MITIGATING THE BOTNET PROBLEM: FROM VICTIM TO BOTMASTER

by

Daniel Ramsbrock
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Information Security and Assurance

Committee:

Dr. Xinyuan Wang, Thesis Director

Dr. Xuxian Jiang, Committee Member

Dr. Duminda Wijesekera, Committee Member

Dr. Hassan Gomaa, Department Chair

Dr. Lloyd J. Griffiths, Dean, The Volgenau School of Information Technology and Engineering

Date: April 21, 2008

Spring Semester 2008
George Mason University
Fairfax, VA

Mitigating the Botnet Problem: From Victim to Botmaster

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University.

By

Daniel Ramsbrock
Bachelor of Science
University of Maryland, 2006

Director: Dr. Xinyuan Wang, Professor
Department of Computer Science

Spring Semester 2008
George Mason University
Fairfax, VA

Acknowledgments

First, I would like to thank my adviser Dr. Xinyuan Wang (Department of Computer Science) for supporting me throughout my studies at George Mason University. He provided the guidance, funding, and materials that enabled me to perform this research. He was also instrumental in solidifying the theoretical and mathematical underpinnings of our botmaster traceback technique. In my research, I drew extensively on his prior work in the field of network traceback.

I would also like to thank committee member Dr. Xuxian Jiang (Department of Computer Science), who provided valuable feedback on our research. He also granted us access to the Agobot source code, from his collection of malware binaries and sources. Having the source code available was extremely helpful in understanding the internal workings of typical bot software.

Committee member Dr. Duminda Wijesekera (Department of Computer Science) played a large role in recruiting me to attend George Mason. When I decided to transfer from the University of Maryland, I was choosing between two strong IT security programs. Part of my reason for choosing George Mason was Dr. Wijesekera's recruiting effort, providing a personal touch and a point of contact during the application process.

Additionally, I would like to thank my fiancée Rachel Bernstein for spending countless hours editing my papers and thesis with me. She provides a unique non-technical perspective, always helping me clarify and streamline my writing.

Finally, I would like to thank the PlanetLab organization for providing their very useful service: a global platform for network testing and experimentation. In particular, I would like to thank Dr. Brian Mark (Department of Electrical and Computer Engineering), George Mason's local coordinator who facilitated our access to PlanetLab.

Table of Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

ASM	Assembly language
BNC	BouNCer. Proxy software for IRC that allows IRC users to make it appear that they are connecting from an different IP address.
C&C	Command and Control. Relating to the sending of commands and receipt of responses from a collection of hosts (usually bots).
CPU	Central Processing Unit
DDoS	Distributed denial-of-service attack. A type of traffic flooding attack originating from many sources at once, making it hard to stop the attack traffic and usually leading to the unavailability of the target system.
DNS	Domain Name System. The Internet-wide service responsible for translating textual domain names into numeric IP addresses.
FPR	False Positive Rate
FTP	File Transfer Protocol
GCC	GNU Compiler Collection. The primary open-source compiler system for the C and C++ programming languages.
HTTP	HyperText Transfer Protocol. The protocol used by Internet web servers to communicate with web browsers.
HTTPS	HyperText Transfer Protocol over Secure Sockets Layer. A secure version of HTTP that is run on top of SSL/TLS.
HZ	Hertz. A unit of frequency measurement in cycles per second.
IDS	Intrusion Detection System. A network system used to inspect traffic and generate alerts when suspicious packets are detected.
IP	Internet Protocol. The network-layer protocol on which the

IRC	Internet Relay Chat. A popular medium for real-time chat via the Internet, and also a mechanism widely used by botmasters for command and control of botnets.
IRQ	Interrupt ReQuest Line. A mechanism used for hardware interrupts in the x86 architecture.
ISP	Internet Service Provider
IT	Information Technology
MITM	Man-in-the-Middle. A type of network-based attack where a third party inserts itself into a client-server communication and impersonates the parties to each other. This allows the third party to observe and/or change all traffic passing between the parties.
OS	Operating System
P2P	Peer-to-Peer. A network topology where nodes connect only to one or more peer nodes rather than a single central node.
PRNG	Pseudo-Random Number Generator
RTAI	Real-Time Application Interface. A Linux kernel patch providing real-time programming capability and highly accurate timing.
SOCKS	A proxy protocol that allows network traffic to be encapsulated between a client and the SOCKS server, making it appear that the connection is originating from the SOCKS server.
SSH	Secure SHell. A secure mechanism for remotely logging into command-line based services such as UNIX and Linux shell accounts. SSH also provides tunnel features: it can forward ports between client and server and also act as a SOCKS server.
SSL/TLS	Secure Sockets Layer / Transport Layer Security. An end-to-end encryption mechanism for application-layer network traffic. Secure Sockets Layer (SSL) was the original name; it has since been renamed to Transport Layer Security (TLS).
TCP	Transmission Control Protocol. A reliable transport-layer protocol that is used for most types of Internet traffic, including web browsing, e-mail, and chat.
TPR	True Positive Rate

TSC	Time Stamp Counter. A special register in the x86 architecture holding the number of CPU instructions executed since start-up.
TTL	Time To Live
URL	Uniform Resource Locator
VM	Virtual Machine. A computer running in a self-contained virtual environment inside a physical computer. A single physical machine can host several VMs, which share its resources.

