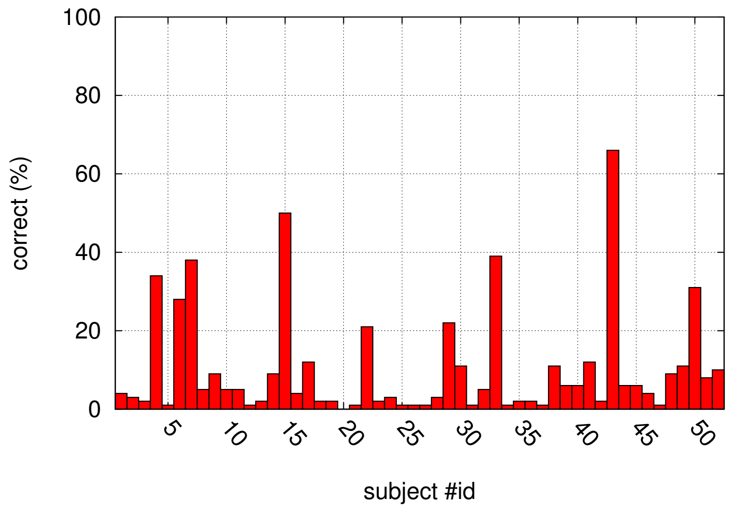


Figure 6.8: True positive rates for anomaly detection using the Manhattan distance and a threshold set with a .05 FP rate.

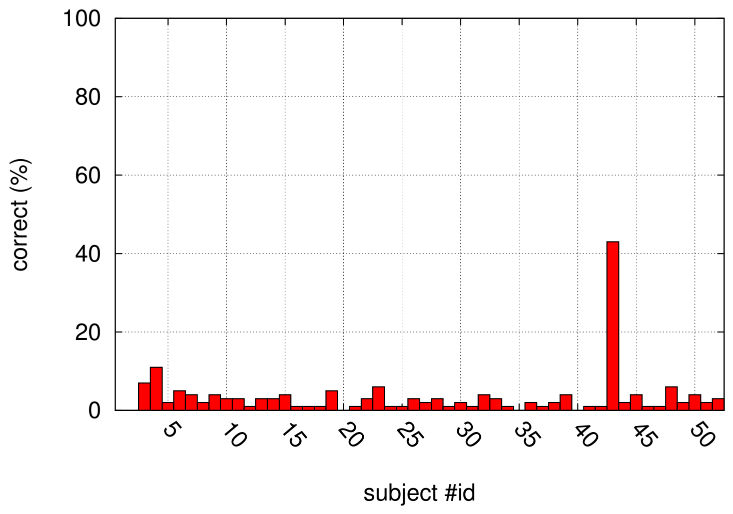


Figure 6.9: True positive rates for anomaly detection using the (scaled) Manhattan distance and a threshold set with a .05 FP rate.

timing vectors for each user. The same 200 samples were then used to establish thresholds for each of the algorithms using a .05 false positive rate. We chose to use the .05 false positive rate (as opposed to .01) to allow for greater resolution within the true positive results. The results of each of the algorithms tested on the individual users are presented in Figure 6.7 for the Euclidean distance, Figure 6.2.2 for the Manhattan distance, and Figure 6.2.2 for the scaled Manhattan distance. Table 6.2.2 presents the mean true positive rates for each of the algorithms on the generated timing vectors. Overall, the average true positive rate across all of the algorithms tested was less than 10% suggesting that these techniques alone would not be very effective in distinguishing the generated timing data.

6.3 Decoy Turing Test

We now discuss the results of a Turing Test [Turing, 1950] to demonstrate BotSwindler's performance regarding the *humanness*, or believability, of the generated simulations. The point of this experiments is to show that adversaries resorting to manual inspection of the user activities would be sufficiently challenged. Although the simulations are designed to delude crimeware, here we focus on convincing humans, a task we posit to be a more difficult feat, making the adversaries task of designing malware that discerns decoys far more difficult. To conduct our initial study, we formed a pool of 25 human judges, consisting of security-minded PhDs, graduate-level students, and security professionals. Their task was to observe a set of 10 videos that capture typical user actions performed on a host and make a binary decision about each video: *real* or *simulated* (*i.e.,* whether the video shows the actions of a real user or those of a simulator). [4].

Our goal was to demonstrate the believability of the simulated actions by showing failure of human judges to reliably distinguish between authentic human actions and those generated with BotSwindler. Our videos contained typical user actions performed on a host such as composing and sending an email message through Gmail, logging into a website of a financial institution such as Citibank or PayPal, and editing text document using Wordpad. For each scenario we generated two videos: one that captured the task performed by a human and another one that had the same task performed by BotSwindler. Each video was designed to be less than a minute long since we assumed that our judges would have limited patience and would not tolerate long-running simulations.

The human generated video samples were created by an independent user who was asked to perform sets of actions which were recorded with a desktop recording tool to obtain the video. Similar actions by another user were used to generate keystroke timing and error models, which could then be used by VMSim to generate keystroke sequences. To generate mouse movements, we rely on movements recorded from a real user. Using these, we experimentally determine upper and lower bounds for mouse movement speed and replay the movements from the real user, but with a new speed randomized within the determined limits. The keyboard and mouse sequences were merged with appropriate simulator parameters such as credentials and URLs to form the simulated sequence which was used to create the decoy videos.

Figure 6.10 summarizes the results of our initial study for each of the 10 videos. The videos are grouped in per-scenario pairs in which the left bars correspond to simulated tasks, while the right bars correspond to the tasks of authentic users on which the simulations are based. The height of the bars reflects the number of judges that correctly identified the

[4]A screenshot of the webpage presented to the judges and the IRB approval for this study can be found in Chapter 9.

given task as real or simulated. The overall success rate was ∼46%, which indicates that VMSim achieves a good approximation of human behavior. The ideal success rate is 50%, which suggests that judges cannot differentiate whether a task is simulated or real.

Figure 6.11 illustrates the overall performance of each judge from our first study. The judges' correctness varies greatly from 0% up to 90%. This variability can be attributed to the fact that each judge interprets the same observed feature differently. For example, since VMSim uses real user actions as templates to drive the simulation, it is able to include advanced "humanized" actions inside simulations, such as errors in typing (*e.g.,* invalid typing of a URL that is subsequently corrected), `TAB` usage for navigating among form fields, auto-complete utilization, and so forth. However, the same action (*e.g.,* `TAB` usage for navigating inside the fields of a web form) is assumed by some judges as a real human indicator, while some others take it as a simulation artifact. This observation is clearly a "toss up" as a distinguishing feature. In this study, feven the highly successful judges could not achieve a 100% accuracy rate. This indicates that given a diverse and plentiful supply of decoys, our system will be believable at some time. In other words, given enough decoys, BotSwindler will eventually cause the malware to reveal itself.

We conducted a second study to see if the order in which the videos were presented to the judges would make a difference. This time, instead of presenting judges with both real and simulated videos for each scenario, we only gave them a single video for each scenario that was either real or simulated. We also randomized the order in which the videos were presented to the judges [5] to control for any bias that ordering may introduce.

The second study relied on a pool of twenty judges. The first ten judges were presented with a list of 5 videos randomly selected between those showing real user actions and those generated by VMSim. Judges 11-15 were given all real videos and judges 16-20 were given videos of simulated actions with both sets serving as control groups. The results for each of the judges are presented in Figure 6.12. Overall, their accuracy was 52%, reaffirming results obtained in the first study and suggesting that the order of the videos was inconsequential. We note that when looking at the control groups alone there appears to be a bias for judges to choose real. The control group containing only simulated videos had an average of 20%

[5]The JavaScript used for the randomization is presented in Chapter 9.

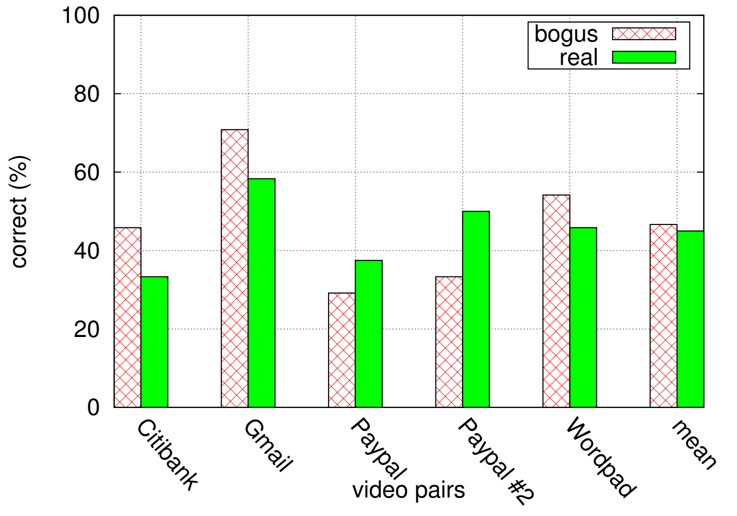


Figure 6.10: Decoy Turing Test results: *real* vs. *simulated*.

correct (most judges chose real) whereas the control group given all real videos had an average of 72% correct. The fact that we only presented judges with a single video for each scenario (as opposed to one real and one simulated) may have contributed to this result. This trend was not apparent in the first study. Moreover, in considering the overall results for the 20 judges, the bias was not evident. Both studies produced results that are statistically equivalent to 50% for significance $p > .05$.

