A System for Generating and Injecting Indistinguishable Network Decoys

Brian M. Bowen, Vasileios P. Kemerlis, Pratap Prabhu,
Angelos D. Keromytis, Salvatore J. Stolfo

Computer Science Department
Columbia University

Abstract

We propose a novel trap-based architecture for detecting passive, "silent", attackers who are eavesdropping on enterprise networks. Motivated by the increasing number of incidents where attackers sniff the local network for interesting information, such as credit card numbers, account credentials, and passwords, we introduce a methodology for building a trap-based network that is designed to maximize the realism of bait-laced traffic. Our proposal relies on a "record, modify, replay" paradigm that can be easily adapted to different networked environments. The primary contributions of our architecture are the ease of automatically injecting large amounts of believable bait, and the integration of different 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. Our network traps consist of several types of monitored passwords, authentication cookies, credit cards, and documents containing beacons to alarm when opened. The efficacy of our decoys against a model attack program is also discussed, along with results obtained from experiments in the field. In addition, we present a user study that demonstrates the believability of our decoy traffic, and finally, we provide experimental results to show that our solution causes only negligible interference to ordinary users.

1 Introduction

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 exfiltrated. Secure protocols such as Wi-Fi 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 (APs) in New York City (NYC), and 48% in London, used advanced security [8]. To make things worse, only 24% of the total APs in NYC, and 19% in London, used a WPA2 variant.

In general, there is little that can be done to detect passive eavesdropping on networks, and the problem is only exacerbated with Wi-Fi due to the range of signals and the absence of physical access barriers. Some techniques that have been applied to wired networks for detecting snoopers, although unreliably, are based on DNS behavior, or network and machine latency [2]. 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 have been the focus of previous efforts. We broadly target two types of attackers:

1. *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 and wireless hotspots), where the only barriers to separate the outside are firewalls or some form of physical security.

2. Those that successfully *infiltrate* the network through attacks at the protocol level [3, 4], password guessing, router hijacking [1, 30], or some vulnerability in Wi-Fi security. As a concrete example, consider the case of the massive credit card heist that occurred at TJX [22], in which attackers exploited the vulnerable WEP protocol to gain internal network access. Once inside, they eavesdropped undetected, acquired additional credentials, and eventually stole over 45 million credit cards [15].

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 accessing a decoy account using sniffed credentials. Our methodology for building a trap-based network 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. While the use of decoys is not a new concept, 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 Wi-Fi networks. However, the methodology is broadly applicable and can be adopted to conventional wired infrastructures. 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 Section 6. Demonstrating decoy efficacy and accuracy against snoopers requires an indeterminate amount of time. In Section 4, 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 [31]. The results presented in Section 5 testify to decoy realism.

The rest of this paper is organized as follows. Related work is introduced in Section 2, while the architectural design and prototype implementation of our system is discussed in Section 3. Legal considerations regarding our methodology are presented in Section 7, and conclusions are in Section 8.

2 Related Work

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 (e.g., load balancing, routing protocols configuration) [16], network simulation and emulation [32], and many more. To support these applications, many software tools have been created, ranging from customizable packet generators, such as Hping [13] and Scapy [23], to large-scale network emulators like ModelNet [32]. Other tools, including Swing [33], focus solely on traffic generation, but with the end goal of realistic TCP/IP or UDP values and statistically accurate timing measures. Similarly, Harpoon [24] 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.

The use of deception, or decoys, plays a valuable role in the protection of systems, networks, and information. The first use of decoys (in the cyber domain) has been credited to Cliff Stoll [27], while trying to catch German hackers breaking into the Lawrence Berkeley National Laboratory (LBNL). His methods included the use of bogus networks, systems, and documents for gathering intelligence on attackers. 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 as to know if anyone accessed them. Eventually, a German hacker was caught, and he was found selling secrets to the KGB.