Abstract

MITIGATING THE BOTNET PROBLEM: FROM VICTIM TO BOTMASTER

Daniel Ramsbrock, M.S.

George Mason University, 2008

Thesis director: Dr. Xinyuan Wang

Despite the increasing botnet threat, research in the area of botmaster traceback is limited. The four main obstacles are 1) the low-traffic nature of the bot-to-botmaster link; 2) chains of "stepping stones;" 3) the use of encryption along these chains; and 4) mixing with traffic from other bots. Most existing traceback approaches can address one or two of these issues, but no single approach can overcome all of them.

Our early work focused on hijacking and sniffing botnet C&C traffic, especially when it is encrypted. We successfully executed MITM attacks on both IRCS- and HTTPS-based botnets. We further developed a kernel-level approach for obtaining millisecond-precision timing in a virtual machine environment, allowing us to run time-based watermarking code on virtual machines.

The major contribution of this work is a novel flow watermarking technique to address all four traceback obstacles simultaneously. Our approach allows us to uniquely identify and

trace any IRC-based botnet flow even if 1) it is encrypted (e.g., via SSL/TLS); 2) it passes multiple intermediate stepping stones; and 3) it is mixed with other botnet traffic. Our watermarking scheme relies on adding whitespace padding characters to outgoing IRC messages at the application layer. This produces specific differences in lengths between randomly chosen pairs of messages in a network flow. As a result, our watermarking technique only requires a few dozen packets to be effective. To the best of our knowledge, this is the first approach that has the potential to allow real-time botmaster traceback across the Internet.

We empirically validated the effectiveness of our botnet flow watermarking approach with live experiments on PlanetLab nodes and public IRC servers on different continents. We achieved virtually a 100% detection rate of watermarked (encrypted and unencrypted) IRC traffic with a false positive rate on the order of 10^{-5}. Due to the message queuing and throttling functionality of IRC servers, mixing chaff with the watermarked flow does not significantly impact the effectiveness of our watermarking approach.

1 Introduction

Botnets are currently one of the most serious threats to computers connected to the Internet. Recent media coverage has revealed many large-scale botnets worldwide. One botnet [30, 31] has reportedly compromised and controlled over 400,000 computers – including computers at the Weapons Division of the U.S. Naval Air Warfare Center, U.S. Department of Defense Information Systems Agency. Another recently discovered botnet is suspected to have controlled 1.5 million computers around the globe [15]. It has been estimated [27] that more than 5 percent of all computers connected to the Internet have been compromised and used as bots. Currently, botnets are responsible for most spam, adware, spyware, phishing, identity theft, online fraud and DDoS attacks on the Internet.

Most existing work on botnet defense [1, 2, 4, 9, 17, 20, 21, 25] has focused on the detection and removal of command and control (C&C) servers and individual bots. While such a capability is a useful start in mitigating the botnet problem, it does not address the root cause: the botmaster. For example, existing botnet defense mechanisms can detect and dismantle botnets, but they usually cannot determine the identity and location of the botmaster. As a result, the botmaster is free to create and operate another botnet by compromising other vulnerable hosts.

To address this issue, our research focused on three different but related areas: 1) design issues faced by the botmaster, allowing us to exploit design weaknesses to break encrypted C&C channels; 2) achieving millisecond precision in a virtualized environment in order to embed time-based watermarks; and 3) developing a practical watermarking scheme for IRC-based botnets, being resilient to stepping stones, encryption, and flow mixing. The first two projects laid the foundation for the full botnet traceback approach, allowing us to leverage our previous work and experience to develop a promising new traceback method that can trace botmasters across the Internet.

2 Literature Review

The botnet research field is relatively new, but many papers have been published in the last few years as the botnet threat has accelerated. As one of the first in the botnet arena, the Honeynet Project [1] provided a starting point for future exploration of the problem. Two of the authors on that project went on to explore infiltration techniques for IRC botnets [16]. A comprehensive study at Johns Hopkins University [28] constructed a honeypot-based framework for acquiring and analyzing bot binaries. The framework can automatically generate rogue bots (drones) to actively infiltrate botnets, which is the first step in injecting a watermark and tracing the botmaster.

Most early botnet work focused on defining, understanding, and classifying botnets. Some examples are papers by Cooke et al. [9], Dagon et al. [11], Ianelli and Hackworth [23], Barford and Yegneswaran [2], and Holz's summary in *Securit*y & *Privac*y [22]. Since then, bot detection has become more of a focal point and many techniques have been proposed. Binkley and Singh [4] presented an anomaly-based detection algorithm for IRC-based botnets. Goebel and Holz [17] reported success with their Rishi tool, which evaluates IRC nicknames for likely botnet membership. They also mention the possibility of using this algorithm on unencrypted HTTP botnets. Karasaridis et al. [25] described an ISP-level algorithm for detecting botnet traffic based on analysis of transport-layer summary statistics. Brodsky and Brodsky [6] proposed a distributed

method for detecting botnet-originated spam. In 2007, Gu et al. [20] detailed their BotHunter approach, which is based on IDS dialog correlation techniques. They also published a related paper in 2008 [21] where they introduce BotSniffer, a tool for detecting C&C traffic in network traces.