The overall results indicate that simulations are highly believable by humans. In cases where they may not be, it is important to remember that the task of fooling humans is far harder than tricking malware, unless the adversary has solved the AI problem and designed malware to answer the Turing Test. Furthermore, if attackers have to spend their time looking at the actions one by one to determine if they are real or not, we consider BotSwindler a success because that approach does not scale for the adversary.

6.4 Virtual Machine Verification Overhead

The overhead of the VMV in BotSwindler is controlled by several parameters including the number of pixels in the screen selections, the size of the search area for a selection, the number of possible states to verify at each point of time, and the number of pixels required to

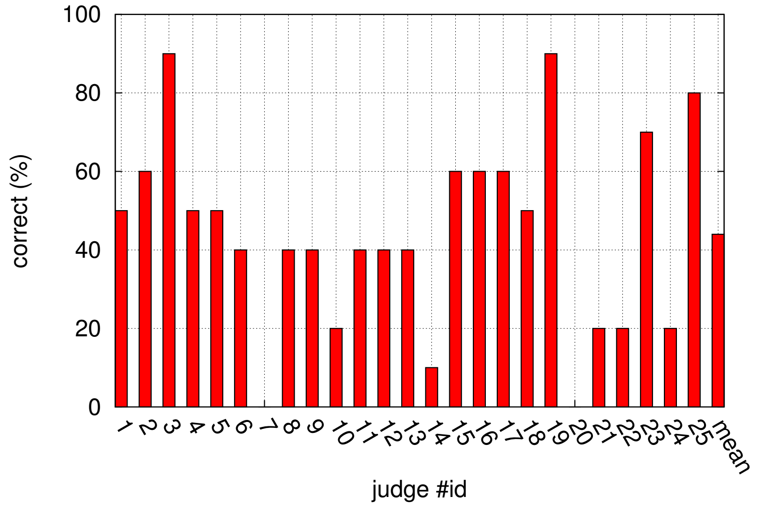


Figure 6.11: Judges' overall performance in our initial study in which judges were presented with real and bogus videos for each of the five senarios.

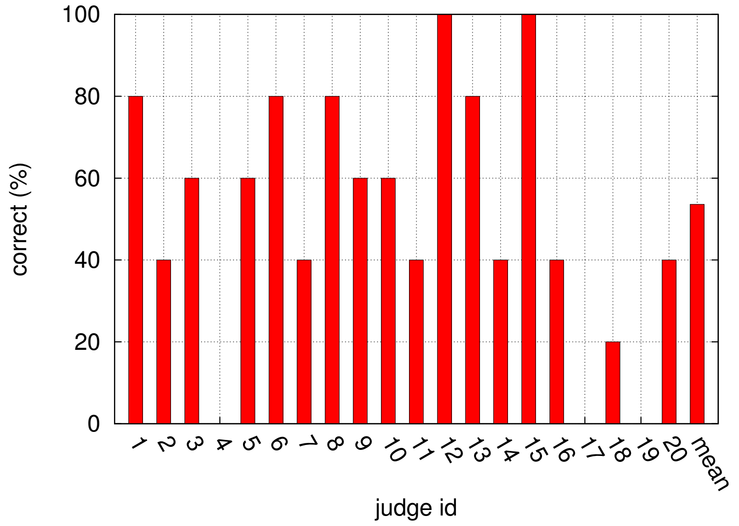


Figure 6.12: Judges' overall performance in our second study in which the order of the videos were randomized and judges were presented with only a single video for each of five scenarios.

match for positive verification. A key observation responsible for maintaining low overhead is that the majority of the time, the VMV process results in a negative verification, which is typically obtained by inspecting a single pixel for each of the possible states to verify. The performance cost of this result is simply that of a few instructions to perform pixel comparisons. The worst case occurs when there is a complete match in which all pixels are compared (*i.e.,* all pixels up to some predefined threshold). This may result in thousands of instructions being executed (depending on the particular screen selection chosen by the simulation creator), but it only happens once during the verification of a particular state. It is possible to construct a scenario in which worse performance is obtained by choosing screen selections that are common (*e.g.,* found on the desktop) and almost completely matches but results in a negative VMV outcome. In this case, obtaining a negative VMV result may cost hundreds of thousands of CPU cycles. In practice, we have not found this scenario to occur; moreover, it can be avoided by the simulation creator.

Table 6.2: Overhead of VMV with idle user.

	Min.	**Max.**	**Avg.**	**STD**
Native OS	.48	.70	.56	.06
QEMU	.55	.95	.62	.07
QEMU w/VMV	.52	.77	.64	.07

Table 6.3: Overhead of VMV with active user.

	Min.	**Max.**	**Avg.**	**STD**
Native OS	.50	.72	.56	.06
QEMU	.57	.96	.71	.07
QEMU w/VMV	.53	.89	.71	.06

In Table 6.2, we present the analysis of the overhead of QEMU[6] with the BotSwindler extensions. The table presents the amount of time, in seconds, to load web pages on our test machine (2.33GHz Intel Core 2 Duo with 2GB 667MHz DDR2 SDRAM) with idle user activity. The results include the time for a native OS, an unmodified version of QEMU

[6]QEMU does not support graphics acceleration, so all processing is performed by the CPU.

(version 0.10.5) running Windows XP, and QEMU running Windows XP with the VMV processing a verification task (a particular state defined by thousands of pixels).

In Table 6.3, we present the results from a second set of tests where we introduce rapid window movements forcing the screen to constantly be refreshed. By doing this, we ensure that the BotSwindler VMV functions are repeatedly called. The results indicate that the rapid movements do not impact the performance on the native OS, whereas in the case of QEMU they result in a $\sim$15% slowdown. This is likely because QEMU does not support graphics acceleration, so all processing is performed by the CPU. The time to load the web pages on QEMU with the VMV is essentially the same as without it. This is true whether the tests are done with or without user activity. Hence, we conclude that the performance overhead of the VMV is negligible.

6.5 Detecting Real Malware with Bait Exploitation

To demonstrate the efficacy of our approach, we conducted two experiments using BotSwindler against crimeware found in the wild. For the first experiment, we injected Gmail and PayPal decoys, and for second, we used decoy banking logins. The experiments relied on Zeus because it is the largest botnet in operation. Zeus is sold as a crimeware kit allowing malicious individuals to create and configure their own unique botnets. Hence, it functions as a payload dissemination framework with a large number of variants. Despite the abundant supply of Zeus variants, many are no longer functional because they require active command and control servers to effectively operate. This requirement gives Zeus a relatively short life span because these services become inactive (*e.g.,* they are on a compromised host that is discovered and sanitized). To obtain active Zeus variants, we subscribed to an active feed of binaries at the Swiss Security blog, which has a Zeus Tracker [abu, 2009] and Offensive Computing [7].

In our first experiment, we used 5 PayPal decoys and 5 Gmail decoys. We deliberately limited the number of accounts to avoid upsetting the providers and having our access removed. After all, the use of these accounts as decoys requires us to continuously poll the

[7]`http://www.offensivecomputing.net`

servers for unauthorized logins as described in Sect. 4.2.1, which could become problematic with a large number of accounts. To further limit the load on the services, we limited the BotSwindler monitoring to once every hour.

We constructed a BotSwindler sandbox environment so that any access to `www.paypal.com` would be routed to a decoy website that replicates the look-and-feel of the true PayPal site. This was done for two reasons. First, if BotSwindler accessed the real PayPal site, it would be more difficult for the monitor to differentiate access by the simulator from an attacker, which could lead to false positives. More importantly, hosting a phony PayPal site enabled us to control attributes of the account (*e.g.,* balance and verified status) to make them more enticing to crimeware. We leveraged this ability to give each of our decoy accounts unique balances in the range of $4,000 - $20,000 USD, whereas in the true PayPal site, they have no balance. In the case of Gmail, the simulator logs directly into the real Gmail site, since it does not interfere with monitoring of the accounts (we can filter on IP) and there is no need to modify account attributes.

The decoy PayPal environment was setup by copying and slightly modifying the content from `www.paypal.com` to a restricted lab machine with internal access only. The BotSwindler host machine was configured with NAT rules to redirect any access directed to the real PayPal website to our test machine. The downside of using this setup is that we lack a certificate to the `www.paypal.com` domain signed by a trusted Certificate Authority. To mitigate the issue, we used a self-signed certificate that is installed as a trusted certificate on the guest. Although this is a potential distinguishing feature that can be used by malware to detect the environment, existing malware is unlikely to check for this. Hence, it remains a valid approach for demonstrating the use of decoys to detect malware in this proof of concept experiment. The banking logins used in the second experiment do not have this limitation, but they may not have the same broad appeal to attackers that make PayPal accounts so useful.

The experiments worked by automating the download and installation of individual malware samples using a remote network transfer. For each sample, BotSwindler conducted various simulations designed from the VMSim language to contain inject actions, as well as other cover actions. The simulator was run for approximately 20 minutes on each of the

116 binaries that were tested with the goal of determining whether attackers would take and exploit the bait credentials. Over the course of five days of monitoring, we received thirteen alerts from the PayPal monitor and one Gmail alert. We ended the study after five days because the results obtained during this period were enough to convince us the system worked[8]. The Gmail alert was for a Gmail decoy ID that was also associated with a decoy PayPal account; the Gmail username was also a PayPal username and both credentials were used in the same workflow (we associate multiple accounts to make a decoy identity more convincing). Given that we received an alert for the PayPal ID as well, it is likely both sets of credentials were stolen at the same time. Although the Gmail monitor does provide IP address information, we could not obtain it in this case. This particular alert was generated because Gmail detected suspicious activity on the account and locked it, so presumably the intruder never got in.