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 for gathering intelligence to understand 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 [25], 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" [26], citing examples that

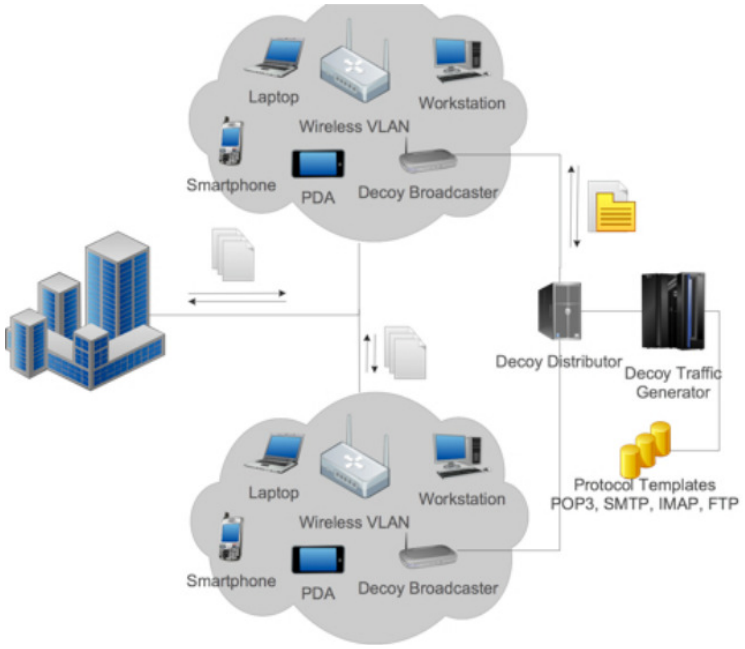

Figure 1: Injection Platform.

include bogus medical records, credit card numbers, and credentials, with descriptions of how they can be used to detect malicious insiders. Oudot [21] 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 [12] 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 application-layer detection are discussed in Section 3.3.

Currently, the decoy/honeytoken creation is a laborious and manual process requiring large amounts of administrative 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*. The *indistinguishability* of our honeyflows, the volume at which they can be produced, and the non-interference with real flows make our work novel.

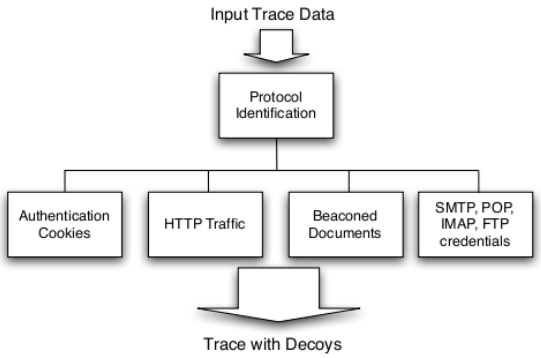

Figure 2: Honeyflow creation process.

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

Figure 3: SMTP Identifiers.

3 Design and 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 deceive the human viewer. In previous work [5], we formally defined a core set of properties, including *believability, non-interference*[1]*, detectability, variability,* and *enticement*, for formally guiding 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. Additionally, 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 of our system, shown in Figure 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 of decoys. The implementation details of the aforementioned components are discussed in the following subsections.

3.1 Automated Decoy Traffic Generator

The Decoy Traffic Generator uses the software API that we developed to produce honeyflows through a multi-step process, as illustrated in Figure 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. Note that we have specifically designed

[1]Note that throughout the paper the use of this term differs from the standard use first introduced by Goguen and Meseguer [10].

Table 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%

the API to handle both types of input due to the ethical and legal issues concerning the recording of network traffic (see Section 7). 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 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 the basis for decoy traffic within which decoys can be added.

Once the API obtains an input trace, a new trace with decoy information is automatically generated 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 that are both simple literal strings or complex regular expressions. For example, Figure 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 1, rather than on 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 the protocol determination does not succeed, the trace is marked as *unknown* and the API proceeds to step 6. We note that success in correctly identifying protocols largely depends on the rules that are manually created.