Methods for leveraging bots' dependence on DNS have also been described: an active DNS hijacking approach by Dagon et al. [12] and a passive method based on DNS blackhole lists [29].

At the host level, most bots can be detected by generic signature-based malware detection techniques. Some sophisticated bots cannot be detected by these generic means, resulting in some recent research focusing on advanced detection techniques. Stinson and Mitchell [36] described a taint-based method which traces network input to system calls in order to identify bot processes.

Barford and Blodgett [3] presented a method for constructing the Botnet Evaluation Environment (BEE), a testbed for realistic botnet research in a controlled setting.

Despite a large amount of literature regarding botnet detection and removal, relatively little work has been done on finding and eliminating the root cause: the botmaster himself. An earlier paper by Freiling et al. [16] describes a manual method of infiltrating a botnet and attempting to locate the botmaster, but the approach does not scale well due to lack of automation.

In the general traceback field, there are two main areas of interest: 1) network-layer (IP) traceback and 2) tracing approaches resilient to stepping stones. The advent of the first

category dates back to the era of fast-spreading worms, when no stepping stones were used and IP-level traceback was sufficient. A leading paper in this area is Savage et al. [33], which introduced the probabilistic packet marking technique, embedding tracing information an IP header field. Two years later, Goodrich [18] expounded on this approach, introducing "randomize-and-link" with better scalability. A different technique for IP traceback is the log/hash-based scheme introduced by Snoeren et al. [34], and enhanced by Li et al. [26].

There are a number of works on how to trace attack traffic across stepping stones under various conditions. For example, [5, 14, 43, 44, 45, 46, 47, 48] used inter-packet timing to correlate encrypted traffic across the stepping stones and/or low-latency anonymity systems. Most timing-based correlation schemes are passive, with the exception of the three active methods [44, 44, 45]. Our proposed method is based on the same active watermarking principle used in these three works. However, our method differs from them in that it uses the packet length, in addition to the packet timing, to encode the watermark. As a result, our method requires much fewer packets to be effective than do methods [44, 44, 45].

3 Exploiting Design Trade-Offs to Circumvent Botnet Encryption

Note regarding terminology: In this chapter, we use the term "botmaster" to refer to both a person who issues commands to a botnet and to a programmer who authors a bot and makes it available to other botmasters (a botnet designer). Also, our usage of "attacker" is somewhat counterintuitive: it refers to a white-hat security professional who is attacking the botnet for research and/or disruption purposes. This person is also frequently described as a "defender" in botnet literature.

3.1 Botnet Requirements and Trade-Offs

When designing a botnet as a secure distributed system, there are several important requirements that the botmaster must consider. We list them roughly in the order of importance to the botmaster.

1. *Anonymity*: The botmaster's identity must be protected when connecting to the botnet.
2. *Stealth*: Its presence must be hard to detect, both at the host and network level.
3. *Robustness*: It must be resilient to active attack and interference by both security professionals and rival botmasters.
4. *Reliability*: The botnet must be available under normal operating circumstances.
5. *Scalability*: It must be able to support very large numbers of bots (if needed).

During design and implementation of a botnet, the botmaster has many choices to make to fulfill the above requirements. Most of these choices are in fact trade-offs, where he has to sacrifice one benefit for another. He needs *Server Addressing* for bots to locate the C&C server and report for duty. He needs *Client Authentication* to ensure that only authorized bots can connect to the server. If employed, *C&C Protection* provides stealthy C&C channels and *Server Authentication* ensures that the bots only connect to the real C&C server. Finally, his choice of *Topology* and *P2P Encryption* (if applicable) also have important security implications.

This list is not exhaustive, but rather an overview of the major trade-offs relevant to active botnet interference. The relevant design requirement(s) are indicated in parentheses.

Server Addressing (Robustness): The botmaster must choose how to direct bots to their C&C server: either with a hard-coded IP address or via dynamic DNS. A hard-coded IP decreases flexibility: if the C&C server is compromised, bots cannot be redirected to a different server. However, relying on DNS opens up new detection vectors [12, 29] and also allows the DNS provider to simply cut off service for the hostname in question once a botnet is detected. Most importantly, the use of DNS enables man-in-the-middle (MITM) attacks, which rely on DNS spoofing.

Client Authentication (Robustness): The bot-master must choose whether he wants to protect his C&C server from intruders. He can do this by requiring a password to access the server, the IRC channel (if applicable), and for issuing commands to the bots. However, requiring passwords forces him to hard-code them into the bot binaries, thus

reducing flexibility. Most botmasters in the wild are willing to make this trade-off, using hard-coded passwords to implement server protection. Despite this measure, we can recover the passwords even when C&C encryption is present.