We attribute the fewer Gmail alerts to the economics of the black market. Although Gmail accounts may have value for activities such as spamming, they can be purchased by the thousands for very little cost[9] and there are inexpensive tools that can be used to create them automatically. Hence, attackers have little incentive to build or purchase a malware mechanism, and to find a way to distribute it to many victims, only to net a bunch of relatively valueless Gmail accounts. On the other hand, high-balance verified PayPal accounts represent something of significant value to attackers. The 2008 Symantec Global Internet Security Threat Report [Symantec, 2008] lists bank accounts as being worth \$10-\$1000 on the underground market, depending on balance.

For the PayPal alerts that were generated, we found that some alerts were triggered within an hour after the corresponding decoy was injected, where other alerts occurred days after. We believe this variability to be a consequence of attackers manually testing the decoys rather than testing through some automatic means. In regards to the quantity of alerts generated, there are several possible explanations that include:

- as a result of the one-to-many mapping between decoys and binaries, the decoys are

[8]We ended the study after 5 days, but a recent examination of the monitoring logs revealed alerts still being generated months after.

[9]We have found Gmail accounts being sold at \$20 per 1000.

exfiltrated to many different dropzones where they are then tested

- the decoy accounts are being sold and resold in the underground market where first the dropzone owner checks them, then resell them to others, who then resell them to others who check them

While the second case is conceivable for credentials of true value, our decoys lack any balance. Hence, we believe that once this fact is revealed to the attacker during the initial check, the attackers have no reason to keep the credentials or recheck them (lending support for the first case). We used only five PayPal accounts with a one-to-many mapping to binaries, making it impossible to know exactly which binary triggered the alert and which scenario actually occurred. We also note that the number of actual attacks may be greater than what was actually detected. The PayPal monitor polls only once per hour, so we do not know when there are multiple attacks in a single hour. Hence, the number of attacks we detected is a lower bound. In addition, despite our efforts to get active binaries, many were found to be inactive, some cause the system to fail, and some have objectives other than stealing credentials.

In the second experiment, we relied on several bank accounts containing balances over $1,000 USD. In contrast with the PayPal experiments, this experiment relied on an actual bank website with authentic SSL certificates. The bank account balances were frozen so that money could not actually be withdrawn. We ran the simulator for approximately 10 minutes on 59 new binaries. Over the course of five days of monitoring, we received 3 alerts from the collaborating financial institution. The point of these experiments is to show that decoy injection can be useful tool for detecting crimeware that can be difficult to detect through traditional means. These results validate the use of financial decoys for detecting crimeware. A BotSwindler system fully developed as a deployable product would naturally include many more decoys and a management system that would store information about which decoy was used and when it was exposed to the specific tested host.

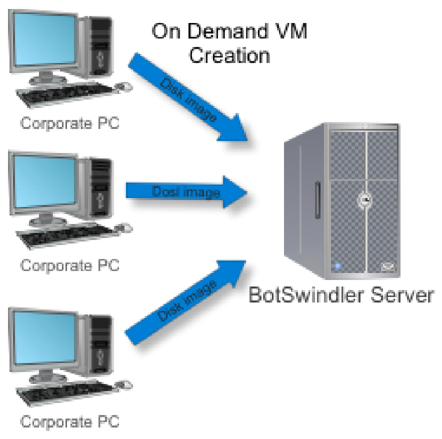


Figure 6.13: Enterprise Deployment of BotSwindler

6.6 Applications of BotSwindler in an Enterprise

Beyond the detection of malware using general decoys, BotSwindler is well suited for use in an enterprise environment where the primary goal is to monitor for site-specific credential misuse and to profile attackers targeting that specific environment. Since the types of credentials that are used within an enterprise are typically limited to business applications for specific job functions, rather than general purpose uses, it is feasible for BotSwindler to provide complete test coverage in this case. For example, typical corporate users have a single set of credentials for navigating their company intranet. Corporate decoy credentials could be used by BotSwindler in conducting simulations modeled after individuals within the corporation. These simulations may emulate system administrative account usage (*i.e.,* logging in as root), access to internal databases, editing of confidential documents, navigating the internal web, and other workflows that apply internally. Furthermore, software monocultures with similar configurations, such as those found in an enterprise, may simplify the task of making a single instance of BotSwindler operable across multiple hosts.

Within the enterprise environment, BotSwindler can run simulations on a user's system when it is idle (*e.g.,* during meetings, at night). Although virtual machines are common in enterprise environments, in cases where they are not used, they can be created on demand from a user's native environment. In Figure 6.13 we show one possible application of BotSwindler is in deployment as an enterprise service that runs simulations over exported copies of multiple users' disk images. In another approach, a user's machine state could be synchronized with the state of a BotSwindler enabled virtual machine [Cully *et al.*, 2008].

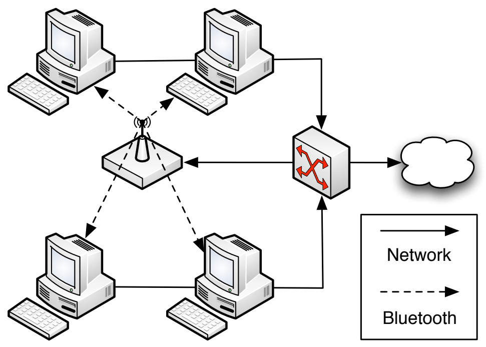


Figure 6.14: Personal workstation environment in which decoys are injected over the Bluetooth protocol from a nearby location.

As an example of an alternative approach, we demonstrated how BotSwindler could be extended to support the Bluetooth protocol in a personal workstation environment. Figure 6.14 shows a depiction of this architecture in which decoys are injected from a central location to several workstations in the nearby vicinity. The decoys are injected over the Bluetooth proxy Bluemaemo[10]. The primary challenge with this architecture lies in verifying the success and failure of simulated actions. Since this architecture does not rely on virtualization, verification must be performed in another manor. An alternative approach to performing the verification can be done using traffic analysis techniques to detect deviations as error conditions. In particular, network level introspection is performed by looking at the IPs of the traffic, the number of conversations, the number of exchanged request/response messages, and the number of bytes transferred in each message. Despite the fact that the traffic is encrypted, it turns out that there are differences that can be observed at the network level when failures occur. The obvious drawback of this approach is that it can only work on a subset of actions – those that induce network level observables. An analysis of the drawbacks of this approach as well as implementation details can be found in [Pappas *et al.*, 2010].

[10]Webstie: http://wiki.maemo.org/Bluemaemo

These applications of BotSwindler can enable the approach to tackle the problem of malware performing long-term corporate reconnaissance. For example, malware might attempt to steal credentials only after they have been repeatably used in the past. This elevates the utility of BotSwindler from a general malware detector to one capable of detecting targeted espionage software.

The application of BotSwindler to an enterprise would require adaptation for site-specific things (*e.g,* internal URLs), but use of specialized decoys does not preclude the use of general decoys like those detailed in Sect. 4.2. General decoys can help the organization identify compromised internal users that could be, in turn, the target of blackmail, either with traditional means or through advanced malware [Bond and Danezis, 2006].

6.7 Limitations and Open Problems

Our approach of detecting malware relies on the use of deception to trick malware to capture decoy credentials. As part of this work, we evaluated the believability of the simulations, but we did so in a limited way. In particular, our study measured the believability of short video clips containing different user workflows. These types of workflows are adequate for the detection of existing threats using short-term deception, but for certain use cases (such as the enterprise service) it is necessary to consider long-term deception, and the believability of simulation command sequences over extended periods of time. For example, adversaries conducting long-term reconnaissance on a system may be able to discover some invariant behavior of BotSwindler that can be used to distinguish real actions from simulated actions, and thus avoid detection. To counter this threat, more advanced modeling is needed to be able to emulate users over extended periods of time, as well as a study that considers the variability of actions over time. For long-term deception, the types of decoys used must also be considered. For example, some malware may only accept as legitimate those credentials that it has seen several times in the past. We can have "sticky" decoy credentials of course, but that negates one of their benefits (determining when a leak happened).

Malware may also be able to distinguish BotSwindler from ordinary users by attempting to generate bogus system events that cause erratic system behavior. These can potentially negatively impact a simulation and cause the simulator to respond in ways a real user would not. In this case, the malware may be able to distinguish between authentic credentials and our monitored decoys. Fortunately, erratic events that result in workflow deviations or simulation failure are also detectable by BotSwindler because they result in a state that cannot be verified by the VMV. When BotSwindler detects such events, it signals the host is possibly infected. The downside of this strategy is that it may result in false positives. As part our future work we will investigate how to measure and manage this threat using other approaches that ameliorate this weakness.

6.8 Host System Summary

BotSwindler is a bait injection system designed to delude and detect crimeware by forcing it to reveal itself during the exploitation of monitored decoy information. It relies on an out-of-host software agent to drive user-like interactions in a virtual machine aimed at convincing malware residing within the guest OS that it has captured legitimate credentials. As part of this work we have demonstrated BotSwindler's utility in detecting malware by means of monitored financial bait that is stolen by real crimeware found in the wild and exploited by the adversaries that control that crimeware. We have presented the results of experiments that show how BotSwindler's simulated workflows can be used to induce malware into observable network action, which can then be detected. In anticipation of malware seeking the ability to distinguish simulated actions from human actions, we designed our system to be difficult to detect by the underlying architecture and the believable actions it generates. To demonstrate the believability of the simulations, we conducted a Turing Test that showed we could succeed in convincing humans about 46% of the time. We assert that if attackers are forced to spend their time looking at the actions on each host it infects one by one to determine if they are real or not in order to steal information, BotSwindler would be a success; the crimeware's task does not scale.