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, as well as protocol payloads (i.e., email bodies, web page contents). 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 3.3). Moreover, the API can also be used to introduce bait HTTP flows that contain monitored URLs, or even 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 the 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. We note that MACs are generated by combining three octets that correspond to those belonging to common vendors along with three random octets.

6. 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 [24], can also be used for enabling the creation of honeyflows that are statistically similar to normal traffic.

7. OS fingerprint models of p0f [35] 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 finally combined into a single trace, which is then broadcasted to the environment.

3.2 Decoy Broadcaster

The Decoy Broadcaster is the architectural component of our system that is responsible for spreading the bait content inside a network segment. It comprises both hardware and software entities. Figure 1 illustrates a decoy broadcaster inside the context of our 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 an AP, so as to maximize the coverage of the replayed traffic[2]. 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* [3] operation by its wireless network interface card (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, either it 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 are 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 [17], 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 [20]—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 [29] driver that supports a wide set of features such as:

- different operation modes: Station, Master, Ad-Hoc, and so on, including the Monitor mode
- 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

[2] Mind you that an active attacker might try to identify bait traffic by communicating directly with a decoy broadcaster for testing whether it is real or not. However, the act of attempting communication reveals the attacker.

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

- 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[4]. To inject the honeyflows we ported Tcpreplay [28], a suite for replaying previously captured traffic for network testing purposes. The typical injection workflow is specified as follows:

1. A new VAP is created in the Decoy Broadcaster and set in Monitor mode.
2. The bait traffic is uploaded into the Decoy Broadcaster [5].
3. Tcpreplay injects the decoy traffic into 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 3.3) as one type of decoy. Since these are valid for only a finite amount of time, they need to be routinely regenerated. More importantly, however, is that decoy traffic must be frequently 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 Generator (see Figure 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.

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

[5] 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.

3.3 Trap-based Decoys

Our trap-based decoys have the inherent property of being *detectable* on their own, so they do not depend on host, or network, monitoring. A benefit of being self-detectable is that the system does not suffer the characteristic performance burden of decoys that do require additional monitoring. This form of decoy is made up of "bait" information, such as online banking logins provided by a collaborating financial institution [6], credit card numbers, login accounts for online servers, and web-based email accounts. The primary requirement for bait is to be detectable when (mis)used. One form of bait that we use are Gmail account credentials, including usernames, passwords, and authentication cookies. In this case, custom scripts access `mail.google.com` and parse the bait account pages to gather account activity information. In case of credit card numbers, providers such as PayPal offer APIs that we begun to use for monitoring their activity. Alternatively, agreements with other financial institutions allow us to be notified when decoy credit card numbers are used. We note that obtaining such agreements can be challenging since financial institutions may not have any incentive to help.

In this work, we make particular use of a certain 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. 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 that 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 additional deceptive layer that is embedded within the application layer of some network protocols (e.g., email attachments and file uploads). Using techniques common to malware, 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 identify the decoy document and its association to the network location of the host accessing it. 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.

The ability to create vast quantities of variable decoys is important. Ideally, one would want to inject unique decoys for different periods of time and for different physical locations. Doing so would allow one to have some idea when and where eavesdropping occurred.

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

4 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.

4.1 Defcon Experiment

Defcon's yearly meeting includes the infamous *wall of sheep* [34], 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 (i.e., the third entry in Figure 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, which were passively monitoring the wireless channel during the conference, from being malicious.

This experiment provides evidence that our system can successfully 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), 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.