C&C Protection (Stealth, Robustness): The bot-master must choose how to protect his C&C channels: no protection (plaintext), obfuscation (unkeyed), or encryption (keyed). Supporting obfuscation introduces some processing overhead, especially on the C&C server. Supporting encryption can add significant overhead, depending on whether a public-key or symmetric-key scheme is used. Public-key schemes have the advantage of not requiring any pre-shared keys, but symmetric-key schemes often require much less overhead, especially during connection initiation. However, symmetric-key schemes have a key management and exchange problem that grows rapidly with each new node.

Server Authentication (Robustness): The bot-master must choose whether he wants to authenticate his C&C server to each bot. He can do this with a classic public-key authentication scheme similar to SSL/TLS, where each bot has a known public key for the server and sends a random nonce encrypted with that key. If the server can decrypt this nonce, then it must have the corresponding secret key and be the authorized C&C server. However, this requires hard-coding of the server's public key into each bot binary, which would render all bots useless if the server is compromised. The vast majority of bots in the wild do not perform server authentication and are therefore subject to MITM attacks. A known exception to this is the advanced Rustock spam bot [8], which uses a hard-coded public-key authentication scheme as described above.

Topology (Scalability, Reliability, Robustness): The botmaster must choose between two major topologies: centralized and peer-to-peer (P2P). Several papers also mention a random topology as an option [9, 11], but this has not been observed in the wild or demonstrated in practice [39]. P2P has the advantage of not providing a single point of failure, but can take much longer to propagate messages and can provide other unique opportunities for attacks. For example, it introduces a large key management problem, making it nearly impossible to reliably authenticate all nodes to each other, thus enabling MITM attacks.

P2P Encryption (Scalability, Stealth): When using a P2P topology and encryption, the botmaster must choose between end-to-end and hop-by-hop encryption. End-to-end encryption provides better performance because each intermediate node does not have to decrypt and re-encrypt messages. It also provides better protection since none of the intermediate nodes can decrypt the traffic they are forwarding. However, it cannot support "broadcast" mode, where each node acts on the command it receives before passing it on. Hop-by-hop encryption is less efficient, but it can support the important broadcast feature. A drawback of hop-by-hop encryption is that each node is (intentionally) a MITM, so a compromise of one bot node reveals all traffic flowing through that node.

Currently, most botmasters are willing to make the trade-offs relating to server protection, but relatively few have moved on to encryption and other advanced techniques. However, some are starting to explore these options [8, 19, 42], and we believe that once the

detection and disruption of unencrypted botnets becomes too expensive to their business, botmasters will start to adopt encryption [23] and other techniques such as P2P.

3.2 Attacks on Botnets

3.2.1 Unencrypted C&C Channels

Botnet literature indicates that currently the most widely used C&C medium is unencrypted IRC traffic [23, 28, 39]. However, many bots have recently started using unencrypted HTTP channels as well [23, 39]. Once an unencrypted botnet is detected, it is usually straightforward to infiltrate it since the attacker can observe all C&C traffic, including passwords.

For all attacks below, we assume that the attacker only has access to a copy of the bot binary. He has the ability to run this binary on his own machine and can thus observe its behavior, both local system interactions and network traffic.

The attacks discussed here are not designed to be a comprehensive listing of all possible attacks on unencrypted C&C channels. Rather, they illustrate the most common methods which security professionals and rival botmasters can use to infiltrate and disable a botnet.

3.2.1.1 Unencrypted IRC

Most IRC botnets implement *Client Authentication* via three passwords: one to connect to the IRC server, one to join the botnet channel, and a third to issue commands to the bots. Since most of these passwords are currently sent in the clear, an attacker can observe

them using a network sniffer. He can use this information to create an automated rogue bot (or "drone") that joins the botnet channel and records all IRC traffic while pretending to be a real bot. This is a well-known approach that has been used by both academic and industrial researchers to infiltrate existing botnets and estimate their size and scope [16, 28].

We tested this attack with an Agobot variant (agobot3-priv4) that we compiled from source code with known passwords. We readily observed the server/channel passwords and the botnet channel name as soon as the bot connected to the server.

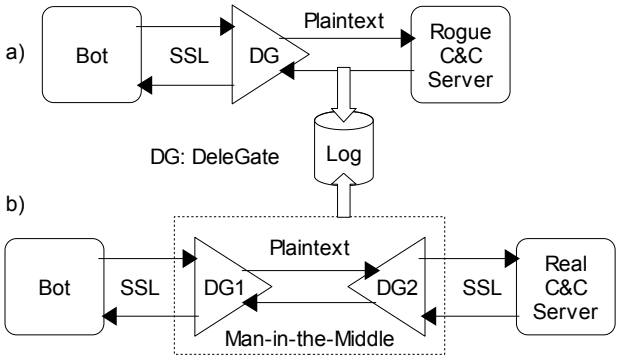

Figure 1: Experimental Setup a) Gray-Box and b) Man-in-the-Middle

To observe the botmaster password in plaintext and use it to issue commands to the botnet, the attacker needs to wait until the real botmaster logs in. We simulated a botmaster login with Agobot and successfully observed the botmaster password being sent in the clear.