Chapter 7

Educating Users and Measuring Organizational Security

In the previous chapters, we demonstrated systems that were intended to deceive attackers by convincing them something is real when in-fact it is not. These systems, when successful, expose attacks that might otherwise go unnoticed. In this chapter, we focus on reversing the task by creating decoys that mimic attackers' actions as a means of measuring organizational security and educating users. Unlike the previous chapters where the goal was to detect actions by malicious insiders or attackers, here we focus on the task of detecting innocent actions by insiders that may put an organization at risk.

Social attacks include those that occur when an attacker uses any of a variety social attack vectors that may range from email and telephone to in-person encounters. According to the 2010 Verizon Data Breach Investigations Report [Baker *et al.*, 2010], social attacks were used in 28% of the breaches for 2009 and nearly a quarter of these attacks occurred due to phishing. In these types of attacks, victims are sent spoofed emails that appear to be benign notifications from a bank, a social networking site, or a software upgrade. When victims take the bait, they are often greeted with some form of malicious software that attempts to install itself on victim's machine. Although there have been many technological advances that seem to hold promise in stopping these attacks, so far, none of them have proven 100% effective allowing the problem to continue. In fact, the vulnerability posed by phishing is often exploited by crimeware distributors such as those addressed in Chapter 6.

The defense approach we are advocating in this chapter involves better educating users to be cautious of suspicious emails. Although traditional training can be beneficial, it is often not enough. Our technique involves testing users' vulnerability using a variety of decoy emails; those that fall victim to our phony phishing attacks are informed so that they may learn and change their behavior later. Subsequent tests of the same users show that this method works, although sometimes it takes several iterations of testing and teaching.

In addition to training users, our technique provides a valuable metric that can be used by organizations to asses their own security and monitor for improvements. The field of computer and communications security begs for a foundational science to guide our designs of systems and to reveal the safety, security, and possible fragility of the complex systems we depend upon today [Stolfo *et al.*, 2011]. To achieve this goal we must devise suitable metrics that can be used to objectively compare and evaluate alternative designs and the security posture of the systems and organizations we have developed. For example, it is very important for Chief Security Officers and the top management of modern organizations, in business and government, to be given the tools they need to answer these fundamental questions: Is my organization secure? Are the personnel sufficiently educated and trained to minimize the risks to the organization? Is my organization complying with strict regulations on managing and safeguarding sensitive data? How do I measure the security risk of a new technology or service provided to our customers? These and many other related questions are often answered qualitatively, if at all, but rarely are hard measurements provided to objectively and scientifically answer these questions. It is especially important to answer these questions longitudinally over time, to understand whether the organizations security posture has improved through employee training or the introduction of new security technologies, policies and practices, or not. With formal metrics, for the first time it would be possible to measure the return on investment of any new security technology fielded. Furthermore, a solid metric may be applied as a means of assessing the strength of one organization relative to others.

The following subsections, we provide an overview of the system designed to create the decoy emails. We then present the results of two rounds of experiments conducted

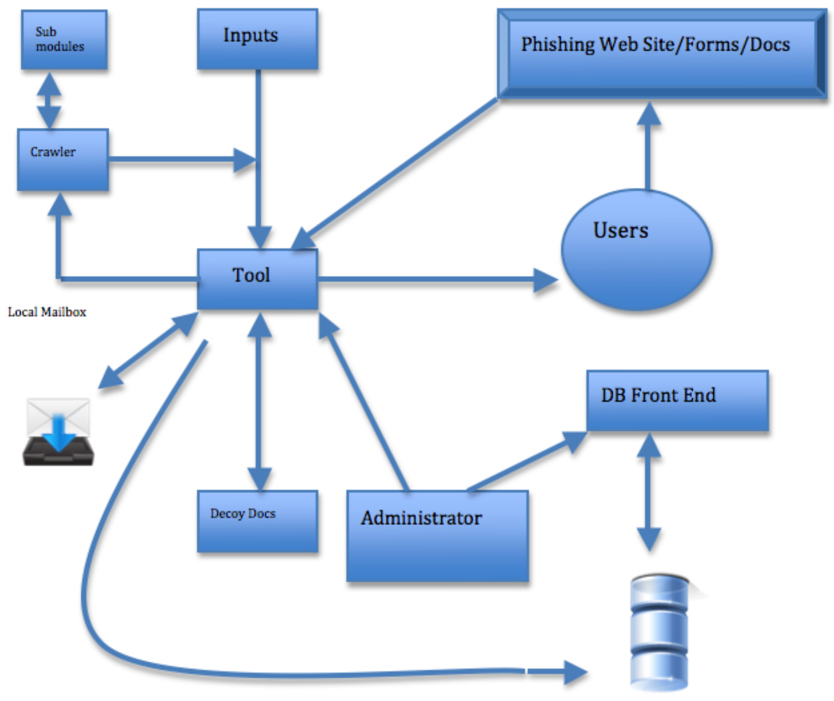


Figure 7.1: Components of the Phony Phish System.

at Columbia University in which approximately 4000 staff members and students were targeted for training with the phony phishing emails. The approved Columbia University IRB protocol is presented in Chapter 9.1.

7.1 Phony Phish System

The goal of the Phony Phish System is to provide an automatic means to generate and send benign phishing emails that can be used to measure an organization's security and educate users. The system consists of several components as shown in Figure 7.1.

Crawler Module: This component was designed to crawl a directory and obtain a list of target identities to perform the experiments on. For the experiments described in 7.2, this module was used to search the Columbia University directory, select users, their role within the university, and which department they belong to.

Email Generator: The email generator integrated all of the components and was used to deliver emails for the experiments. This component takes real email as input and performs processing on them to change names and using the Stanford Named Entity Recognition [1] engine. It also functions to anonymize user identify information through the use of unique hashes. For the generation of beacon'ed documents, the email generator relies on the Decoy Document Distributor introduced in chapter 4.

Web Application: A web application is used to collect user responses when they click on links and submit forms containing credentials. It tracks responses using a base 64 encoded query string that is attached to user requests. The system does not the store the identify of users, only the time at which the link was clicked, the department that the user belongs to, and the role of a user within the organization.

7.2 Experimental Analysis and Results

Experiments were conducted by sending 500 emails for each of four different types of decoy emails. Using standard statistical techniques [Krejcie and Morgan, 1970], this sample size was determined to be significant for measuring a single population parameter (*i.e.,* will a user open an email) with a 5% margin of error and 95% confidence for the approximately 70,000 IDs in the Columbia University directory. A second consideration for the choice of using 500 was for practical reasons. Our intent was to have a sample size large enough to draw scientifically significant conclusions without burdening an unnecessary number of subjects. Although we had permission from the university, the subjects were unwitting participants. The nature of this kind of experiment has the potential to cost users in both time and aggravation. Given that this was our first attempt at such an experiment, we decided we would start with 500 emails for each of the four types and adjust as necessary.

The decoy emails were modeled after various types of phishing attacks that occur in the wild. All of the emails were sent using an external email account from a popular webmail provider. Users that fell victim to the phony phishing emails were presented with the

[1] http://nlp.stanford.edu/ner/index.shtml

following message:

The Columbia University IDS Lab is conducting experiments designed to measure the security posture of large organizations and to educate users about safe practices so that they avoid falling prey to malicious emails. The emails automatically generated and sent to users of Columbias network and email system are designed to test whether users violate basic security policies. Although our emails are completely benign, please be aware that many emails are sent that are designed to trick unsuspecting users into giving up identity information.

The four different types of emails and their results are summarized below:

- **Email with internal URLs:** The content of these emails were from email received with an external source, but the URLs were changed to point to our IDS severs. The goal of these emails were to see how many users bothered to look at the address of the recipient before opening the email.
- **Email with external URLs:** The content of these emails was modified from emails received with an external source. The emails were designed to lure those interested in obtaining the Apple iPad. The URLs were changed to point to our external servers in the .info domain.
- **Forms to obtain credentials:** The content of these emails contained links to forms asking users for credentials to see how many users were willing to expose their credentials. Credentials were not stored.
- **Beacon Documents:** These emails contained PDF attachments that emitted a beacon to our servers when opened. The beacons were designed so that every user emitted a unique response enabling us to track them. An evaluation of the beacons is provided in Chapter 4.2.3.

Table 7.1 and Table 7.2 provide an overview of all of the results obtained from two rounds of experiments. Over the course of several weeks, offenders were repeatedly targeted until they stopped falling victim. The results between the two rounds of experiments were fairly consistent. The most important point that can be gleaned from the data:

Table 7.1: The number of responses for each round for the first experiment to measure the user response to Phony Phish.

Decoy Type	1^{st} **Round**	2^{nd} **Round**	3^{rd} **Round**	4^{th} **Round**
Email with internal URLs	52	2	0	NA
Email with external URLs	177	15	1	0
Forms to obtain credentials[2]	39/20	4/1	0	NA
Beacon Documents	45	0	NA	NA

Table 7.2: The number of responses for each round for the second experiment to measure the user response to Phony Phish.

Decoy Type	1^{st} **Round**	2^{nd} **Round**	3^{rd} **Round**	4^{th} **Round**
Email with internal URLs	69	7	1	0
Email with external URLs	176	10	3	0
Forms to obtain credentials	69/50	10/9	0	NA
Beacon Documents	71	2	0	NA

In all cases, users can be trained to be cautious of suspicious looking emails, but sometimes it takes several iterations of testing. In our experiments, the slowest learners took at most four iterations as shown in 7.2.

