Automating the Injection of Believable Decoys to Detect Snooping

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

Department of Computer Science
Columbia University
New York, NY, USA
{bb2281, vk2209, pvp2105, ak2052, sjs11}@columbia.edu

ABSTRACT

We propose a novel trap-based architecture for enterprise networks that detects "silent" attackers who are eavesdropping network traffic. The primary contributions of our work are 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. Finally, we present results of a user study that demonstrates the believability of our automatically generated decoy traffic.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*invasive software, unauthorized access*; K.6.m [**Management of Computing and Information Systems**]: Miscellaneous—*security*

General Terms

Design, Measurement, Security

Keywords

Decoys, Honeyflow, Honeytoken, Traffic generation, Trap-based defense, Deception

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 absconded. 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 the corporate access points in New York City (NYC) and 48% in London used advanced security [6]. To make things worse, only 24% and 19% of the NYC and London total APs respectively, 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 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 and wireless hotspots), where the only barriers to separate the outside are firewalls or some form of physical security.
- Those that successfully infiltrate the network through attacks at the protocol level [3, 4], password guessing, router hijacking [1, 16], or some vulnerability in Wi-Fi security. As a concrete example, consider the case of the massive credit card heist that occurred at TJX [11] 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 [8].

WiSec'10, March 22–24, 2010, Hoboken, New Jersey, USA.

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.

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 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 that by conducting a user study analogous to the Turing Test [17]; results are presented in Section 5 that testify to decoy realism.

2. RELATED WORK

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 gathering intelligence for understanding 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, which he defines as "a honeypot that is not a computer" [13], 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 [10] gave a simple example of how honeypots can be used on wireless networks, but in his case, all of the sessions are the same, making them trivial to avoid. Grundschober [7] 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 those sessions. 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 3.3.

Currently, the decoy/honeytoken creation is a laborious and manual process requiring large amounts of 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.

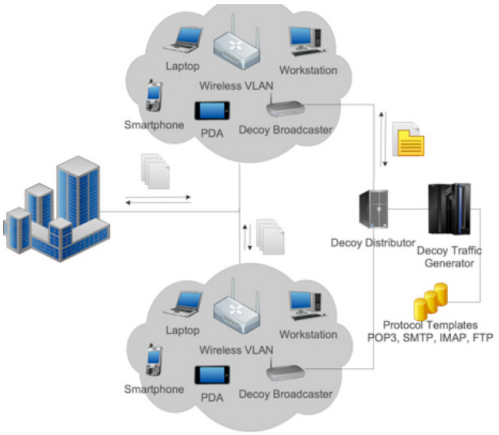

Figure 1: Injection Platform

3. 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. To guide in the creation of decoys, we leverage a core set of decoy properties defined in [5] which include *believability, non-interference, detectability, variability,* and *enticement*. We used some of these properties to aid in 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 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 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 shown 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—we have specifically designed the API to handle both types of input. 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 of the content is limited, which may subtract from the realism

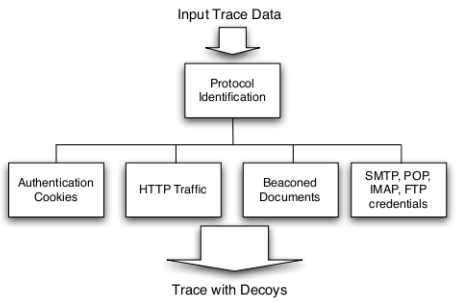

Figure 2: Honeyflow creation process

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

Figure 3: SMTP Identifiers

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.

Once the API obtains an input trace, a new trace is automatically created with decoy information as follows. First, each input trace consists of multiple protocols and TCP sessions. We demultiplex each session/protocol into individual trace files for simpler processing. Configuration information (*e.g.,* decoy information, IP/MAC addresses of emulated network) is read from a user specified configuration file. Each of the demultiplexed trace files are 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 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 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. Identified traces are then passed through a protocol *modifier* function to insert decoy information. The API supports rules for adding bait to protocol headers (*e.g.,* Gmail cookies, SMTP passwords) and protocol payloads (*i.e.,* email body, web page content). Our implementation includes rules for creating several types of decoys including: Gmail authentication cookies, URLs, passwords for unencrypted protocols (SMTP, POP, IMAP), and beaconed documents as email attachments [5]. The API can also be used to introduce bait HTTP flows that contain monitored URLs. The implementation can easily be extended to support other protocols by manually determining protocol identifiers and creating a new rule for them. Our API also handles protocol complexities such as:

1. If multi-packet editing is required (*e.g.,* insert decoy file attachment in POP3 trace), we buffer the data in memory. When a boundary is found (*e.g.,* 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.
2. The API formats the decoy information appropriately for the given protocol (*e.g.,* Base64 for POP3 attachments).