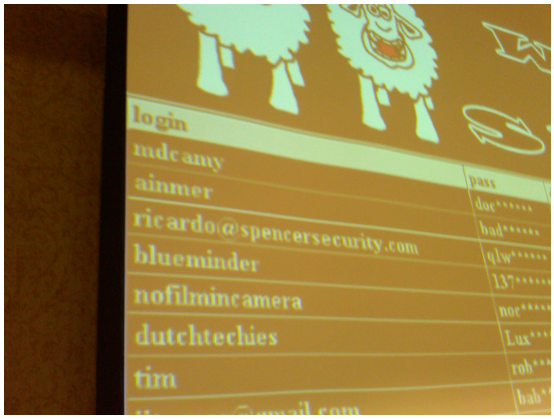

Figure 4: Defcon '09 Wall of Sheep.

4.2 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 [22]. In particular, we model attackers that attempt to harvest login cookies in mass (also known as SideJacking [11]), 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 [18]), 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. 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.

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

Table 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

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. As an additional test of the cookies, we manually sniffed the network, extracted the cookies, and put them into a browser session using a cookie editor. This ensured there was no bias in our code. 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.

Table 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 conducted on decoys before the bait injection system was constructed, to measure the 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. Combining this information with other traps will likely reveal a snooper with high accuracy [7].

5 Believability of Bogus Traffic: A Decoy Turing Test

Alan Turing proposed a method to demonstrate "artificial intelligence" through the failure of human judges to distinguish between human and machine conversational simulators [31]. 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

[7] We are beginning to run the bait injection system and will report in the final version any events we may capture in the wild.

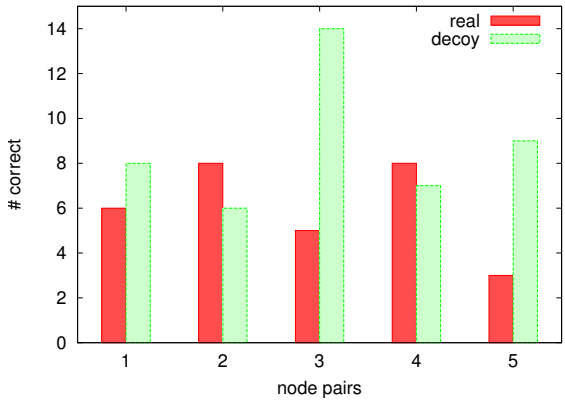

Figure 5: DTT Results: Real vs. Decoy.

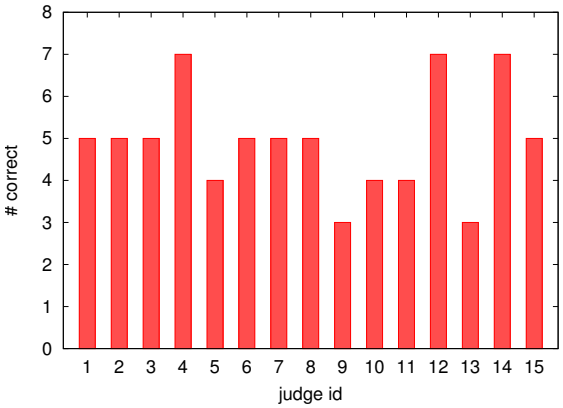

Figure 6: DTT Results: Users' Correctness.

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

In our experiment, human judges were solicited and selected based on their prior knowledge of networking protocols and experience in examining network traces. Our final pool of 15 judges consisted of PhD's and graduate students in the network security field, a staff member from the computing research facility of our department, and a security professional from an antivirus company. 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 Section 3, but with slight modifications to enable a structured study. We constructed our test data set including traffic from only 10 hosts, assuming the judges would have limited patience 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 2, for producing 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 spend at least 15 minutes in their analysis and they were permitted to use any automated or analysis tool to aid in making the decision.

5.1 Results and Discussion

Figure 5 summarizes the results for each of the 10 hosts. The hosts are arranged in pairs, in which the right bars correspond to decoys, and the left bars correspond to the authentic traffic. The height of the bars reflects the number of judges that correctly decided whether a given host was real or decoy. Although these results alone suggest that judges were able to discern decoys more regularly than authentic hosts, as shown by the height of the bars on the right, it is important to take into consideration the judges' overall correctness. Figure 6 shows the overall correctness for each of the fifteen judges. Overall, the judges were 49.9% correct, on average, suggesting that we have achieved the goal of indistinguishable decoys. After interviewing the judges, we concluded that the bias for decoys in Figure 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.