Some other observations are that it appears users are less likely to respond to emails that appear to be from internal sources, but have an external sender address. These emails were indeed suspicious because of this, but we do not have a good way to account for the differences in the content. For example, the external emails (row 2) appear pertain to the Apple iPad. At the time the emails were sent out, the emails would have been appealing to the masses. On the other hand, the internal emails (row 1) resembled those distributed by the university and are likely less appealing to the masses. Hence, there is insufficient data to make any conclusion concerning these differences.

The number of users that actually entered their credentials to the bogus forms seemed alarmingly high. We did not record the credentials and we did not validate them to ensure they were valid. However, we believe it is likely that at least some of the users entered valid credentials.

7.3 Metrics Conclusion

The previous sections provided an overview of our system designed to create phony phishing emails. We presented the results of two rounds of experiments conducted at Columbia University in which approximately 4000 staff members and students were targeted for training using the bogus phishing emails. The results presented in the previous section suggest that users can be trained using decoy technology to be cognizant of potential threats. Applying the same set of organizations laterally across multiple organizations can be a useful in measuring one organization's security posture relative to another's.

Part III

Conclusions

Chapter 8

Conclusions

This dissertation introduced many concepts, methods, and architectures aimed at trapping a wide variety of potential attackers with varying levels of sophistication. The systems presented in this dissertation introduced scalable and automated approaches toward a trap-based defensive system. They cause attackers to have to expend considerable effort to identify realistic useful information from bogus information that are intended to deceive. Naturally, the probability of exposing a malicious insider with trap-based defense tactics increases with the amount of decoy information that is generated and disseminated. As part of this dissertation, we have shown that network and host decoys can be used to detect attackers.

In summary, the main contributions of the dissertation include the following:

- **Generally Applicable Properties:** We have introduced a novel set of properties to guide in the design of decoy systems. These properties serve as goals for decoys and systems described in this dissertation.

- **Automatic Generation of Decoys:** A large-scale automated creation and management system for deploying decoys that can be used to detect malicious activity. This provides a means for ordinary users to deploy honey documents without having to setup sophisticated honeypot systems and sensors.

- **Decoy Networking:** We have demonstrated a system that shows the feasibility of automatically generating large amounts of believable network decoys. We presented results to show that this can be done without interfering with normal operations. We used human subjects to evaluate the believability of the generated decoys and showed that is difficult to distinguish from the real thing; our experienced judges achieved only 52% accuracy on average, nearly equivalent to random guessing. We provided a statistical analysis to show that the timing of the generated traffic is statistically similar to that of the real traffic. We demonstrated decoy efficacy against automated tools, designed to harvest and exploit credentials in mass by sniffing network transmissions. Moreover, we evaluated our system in a real wireless network that someone was monitoring and successfully detected eavesdropping and exploitation attempts.

- **Decoy Host System:** For the decoy host system, we created BotSwindler, a bait injection system designed to delude and detect crimeware causing it to reveal itself during the exploitation of monitored decoy information. It relies on an out-of-host software agent to drive user-like interactions in a virtual machine aimed at convincing malware residing within the guest OS that it has captured legitimate credentials. As part of this work we have demonstrated BotSwindler's utility in detecting malware by means of monitored financial bait that is stolen by real crimeware found in the wild and exploited by the adversaries that control that crimeware. We have presented the results of experiments that show how BotSwindler's simulated workflows can be used to induce malware into observable network action, which can then be detected. In anticipation of malware seeking the ability to distinguish simulated actions from human actions, we designed our system to be difficult to detect by the underlying architecture and the believable actions it generates. To demonstrate the believability of the simulations, we conducted a Turing Test that showed we could succeed in convincing humans about 46% of the time. We assert that if attackers are forced to spend their time looking at the actions on each host it infects one by one to determine if they are real or not in order to steal information, BotSwindler would be a success; the crimeware's task does not scale.

- **Security Metrics:** We introduced an expanded role for decoys to show that they can also be useful for educating users and measuring an organization's security posture.

> The field of computer and communications security begs for a foundational science to guide our designs of systems and to reveal the safety, security, and possible fragility of the complex systems we depend upon today. As we have shown, the use of decoys is a promising direction for gathering security metrics and training users on their innocent mistakes. We presented the results of two rounds of experiments conducted at Columbia University in which approximately 4000 staff members and students were targeted for training using the bogus phishing emails. The results presented in the previous section suggest that users can be trained using decoy technology to be cognizant of potential threats.

The use of decoy technology as a cyber defense strategy is relatively immature, but it is an area for which their are significant opportunities for defenders. Although the threats and adversaries may vary, in each context where a system is threatened, decoys can be used to deny critical information to adversaries making it harder for them to achieve their target goal.

Part IV

Appendices

Chapter 9

Experimental Details

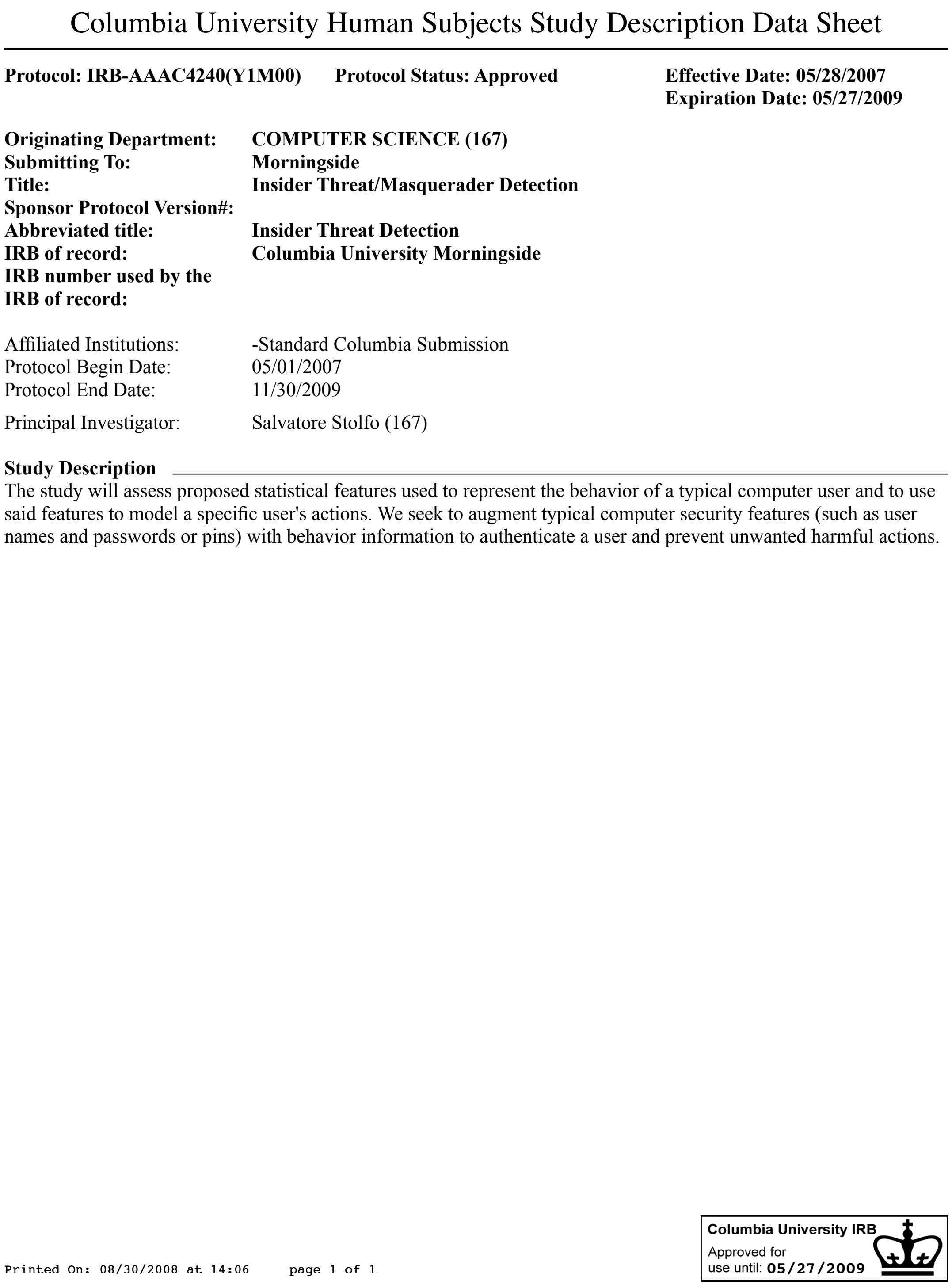
This Appendix includes the IRBs and miscellaneous code referenced in the previous chapters.

9.1 IRB Approvals

Research studies that rely on human subjects for testing require approval from the university institutional board (IRB). Figure 9.1 presents the approval from the IRB for the Turing Test studies in Chapters 5 and 6.3. Figure 9.2 presents the approval from the IRB for the user studies presented in Chapter 7.

9.2 BotSwindler Study Description

Figure 9.3 presents a screenshot of the webpage used for the study in Chapter 6.3. An abbreviated version of the JavaScript that was used to randomize the order of the links to videos is presented in Chapter 9.4.