Rules are also used for the replacement of MACs and IPs to those from a predefined set to suit the environment. Additional variability and randomness are introduced to the honeyflows using the following techniques:

1. For identified TCP server protocols the client port is randomly generated.
2. 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.
3. Parameterization of temporal features (*e.g.,* total flow time, inter-packet time) that can be extracted from Netflow or packet trace data [12], enable the creation of honeyflows that are statistically similar to normal traffic.

OS fingerprint models of *p0f* [19] are also used to generate honeyflows that resemble the host OS. For example, to generate traffic that appears to emanate from a Linux host, we avoid creating traffic that appears to have come from the MS Outlook email client. Finally, the demultiplexed traces are combined into a single trace, which is then broadcasted to the environment.

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%

3.2 Decoy Broadcaster

The goal of our system is the realization of an inexpensive mechanism for broadcasting bait content over a network. Figure 1 indicates that it is comprised of both hardware and software entities. 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 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. However, an additional requirement of the decoy broadcaster is the support of *monitor mode*. In all other modes, injection either fails or it is 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` [9], a mesh router with 32 MB DRAM and 8 MB flash. Accton's wireless NIC uses the *MadWifi* [15] driver, which supports monitor mode. We tweaked the driver in order to suppress 802.11 ACK frames, since we have our own ACK frames recorded as part of the decoy traffic; and ignore ACK timeouts in injected frames.[1] Finally, to inject the honeyflows we ported *Tcpreplay* [14], a suite for replaying previously captured traffic for network testing purposes.

The most important property of the decoy repository on broadcasters is *freshness*. 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. Most 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 repetition of content or on the retransmissions of protocol header parts (*e.g.,* TCP seq numbers, IP TTL, TCP/UDP src port numbers, IP ID).

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 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[2], 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.

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. Thus, the 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.

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

[2]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 such 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 a realistic environment we conducted an experiment at the Defcon '09 hacking conference in Las Vegas. Defcon's yearly meeting includes the infamous *wall of sheep* [18], which is an interactive demonstration of what can happen when network users do not use the protection of encryption. Defcon staff eavesdrop on the network traffic for unencrypted credentials, which they later post on a publicly accessible wall as a 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, 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), 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 broadcast was not adequately sniffed by the Defcon staff. We later learned that Defcon staff were monitoring the switch mirroring ports as opposed to WiFi radio channels. However, this is orthogonal to our experiment.

5. BELIEVABILITY OF BOGUS TRAFFIC: A DECOY TURING TEST

Alan Turing proposed [17] 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 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 department computing research facility, 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 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 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 4 summarizes the results for each of 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 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 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 5 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. Interviewing the judges we concluded that the bias for decoys in Figure 4 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 traffic was decoy 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. 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.

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 to try and correlate traffic in order to distinguish real traffic from decoys. This scenario was outside the scope of 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

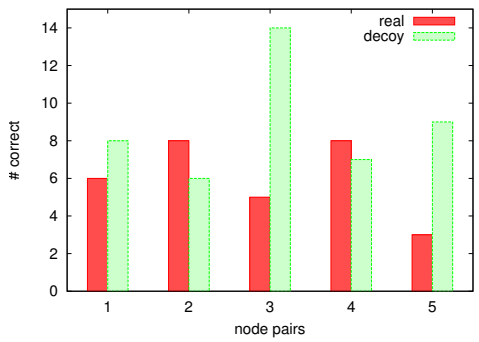

Figure 4: DTT Results: Real vs. Decoy.

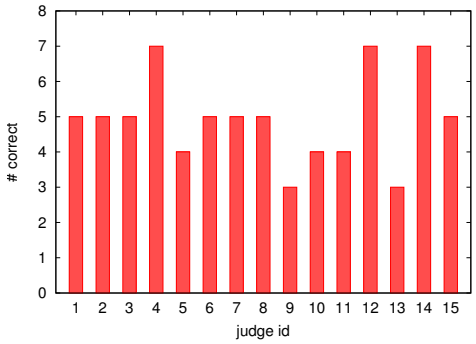

Figure 5: DTT Results: Users' Correctness.

we did not implement this, the problem can be easily fixed by only broadcasting decoy traffic when there is real traffic.

The believability of the 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 it may not have been possible to otherwise.

6. 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 from the real thing; our experienced judges achieved only 49.9% 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.

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

7. 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] 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.

[7] 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*, September 1998.

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

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

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

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

[12] 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.

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

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

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

[16] 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.

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

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

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