Once the attacker has the server/channel passwords, the channel name, and the botmaster password, he is free to impersonate the botmaster at will. This includes the ability to send

the commands that cause bots to lock down their hosts and self-terminate, effectively shutting down the entire botnet attached to the C&C server. We verified this approach with Agobot and successfully sent the `bot.secure` and `bot.remove` commands to shut down our local test botnet.

This technique is highly effective, disabling entire botnets at once. While it is feasible and perfectly legal in a laboratory setting, this may not be true for Internet-wide live botnets. The computers infected with bots belong to hundreds of users, and impersonating the botmaster and sending commands to them constitutes unauthorized access to these computers. Most likely, only authorized law enforcement agencies would be able to use it legally.

3.2.1.2 Unencrypted HTTP

Like IRC bots, HTTP bots can also require a server password for *Client Authentication*. The simplest mechanism is to have a dynamic server page require a password as a plaintext GET parameter before it issues commands to the bot. However, this scheme is vulnerable to the same trivial sniffing attack as the IRC passwords.

The definition of "botmaster password" is not clear in the realm of HTTP bots. One candidate is the SSH or FTP password that is required to upload command files to the web server. This is equivalent to the IRC concept of a botmaster password, since this is what the attacker needs to send commands to the entire live botnet. However, this password is very hard to compromise from the perspective of the attacker. He has no

information about the botmaster's real location or IP, much less the ability to monitor traffic between the botmaster and the web server.

An alternative attack (not requiring a password) is DNS redirection, similar to [12]. This attack exploits the fact that DNS is used for *Server Addressing*. Using this attack on an Internet-wide scale requires the cooperation of the DNS operator, having them point to a web server under the attacker's control, where he can post arbitrary commands. However, the attack also works on a smaller scale: If the local network administrator sets up a spoofed DNS record, he can direct all bots on his network to the rogue web server.

Another possibility for the botmaster password is to include a password (or its hash) with each command file. However, as with IRC, the password/ hash is exposed in plaintext every time a bot polls the server and can be easily replayed by the attacker.

3.2.2 Encrypted C&C Channels

In the previous section, we illustrated how unencrypted botnets can be infiltrated and even shut down with relatively little effort. The next logical step for botmasters is to use encrypted C&C channels to prevent trivial attacks. Intuitively, this measure would seem to defeat all of the attacks presented in Section 3.2.1. However, we have developed two network-level attacks that can recover the contents of encrypted botnet C&C channels.

These attacks exploit primarily the *Server Addressing* and *Server Authentication* trade-offs, requiring the manipulation of a DNS lookup and a lack of server authentication. They work on both IRC and HTTP-based bots, and may also prove useful for P2P