Columbia University Human Subjects Study Description Data Sheet

Protocol: IRB-AAAC4240(Y1M00)	**Protocol Status: Approved**	**Effective Date: 05/28/2007** **Expiration Date: 05/27/2009**

Originating Department:	**COMPUTER SCIENCE (167)**
Submitting To:	**Morningside**
Title:	**Insider Threat/Masquerader Detection**
Sponsor Protocol Version#:	
Abbreviated title:	**Insider Threat Detection**
IRB of record:	**Columbia University Morningside**
IRB number used by the IRB of record:	
Affiliated Institutions:	-Standard Columbia Submission
Protocol Begin Date:	05/01/2007
Protocol End Date:	11/30/2009
Principal Investigator:	Salvatore Stolfo (167)

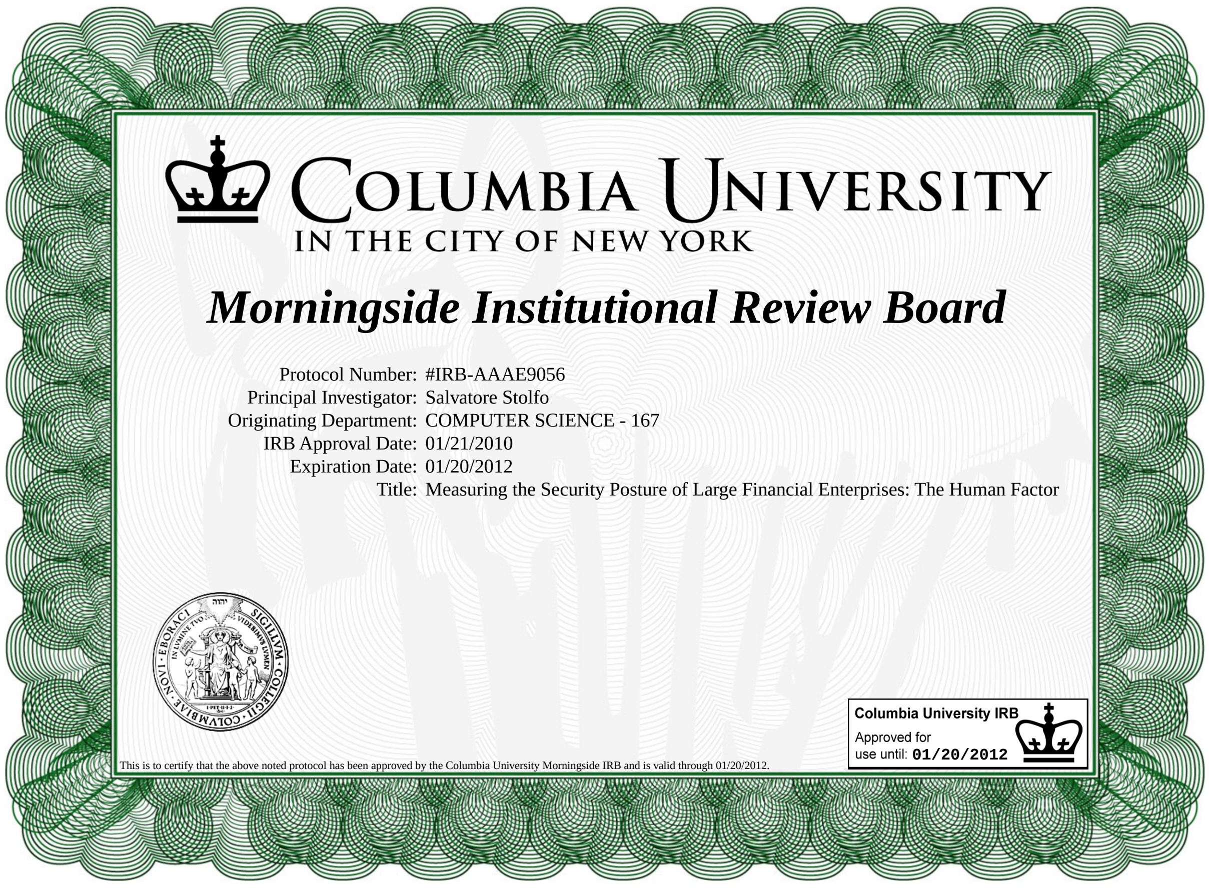
Study Description

The study will assess proposed statistical features used to represent the behavior of a typical computer user and to use said features to model a specific user's actions. We seek to augment typical computer security features (such as user names and passwords or pins) with behavior information to authenticate a user and prevent unwanted harmful actions.

Printed On: 08/30/2008 at 14:06 page 1 of 1

Columbia University IRB
Approved for
use until: 05/27/2009

Figure 9.1: Columbia University IRB for Turing Test user studies



COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

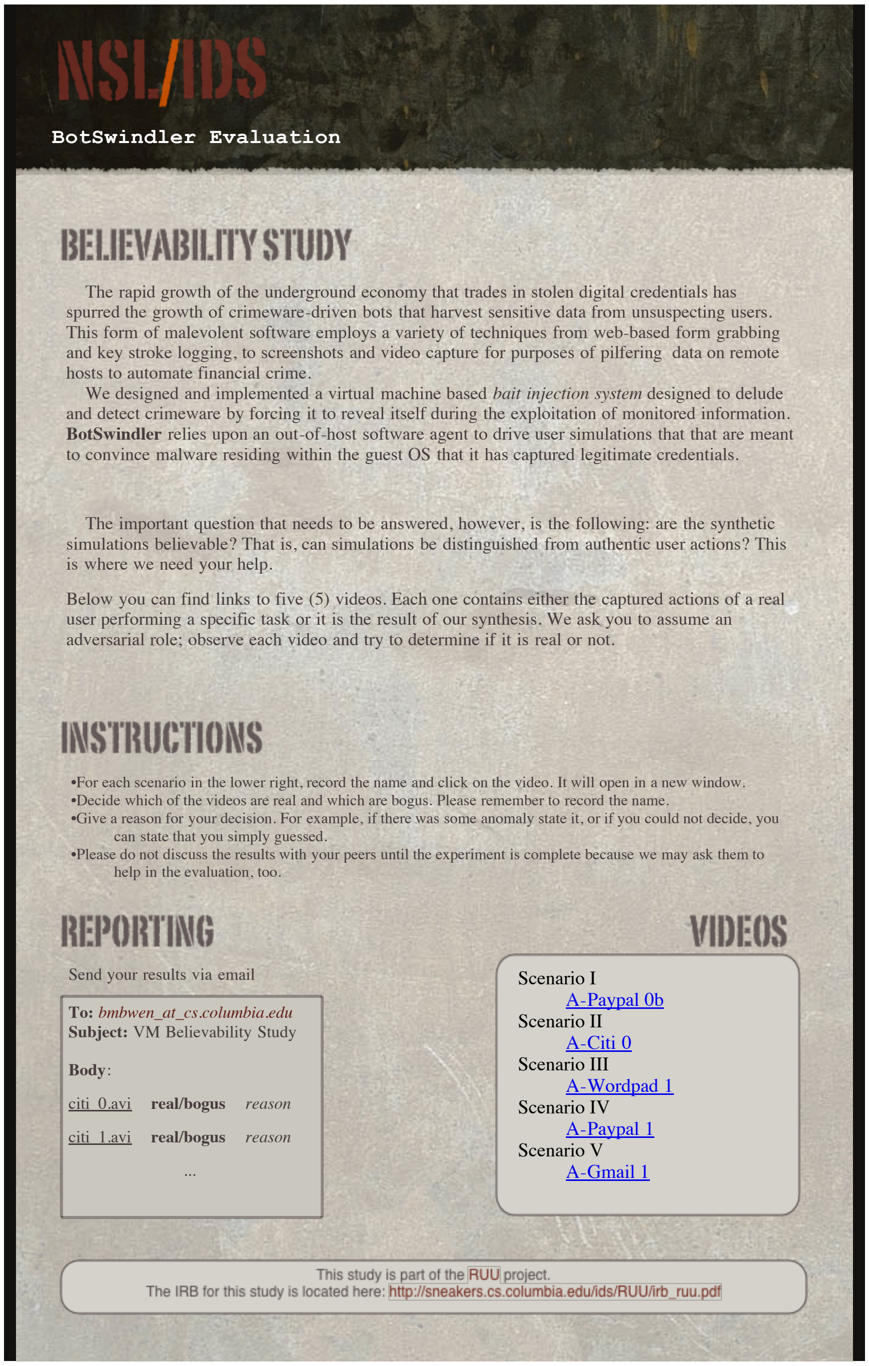
Morningside Institutional Review Board

Protocol Number:	#IRB-AAAE9056
Principal Investigator:	Salvatore Stolfo
Originating Department:	COMPUTER SCIENCE - 167
IRB Approval Date:	01/21/2010
Expiration Date:	01/20/2012
Title:	Measuring the Security Posture of Large Financial Enterprises: The Human Factor

This is to certify that the above noted protocol has been approved by the Columbia University Morningside IRB and is valid through 01/20/2012.

Columbia University IRB
Approved for
use until: 01/20/2012

Figure 9.2: Columbia University IRB for the metrics user study



NSL/IDS

BotSwindler Evaluation

BELIEVABILITY STUDY

The rapid growth of the underground economy that trades in stolen digital credentials has spurred the growth of crimeware-driven bots that harvest sensitive data from unsuspecting users. This form of malevolent software employs a variety of techniques from web-based form grabbing and key stroke logging, to screenshots and video capture for purposes of pilfering data on remote hosts to automate financial crime.

We designed and implemented a virtual machine based *bait injection system* designed to delude and detect crimeware by forcing it to reveal itself during the exploitation of monitored information. **BotSwindler** relies upon an out-of-host software agent to drive user simulations that that are meant to convince malware residing within the guest OS that it has captured legitimate credentials.