Although it is not immediately clear from the figures, one of the judges successfully identified an initial deficiency in the decoys that allowed him to positively distinguish decoys. This judge achieved 7 out of 10 correct in the DTT by examining the manufacturer of the NICs. The judge observed 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 the traffic was bogus or not. We have since fixed this problem by using more common vendors for our fake MACs, but this incident does speak to the challenge of getting bogus traffic to look real, especially in the eyes of highly knowledgeable judges. 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 there were actually 3 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, and hence, this judge was simply lucky. 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.

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 frames, control frames, beacons). 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, and try to correlate traffic, in order to distinguish real traffic from decoys. This scenario was outside the scope of our DTT. We plan to address this in future work, 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.

6 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 and corrupt authentic data, or confuse 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. Injected packets could potentially interfere with legitimate network services. However, this risk is mitigated because the packets are not addressed to legitimate hosts and they are not forwarded by the AP. We also have to ensure that we not interfere with existing security infrastructure such as network monitors. In our case, the Wi-Fi channels were not being monitored, so this is not an issue. However, if deployed on a wired network, this could be a concern. In particular, care would have to be taken to ensure that injected honeyflows do not trigger false alerts.

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 the 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, which demonstrate that our approach causes minimal interference to ordinary users.

6.1 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 Appendices A and B. The two embedded devices (i.e., Hermes and Hades) that we utilized in our experiments run our customized OpenWRT image (kernel v2.6.26), whereas everything else runs GNU/Linux (kernel v2.6.24). 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 [8].

[8] 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.

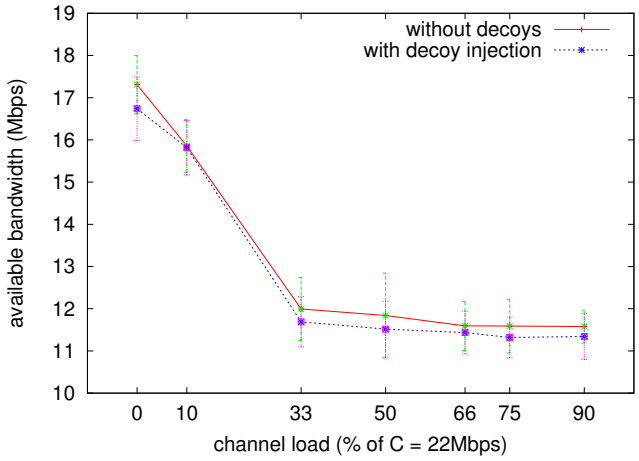

Figure 7: Interference cost.

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 overestimating 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 7. 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

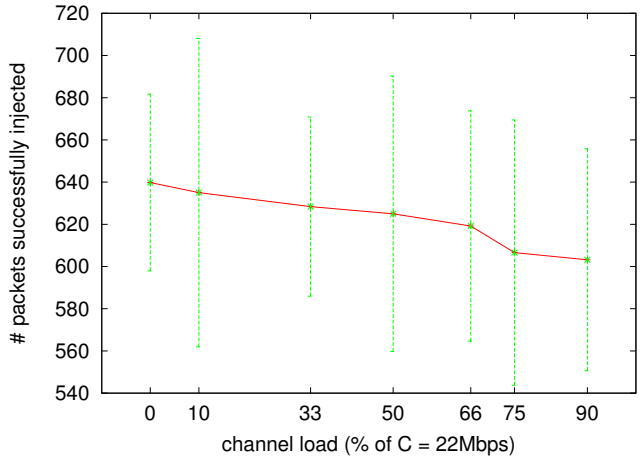

Figure 8: Packets successfully injected.