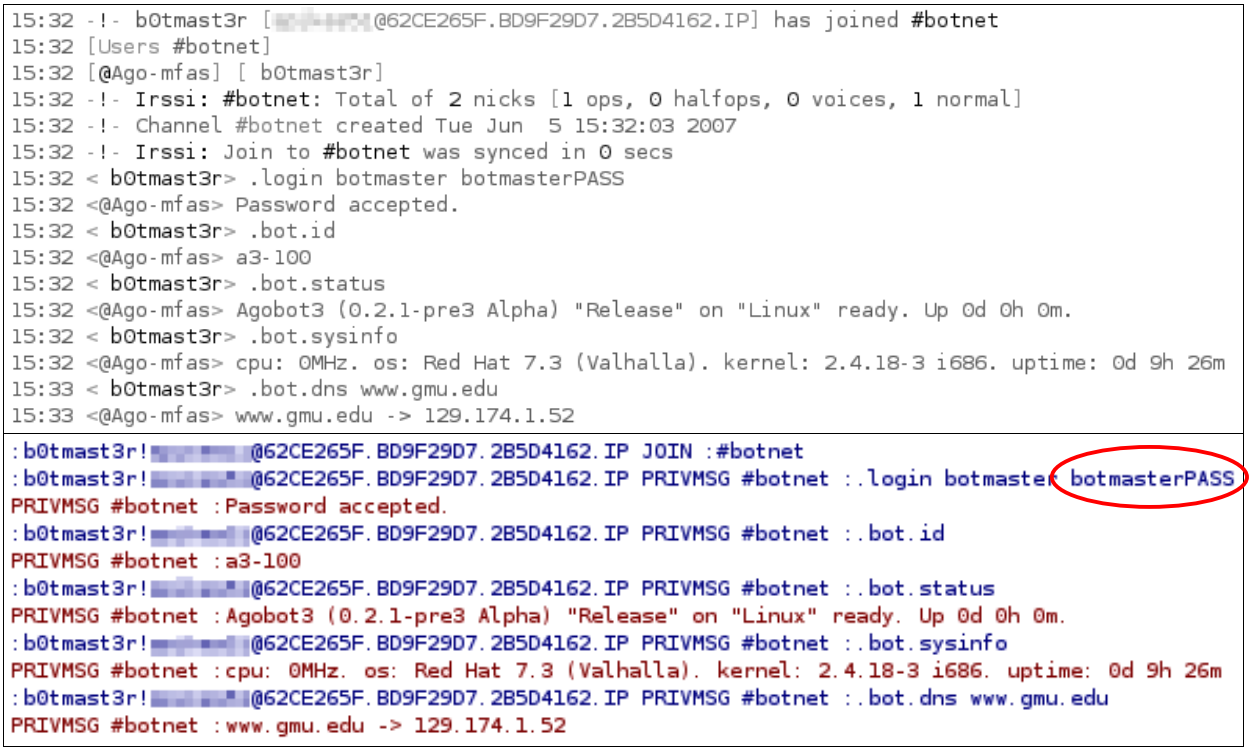

Figure 2: Agobot Encrypted IRC: The botmaster's view (top) and MITM network trace

botnets. As with the attacks on unencrypted C&C channels, we assume that we only have access to the bot binary and network traffic.

For the attacks, we assume that SSL/TLS is being used to provide end-to-end encryption, as this is a likely choice and is already supported by some bots, such as Agobot. If a custom encryption scheme is in place, the general methods presented here will still work as long as the encryption algorithm is known.

3.2.2.1 Gray-Box Analysis

This attack starts by simply running the bot and letting it connect to its normal C&C server. Even though all communications with the C&C server are encrypted, the attacker can readily recover its DNS name and IP address because DNS queries are performed

without encryption. The majority of bots use dynamic DNS for *Server Addressing* to give their C&C servers flexibility and survivability. Thus, the attacker can intercept the DNS query and use its contents to set up DNS spoofing. To do this, he creates a matching local DNS entry that points to an IP address of his choice, either via a local DNS server or the static host definition file.

At this IP, the attacker sets up his rogue C&C server with SSL/TLS support and full traffic logging. An alternative to logging is to use an SSL/TLS gateway like stunnel [37] or DeleGate [13] to translate the incoming SSL/TLS connection to an unencrypted stream (as in our experiment, see Figure 1a). This simplifies data collection since all traffic can be logged with a network sniffer. When the bot next connects to its C&C server, it is directed to the rogue server. Assuming the bot does not perform *Server Authentication*, the attacker can observe the complete interaction between the bot and his server in plain-text, including any server and channel passwords, as well as the channel name (with IRC).

Having gathered this information, the attacker can compile his own version of the bot (assuming he has the source code of the bot family available). This rogue bot can again join the live botnet and perform a number of monitoring functions. Since the rogue bot has a legitimate encrypted connection with the server, it can observe and log all commands sent by the botmaster, and it may also be able to obtain a membership listing of the botnet, depending on the server configuration. With his rogue bot monitoring all botnet commands, the attacker can also recover the botmaster password when it is sent,

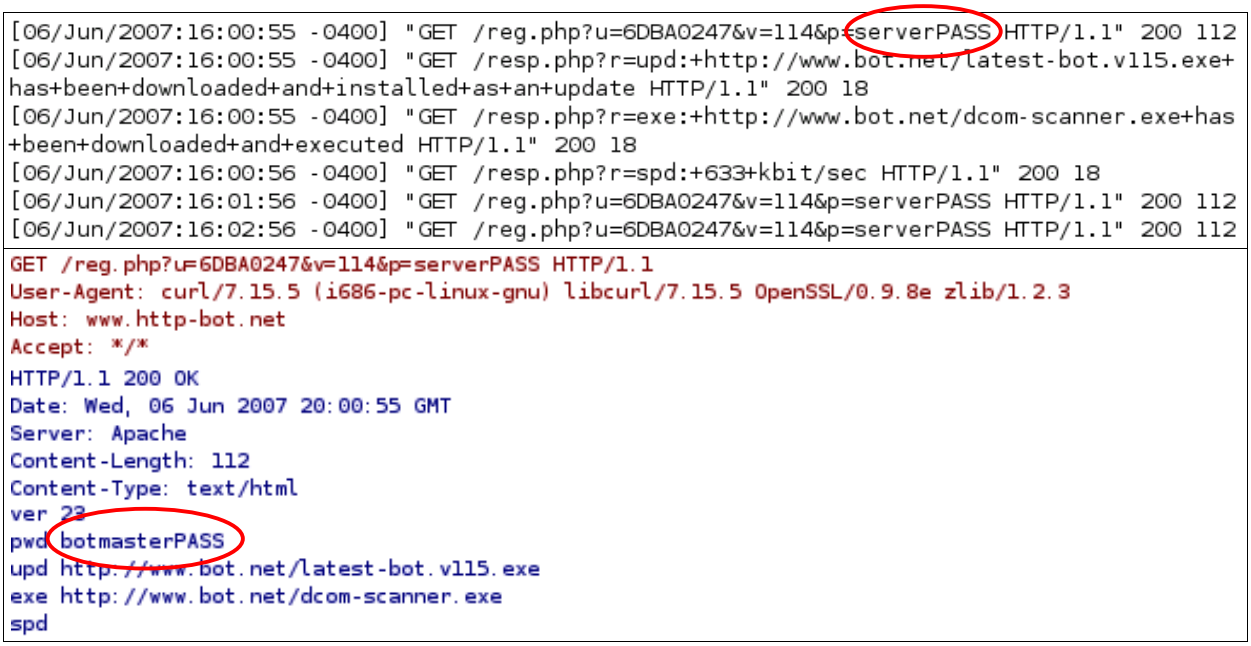

Figure 3: Proof-of-Concept HTTPS Bot: The botmaster's log (top) and MITM network trace

just as in the unencrypted case in Section 3.2.1. This again creates an opportunity for shutting down the entire botnet, assuming the legal authority exists.