The important question that needs to be answered, however, is the following: are the synthetic simulations believable? That is, can simulations be distinguished from authentic user actions? This is where we need your help.

Below you can find links to five (5) videos. Each one contains either the captured actions of a real user performing a specific task or it is the result of our synthesis. We ask you to assume an adversarial role; observe each video and try to determine if it is real or not.

INSTRUCTIONS

- For each scenario in the lower right, record the name and click on the video. It will open in a new window.
- Decide which of the videos are real and which are bogus. Please remember to record the name.
- Give a reason for your decision. For example, if there was some anomaly state it, or if you could not decide, you can state that you simply guessed.
- Please do not discuss the results with your peers until the experiment is complete because we may ask them to help in the evaluation, too.

REPORTING

Send your results via email

To: *bmbwen_at_cs.columbia.edu*
Subject: VM Believability Study

Body:

citi_0.avi **real/bogus** *reason*

citi_1.avi **real/bogus** *reason*

...

VIDEOS

Scenario I
A-Paypal 0b
Scenario II
A-Citi 0
Scenario III
A-Wordpad 1
Scenario IV
A-Paypal 1
Scenario V
A-Gmail 1

This study is part of the RUU project.
The IRB for this study is located here: http://sneakers.cs.columbia.edu/ids/RUU/irb_ruu.pdf

Figure 9.3: The description of the user study given to each of the participants.

```
<script language="JavaScript">
var urls = new Array(5);
for (var i = 0; i < 5; i++) {
        urls[i] = new Array(2);
}
urls[0][0] = "<a href="path_to_citi_0.avi">A-Citi 0</a> "
urls[0][1] = "<a href="path_to_citi_1.avi">A-Citi 1</a>"
urls[1][0] = "<a href="path_to_gmail_0.avi">A-Gmail 0</a>"
urls[1][1] = "<a href="path_to_gmail_1.avi">A-Gmail 1</a>"
...
// Sort the arrays randomly
urls.sort(function() {return 0.5 - Math.random()})
for (var i = 0; i < 5; i++) {
        urls[i].sort(function() {return 0.5 - Math.random()});
}
</script>
<body><dl>
<dt>Scenario I</dt>
<dd><script type="text/javascript">document.write(urls[0][0]);
</script> </dd>
<dt>Scenario II</dt>
<dd><script type="text/javascript">document.write(urls[1][0]);
</script></dd>
...
</dl></body></html>
```

Figure 9.4: JavaScript to randomize the links to videos in the BotSwindler study.

Part V

Bibliography

Bibliography

[18 U.S.C §2511 P1, 2010] 18 U.S.C §2511 P1. Title 18, part 1, chapter 119, 2010. `http://www.law.cornell.edu/uscode/18/2510.html`.

[18 U.S.C §§3121-3127, 2010] 18 U.S.C §§3121-3127. Title 18, part 2, chapter 206, 2010. `http://www.law.cornell.edu/uscode/html/uscode18/usc_sup_01_18_10_II_20_206.html`.

[abu, 2009] abuse.ch zeus tracker, November 2009.

[Ahmed and Traore, 2007] Ahmed Awad E. Ahmed and Issa Traore. A New Biometric Technology Based on Mouse Dynamics. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 4(3):165–179, 2007.

[Akritidis *et al.*, 2007] Periklis Akritidis, W Y Chin, Vinh The Lam, Stelios Sidiroglou, and Kostas G Anagnostakis. Proximity breeds danger: Emerging threats in metro-area wireless networks. In *Proceedings of the 16th USENIX Security Symposium*, pages 323–338, August 2007.

[AntiSniff, 2009] AntiSniff. L0pht Heavy Industries, 2009. `http://packetstormsecurity.org/sniffers/antisniff/`.

[Araújo *et al.*, 2005] Lívia C. F. Araújo, Luiz H. R. Sucupira Jr., Miguel G. Lizárrage, Lee L. Ling, and João B. T. Yabu-Uti. User authentication through user authentication through typing biometrics features. *IEEE Transactions on Signal Processing*, 53 Issue:2:851–855, 2005.

[Baker *et al.*, 2010] Wade Baker, Mark Goudie, Alexander Hutton, C. David Hylender, Jelle Niemantsverdriet, Chistopher Novak, David Ostertag, Chistopher Porter, Mike Rosen, Bryan Sartin, and Peter Tippett. 2010 Data Breach Report. Technical report, Verizon Risk Team and the United States Secret Service, 2010.

[Beck and Tews, 2009] Martin Beck and Erik Tews. Practical attacks against wep and wpa. In *Proceedings of the 2nd ACM Conference on Wireless Network Security (WiSec)*, pages 79–86, March 2009.

[Bell and Whaley, 1982] J. Bell and B. Whaley. *Cheating and Deception.* Transaction Publishers, New Brunswick, NJ, 1982.

[Bellard, 2005] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proc. of USENIX Annual Technical Conference*, pages 41–46, Anaheim, CA, USA, April 2005.

[Bittau *et al.*, 2006] Andrea Bittau, Mark Handley, and Joshua Lackey. The final nail in wep's coffin. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 386–400, May 2006.

[Bond and Danezis, 2006] Mike Bond and George Danezis. A pact with the devil. In *Proc. of the New Security Paradigms Workshop (NSPW)*, pages 77–82, Dagstuhl, Germany, September 2006.

[Borders *et al.*, 2006] Kevin Borders, Xin Zhao, and Atul Prakash. Siren: Catching evasive malware. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 78–85, Oakland, CA, USA, May 2006.

[Chandrasekaran *et al.*, 2007] Madhusudhanan Chandrasekaran, S. Vidyaraman, and S. Upadhyaya. Spycon: Emulating user activities to detect evasive spyware. In *Proc. of the Performance, Computing, and Communications Conference (IPCCC)*, pages 502–509, New Orleans, LA, USA, May 2007.

[Chen and Noble, 2001] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proc. of the 8th Workshop on Hot Topics in Operating System (HotOS)*, pages 133–138, Washington, DC, USA, May 2001.

[Clark and Wilson, 1987] D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. pages 184–194, 1987.

[Core Security, 2010] Core Security. Core impact pro, 2010.

[Cracknell *et al.*, 2008] Phil Cracknell, Konstantin Gavrilenko, and Andrew Vladimirov. The wireless security survey of new york city. White paper 4th edition, RSA, The Security Division of EMC, 2008.

[Cully *et al.*, 2008] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 161–174, San Francisco, CA, USA, April 2008.

[Detristan *et al.*, 2003] T. Detristan, T. Ulenspiegel, M.S., and Von Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack 11, 61-9*, 2003.

[Dovrolis *et al.*, 2004] Constantinos Dovrolis, Parameswaran Ramanathan, and David Moore. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transactions on Networking (TON)*, 12(6):963–977, December 2004.

[Duda *et al.*, 2000] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification.* Wiley-Interscience, 2000.

[Early *et al.*, 2003] James P. Early, Carla E. Brodley, and Catherine Rosenberg. Behavioral authentication of server flows. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, page 4, 2003.

[Egele *et al.*, 2007] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proc. of the USENIX Annual Technical Conference*, pages 233–246, Santa Clara, CA, USA, June 2007.

[Garfinkel and Rosenblum, 2003] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. of Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, USA, February 2003.

[Gianvecchio and Wang, 2007] Steven Gianvecchio and Haining Wang. Detecting covert timing channels: an entropy-based approach. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 307–316, 2007.

[Graham, 2007] Robert Graham. Sidejacking with hamster. Technical report, Errata Security, 2007.

[Grundschober and Dacier, 1998] Stephane Grundschober and Marc Dacier. Design and implementation of a sniffer detector. In *Proceedings of the 1st International Workshop on the Recent Advances in Intrusion Detection (RAID)*, September 1998.

[Hall *et al.*, 2009] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[Higgins, 2009] Kelly Jackson Higgins. Up to 9 percent of machines in an enterprise are bot-infected, September 2009.

[Holz *et al.*, 2009] Thorsten Holz, Markus Engelberth, and Felix Freiling. *Learning More about the Underground Economy: A Case-Study of Keyloggers and Dropzones*, volume 5789 of *Lecture Notes in Computer Science (LNCS)*, pages 1–18. Springer Berlin / Heidelberg, September 2009.

[Hping,] Hping. Active Network Security Tool. `http://www.hping.org`.

[Ilett, 2005] Dan Ilett. Trojan attacks microsoft's anti-spyware, February 2005.

[Jay *et al.*, 2007] Caroline Jay, Mashhuda Glencross, and Roger Hubbol. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Transactions on Computer-Human Interaction*, 14 No. 2, 2007.

[Jiang and Wang, 2007] Xuxian Jiang and Xinyuan Wang. "Out-of-the-Box" monitoring of vm-based high-interaction honeypots. In *Proc. of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 198–218, Cambridge, MA, USA, September 2007.

[Jones *et al.*, 2006] Stephen. T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proc. of the USENIX Annual Technical Conference*, pages 1–14, Boston, MA, USA, March 2006.

[Katz and Lindell, 2007] John Katz and Yehuda Lindell. *Introduction to Modern Cryptography: Principles and Protocols.* Chapman & Hall/Crc Cryptography and Network Security Series, 2007.

[Killourhy and Maxion, 2009] Kevin S. Killourhy and Roy A. Maxion. Comparing Anomaly Detectors for Keystroke Dynamics. In 39^{th} *Annual International Conference on Dependable Systems and Networks (DSN)*, Los Alamitos, CA, USA, June-July 2009. IEEE Computer Society Press.