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 the traffic by *maintaining* the corresponding timing information. Replaying at higher rates will not give us any benefit as Figure 8 suggests: number of packets that were injected by Hades and successfully intercepted by Zeus. We used one VAP (see Section 3.2) 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.

Our experimental testbed, though simple, indicates 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, can 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, and allow us to further investigate larger-scale deployments.

7 Risks and 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[9] limit access to network data. The Wiretap statute [6] regulates the collection of electronic communications contents, and in general, it prohibits third parties from intercepting and recording traffic. The Pen/Trap statute is similar [7], but regulates the recording of addressing 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.

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 he may not have been able to otherwise.

8 Summary

Decoy trap-based security defenses, and deception in general, are powerful tools against a wide range of threats in wireless environments. We have demonstrated a system that shows the feasibility of automatically generating large amounts of believable decoy information, without interfering with normal operations. We used human subjects to evaluate the believability of the generated decoys, and showed that is difficult to distinguish them from the real thing; our experienced judges achieved only 49.9% accuracy on average, which is equivalent to random guessing. We also demonstrated decoy efficacy against automated tools that are designed to harvest and exploit credentials in mass by sniffing network transmissions. Finally, 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 and try to correlate traffic, or those that may use additional sources (like an administrator) to discern decoys without testing them.

Acknowledgements

This work was supported in part by the National Science Foundation through Grant CNS-09-14312 and by ONR MURI N00014-07-1-0907. Any opinions, findings, and conclusions or recommenda-

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

tions expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or ONR.

References

[1] P. Akritidis, W. Y. Chin, V. T. Lam, S. Sidiroglou, and K. 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.

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

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

[4] A. Bittau, M. Handley, and J. Lackey. The final nail in wep's coffin. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 386–400, May 2006.

[5] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside attackers using decoy documents. In *Proceedings of the 5th International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 51–70, September 2009.

[6] U. S. Code. Title 18, part 1, chapter 119. `http://www.law.cornell.edu/uscode/18/2510.html`.

[7] U. S. Code. Title 18, part 2, chapter 206. `http://www.law.cornell.edu/uscode/18/3121.html`.

[8] P. Cracknell, K. Gavrilenko, and A. Vladimirov. The wireless security survey of new york city. White paper 4th edition, RSA, The Security Division of EMC, 2008.

[9] C. Dovrolis, P. Ramanathan, and D. Moore. Packet-dispersion techniques and a capacity-estimation methodology. *IEEE/ACM Transactions on Networking (TON)*, 12(6):963–977, December 2004.

[10] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 3rd IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.

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

[12] S. Grundschober and M. 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.

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

[14] M. Li, M. Claypool, and R. 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.

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

[16] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic matrix estimation: Existing techniques and new directions. *ACM SIGCOMM Computer Communication Review*, 32(4):161–174, October 2002.

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

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

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

[20] OpenWRT. `http://www.openwrt.org`.

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

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

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

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

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

[26] L. Spitzner. Honeytokens: The other honeypot. Technical report, SecurityFocus, 2003.

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

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

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

[30] A. Tsow, M. Jakobsson, L. Yang, and S. Wetzel. Warkitting: the drive-by subversion of wireless home routers. *Journal of Digital Forensic Practice*, 1(3):179–192, 2006.

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

[32] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36:271–284, December 2002.

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

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

[35] M. Zalewski. [the new p0f]. `http://lcamtuf.coredump.cx/p0f.shtml`.

A 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 3.2). 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.

B Bandwidth estimation

The capacity of Hermes (i.e., Hermes Wi-Fi network) is approximated using pathrate [9]. Different channel loads where emulated using the invasive nuttcp [19] tool, whereas the available bandwidth of Poseidon was estimated using wbest [14]. 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 that a path can provide to an application, given the path's current cross traffic load.