3.2.2.2 Man-in-the-Middle Attack

This attack is more sophisticated than the gray-box analysis above since it allows the attacker to spy on a live botnet connection rather than just forcing the bot to connect to the attacker's rogue C&C server. It is an implementation of the well-known MITM attack on SSL/TLS, exploiting the lack of authentication between the bots and the C&C server (see Section 3.1).

The key to this attack is to force the bot to connect to a third party (the MITM) controlled by the attacker when it tries to connect to the real C&C server. The MITM accepts the encrypted connection from the bot and simultaneously initiates a second encrypted

connection to the real C&C server (under a different session key). All traffic passing through this connection is first decrypted on the MITM machine, then logged, re-encrypted, and passed along on the second connection (see Figure 1b). As with the gray-box analysis, DNS spoofing is required in order to trick the bot into connecting to the MITM rather than the real C&C server.

While the concept of a MITM attack is not new, its application to encrypted botnet C&C channels is. It works on individual hosts but could also be used to sweep entire networks for bots with encrypted C&C channels. The DNS redirection would need to be done on a local DNS server under the control of the network administrator so that all connections to suspicious dynamic DNS names (or even entire domains) get routed to the MITM. There, the decrypted traffic can be analyzed for content using established network-based bot detection techniques to separate legitimate traffic from bot traffic. By its nature, this approach would have severe privacy implications, but these may be acceptable depending on the context. It would not be appropriate for an Internet Service Provider (ISP), but within a company or government agency, it might be considered part of standard network monitoring.

The main limitation of both the gray-box analysis and the MITM attack is that they only work when no *Server Authentication* is taking place, as described in Section 3.1. However, even if the bot enforces authentication and the MITM attack fails to reveal the contents of the connection, it will still achieve its ultimate goal: the bots will refuse to connect to the MITM due to mismatching public keys. They are also unable to obtain the

real IP of the C&C server since the local DNS server is misleading them. Thus, the bots are unable to connect to their C&C server and are useless to the botmaster.

If the bot enforces *Server Authentication*, there are host-based methods of recovering plaintext and key material from the bot-infected machine. Chiang and Lloyd illustrate a bot-specific technique for key recovery in [8]. In a related paper, Jiang and Wang demonstrate a generic method for transparently intercepting system calls in virtual machines with VMscope [24]. This could allow us to recover the plaintext and possibly the key material directly from the bot-infected host.

3.3 Experiment Results

To verify our attacks on encrypted C&C channels, we conducted four experiments: the gray-box analysis and the MITM attack, both on two different types of bots. The first bot is a standard version of the IRC-based Agobot (agobot3-priv4), which includes SSL/TLS support. The second bot is a proof-of-concept implementation of an HTTP bot with SSL/TLS. Figure 1 illustrates our experimental setup in detail.

3.3.1 Agobot IRC Bot with SSL/TLS

We acquired the source code for Agobot on the Internet and discovered that it supports IRC over SSL/TLS via the `use_ssl` configuration variable. It seems that this feature is not yet widely used in the wild, so we were unable to locate a "live" Agobot binary with SSL/TLS enabled. For our experiments, we compiled the source tree with some reasonable configuration settings: it connects to the fictional host `evil.bot.net` on port 6667 and

joins channel `#botnet`. The IRC server password is `serverPASS`, the channel password is `channelPASS`, and the botmaster's login is `botmaster`, with password `botmasterPASS`.

For the gray-box analysis, we used DeleGate to emulate an SSL/TLS-enabled IRC server. First, we verified that we could observe the bot's DNS query in plaintext (assuming no prior knowledge of the C&C server's DNS name). Next, we modified the static DNS on the bot host machine so that instead of connecting to the real C&C server, it connected to the DeleGate proxy, which decrypted the packets and passed them on to a non-SSL/TLS IRC server running UnrealIRCD [40]. We logged all network traffic passing between DeleGate and the IRC server and readily observed the server password, channel name, and channel password. Armed with this information, we could now build our own rogue bot and infiltrate the live botnet as described in Section 3.2.1.1.