[Krejcie and Morgan, 1970] R. V. Krejcie and D. W. Morgan. Determining sample size for research activities. *Educational and psychological measurement*, 30:607–610, 1970.

[Krishnan *et al.*, 2010] Srinivas Krishnan, Kevin Snow, and Fabian Monrose. Trail of bytes: Efficient support for forensic analysis. In *Proceedings of the 17th ACM conference on Computer and Communications Security*, pages 50–60, Chicago, Illinois, USA, 2010.

[Kumaraguru *et al.*, 2007] Ponnurangam Kumaraguru, Yong Rhee, Alessandro Acquisti, Lorrie F. Cranor, Jason Hong, and Elizabeth Nunge. Protecting people from phishing: The design and evaluation of an embedded training email system. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '07)*, San Jose, California, 2007.

[Lee and Xiang, 2001] Wenke Lee and Dong Xiang. Information-Theoretic Measures for Anomaly Detection. In *IEEE Symposium on Security and Privacy (S&P)*, pages 130–143, Washington, DC, USA, 2001. IEEE Computer Society.

[Li *et al.*, 2007] Wei-jen Li, Salvatore Stolfo, Angelos Stavrou, Elli Androulaki, and Angelos D. Keromytis. A study of malcode-bearing documents. In *Proceedings of the* $4^t h$ *international conference on Detection of Intrusions and Malware, and Vulnerability Assessment DIMVA)*, pages 231–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[Li *et al.*, 2008] Mingzhe Li, Mark Claypool, and Robert Kinicki. Wbest: a bandwidth estimation tool for ieee 802.11 wireless networks. In *Proceedings of the 33rd IEEE Conference on Local Computer Networks (LCN)*, pages 374–381, October 2008.

[Matwyshyn *et al.*, 2010] Andrea Matwyshyn, Ang Cui, Angelos D. Keromytis, and S. J. Stolfo. Ethics in security vulnerability research. *IEEE Security and Privacy, Basic Training (R. Ford and D. Frincke, Eds.)*, 2010.

[McGlasson, 2007] Linda McGlasson. Tjx update: Breach worse than reported. Article, Bank Info Security, 2007.

[McRae and Vaughn, 2007] Craig M. McRae and Rayford B. Vaughn. Phighting the phisher: Using web bugs and honeytokens to investigate the source of phishing attacks. In *Proceedings of the* 40th *Hawaii International Conference on System Sciences (HICSS)*, pages 270c – 270c, Washington, DC, USA, 2007. IEEE Computer Society.

[Medina *et al.*, 2002] Alberto Medina, Nina Taft, Kav Salamatian, Supratik Bhattacharyya, and Christophe Diot. Traffic matrix estimation: Existing techniques and new directions. *ACM SIGCOMM Computer Communication Review*, 32(4):161–174, October 2002.

[Messmer, 2009] Ellen Messmer. America's 10 most wanted botnets, July 2009.

[Mini router, 2009] Mini router. Open-Mesh, 2009. `http://www.open-mesh.com`.

[Monrose and Rubin, 1997] Fabian Monrose and Aviel Rubin. Authentication via Keystroke Dynamics. In 4th *ACM Conference on Computer and Communications Security (CCS)*. ACM, April 1997.

[Morse, 2009] Andrew Morse. Google's gmail service suffers another shutdown glitch. Article, Wall Street Journal, 2009.

[New York State Office of Cyber Security & Critical Infrastructure Coordination, 2005] New York State Office of Cyber Security & Critical Infrastructure Coordination. Gone phishing. A Briefing on the Anti-Phishing Exercise Initiative for New York State Government. Aggregate Exercise Results for public release., 2005.

[nuttcp, 2010] nuttcp, 2010. `ftp://ftp.lcp.nrl.navy.mil/u/bill/beta/nuttcp/`.

[Ohm *et al.*, 2007] Paul Ohm, Douglas C. Sicker, and Dirk Grunwalk. Legal issues surrounding monitoring during network research. In *Proceedings of the* 7th *ACM SIGCOMM conference on Internet measurement*, pages 141–148, San Diego, CA, USA, 2007. ACM.

[OpenWRT, 2009] OpenWRT. OpenWRT, 2009. `http://www.openwrt.org`.

[Oudot, 2004] Laurent Oudot. Wireless honeypot countermeasures. Technical report, SecurityFocus, 2004.

[Pappas *et al.*, 2010] Vasilis Pappas, Brian M. Bowen, and Angelos D. Keromytis. Short paper: Crimeware swindling without virtual machines. In *Proceedings of the 13th Information Security Conference (ISC)*, Boca Ratan, FL, USA, 2010. Springer.

[Payne *et al.*, 2007] Bryan D. Payne, Martim Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[Peng *et al.*, 2006] Pai Peng, Peng Ning, and Douglas S. Reeves. On the secrecy of timing-based active watermarking trace-back techniques. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[Pereira, 2007] Joseph Pereira. How credit-card data went out wireless door. Article, Wall Street Journal, 2007.

[Phishme.com, 2011] Phishme.com. Phishme.com, 2011.

[Richardson, 2009] R. Richardson. Csi computer crime and security survey. Technical report, CERT, 2009.

[Scapy,] Scapy. `http://www.secdev.org/projects/scapy/`.

[Shneiderman, 1984] Ben Shneiderman. Response time and display rate in human performance with computers, September 1984.

[Smith, 2000] R. M. Smith. Microsoft word documents that phone home. 2000.

[Sommers and Barford, 2004] Joel Sommers and Paul Barford. Self-configuring network traffic generation. In *Proceedings of the 4th ACM SIGCOMM Internet Measurement Conference (IMC)*, pages 68–81, October 2004.

[Song *et al.*, 2007] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proc. of the 14th ACM conference on Computer and Communications Security (CCS)*, pages 541–551, Alexandria, VA, USA, 2007.

[Song, 2011] Yingbo Song. Network traffic-behavior modeling, synthesis, and anonymization. Thesis Proposal, 2011.

[Spitzner, 2003a] Lance Spitzner. Honeypots: Catching the insider threat. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, pages 170–179, December 2003.

[Spitzner, 2003b] Lance Spitzner. Honeytokens: The other honeypot. Technical report, SecurityFocus, 2003.

[Srivastava and Giffin, 2008] Abhinav Srivastava and Jonathon Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Proc. of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 35–58, Cambridge, MA, USA, September 2008.

[Stolfo *et al.*, 2011] Sal Stolfo, Steven Bellovin, and David Evans. Measuring security. *IEEE Security & Privacy Magazine*, pages 72–77, 2011.

[Stoll, 1988] Clifford Stoll. Stalking the wily hacker. *Communications of the ACM*, 31(5):484, May 1988.

[Sthlberg, 2007] Mika Sthlberg. The trojan money spinner. Technical report, F-Secure Corporation, September 2007.

[Symantec, 2008] Symantec. Trends for july - december '07. White paper, April 2008.

[Tcpreplay, 2009] Tcpreplay, 2009. `http://tcpreplay.synfin.net/trac/`.

[The Honeynet Project, 2003] The Honeynet Project. Know your enemy: Sebek, a kernel based data capture tool. Technical report, 2003.

[The Honeynet Project, 2010] The Honeynet Project, 2010. `http://www.honeynet.org`.

[the madwifi project, 2009] the madwifi project, 2009. `http://madwifi-project.org`.

[top, 2009] Researcher uncovers massive, sophisticated trojan targeting top businesses, July 2009.

[Tsow *et al.*, 2006] Alex Tsow, Markus Jakobsson, Liu Yang, and Susanne Wetzel. Warkitting: the drive-by subversion of wireless home routers. *Journal of Digital Forensic Practice*, 1(3):179–192, 2006.

[Turing, 1950] Alan Mathison Turing. Computing machinery and intelligence. *Mind, New Series*, 59(236):433–460, October 1950.

[Vahdat *et al.*, 2002] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36:271–284, December 2002.

[Vishwanath and Vahdat, 2009] Kashi Venkatesh Vishwanath and Amin Vahdat. Swing: Realistic and responsive network traffic generation. *IEEE/ACM Transactions on Networking (TON)*, 17(3):712–725, June 2009.

[Wall of Sheep, 2009] Wall of Sheep, 2009. `http://www.wallofsheep.com/`.

[Willems *et al.*, 2007] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, pages 32–39, Oakland, CA, USA, March 2007.

[xvf, 2009] Xvfb(1), November 2009.

[Yin *et al.*, 2007] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panaroma: Capturing system-wide information flow for malware detection and analysis. In *Proc. of the 14th ACM conference on Computer and Communications Security (CCS)*, pages 116–127, Alexandria, VA, USA, 2007.

[Yuill *et al.*, 2004] Jim Yuill, Mike Zappe, Dorothy Denning, and Fred Feer. Honeyfiles: Deceptive files for intrusion detection. In *Proceedings of the 5th Annual IEEE SMC Information Assurance Workshop (IAW)*, pages 116–122, June 2004.

[Yuill *et al.*, 2006] J. Yuill, D. Denning, and F. Feer. Using deception to hide things from hackers : Processes, principles, and techniques. *Journal of Information Warfare*, 5(3):26–40, November 2006.

[Zalewski, 2006] Michal Zalewski. [the new p0f], 2006. `http://lcamtuf.coredump.cx/p0f.shtml`.

[zeu, 2009] Measuring the in-the-wild effectiveness of antivirus against zeus. Technical report, Trusteer, September 2009.