For the MITM attack, we left the DNS spoofing in place. We used two separate DeleGate proxies: one to handle the encrypted connection with the bot and one to handle the encrypted connection with the C&C server (see Figure 1b). These two instances communicated via the local loopback interface in plaintext, allowing us to observe and log all traffic passing between the bot and the C&C server. The top of Figure 2 shows the IRC channel as the botmaster saw it during the MITM attack. At the bottom, Figure 2 shows the decrypted network trace from the MITM between the bot and the IRC server. The lines starting with `:b0tmast3r` are the commands received from the server and the ones starting with `PRIVMSG` are the bot's responses. The recovered plaintext botmaster password is circled. Due to space constraints, we were unable to show the full login

exchange between the bot and IRC server, which revealed the server and channel passwords in plaintext.

3.3.2 Proof-of-Concept HTTP Bot with SSL

Despite extensive searching, we were unable to find any bots which currently use or even support HTTP over SSL as a C&C channel. McAfee Avert Labs supports this finding, stating in an e-mail to us that they see "many encoding or encryption mechanisms on top of HTTP but no proper usage of HTTPS." However, this possibility has often been mentioned in existing botnet literature [17, 23], and we believe that many future bots will utilize it.

Due to the lack of HTTPS bots, we used Perl to implement a simple proof-of-concept bot that uses cURL [10] for the underlying HTTPS operations. The small command set it supports is based on the Bobax HTTP bot [35]. It connects to one of our SSL/TLS-enabled Apache/PHP web servers, which simulates the C&C server. It uses the fictional host 'www.http-bot.net,' the standard HTTPS port 443, and it ignores invalid SSL/TLS certificates. As with most HTTP bots, the botmaster posts a list of commands to the web server, and the bot polls a specific URL on that server at regular intervals for an updated command list (60 seconds in our case). When commands are executed, our bot merely prints an acknowledgment message but does not perform any actions. The bot returns its results to the C&C server via GET parameters in the URL. The bot supports the basic protection mechanisms outlined above: a server password (`serverPASS`) is required before a command file is sent and the bot can require a botmaster password (`botmasterPASS`) to be sent with each command file.

For the gray-box analysis, we again verified that we could observe the bot's initial DNS query, then added an entry to direct the bot to our local Apache web server. As expected, the bot executed the commands specified and returned the proper responses, and we observed the entire exchange in plaintext. We used a version of the bot that did not require a botmaster password with each command file. If the bot had required such a password, we would need to run the MITM attack first and observe the bot polling the real C&C server (which would disclose the botmaster password contained in the command file).

For the MITM attack, we used nearly the same setup as for the IRC MITM experiment. The only difference were the ports used by the DeleGate gateways and the destination of the forwarding. We again captured the plaintext of all traffic passing between DeleGate and the HTTPS server, readily observing not only the server password but also the botmaster password included with the command file (we had our bot require a botmaster password in this experiment). Again, the bot executed its commands and returned the proper responses to the server. The top of Figure 3 shows the botmaster's view of server log on the C&C server, showing the bot's responses (`resp.php`) and polling of the command URL once per minute (`reg.php`). It only executed the commands and returned results the first time since the version of the command file (`ver 23`) remained the same the next two times. The bottom half of Figure 3 shows the decrypted network trace from the MITM between the bot and the web server. The first four lines are the bot's request, and the remaining lines are the server's response (i.e. the commands to be executed). Both the server password and botmaster password (circled) were visible in plaintext.

4 Performing Highly Time-Sensitive Traffic Watermarking on Virtual Machines

Traditionally, applications requiring a high level of timing accuracy have not been able to run successfully on virtual machines (VMs) such as VMware, QEMU, and Microsoft Virtual PC.

Operating-system level methods of timekeeping rely on the assumption that the OS has a monopoly on the CPU, but this is not the case in a virtual environment. As a result, the internal time of a VM is very unstable, especially at sub-second intervals. It frequently speeds up and slows down, depending on the interrupts and load on the VM and its host.

To address this problem, we have developed an approach that uses the host machine's clock to compensate for the inconsistencies in the VM time. In our case, the objective was to port a time-based watermarking engine from a real-time setting with dedicated hardware to a virtual environment. This software requires a handler function to run once per millisecond to check for new packets to send out, and it can tolerate no more than a few milliseconds of deviation from this schedule.

4.1 Introduction

This chapter describes the challenges we encountered in porting a custom time-based network-flow watermarking engine from a real-time setting to a virtual environment.

Our watermark engine is implemented as a Linux kernel module and an extension for iptables, Linux's packet-routing system. Previously, it ran only on two dedicated hardware machines (x86 architecture), each running a custom-compiled 2.6.10 kernel with the Real Time Application Interface (RTAI) add-on. Given the complexity of configuring the system, it would be very time-consuming to migrate to a new environment. To create a more flexible and portable solution, we turned to VMware. This allows us to have a very specific configuration inside the VM, but still be able to move the VM to any desired host.

However, achieving the highly accurate timing required for the watermark engine presented several challenges in the virtual environment. First, RTAI only runs on physical hardware, so we had to find ways to replace it. Second, the time inside the VM is very irregular, especially at sub-second intervals. Finally, we ran into some issues with 64-bit division in a 32-bit kernel. The 64-bit values are necessary because a 32-bit value counting microseconds can only cover a period of about 71 minutes before it overflows. By contrast, a 64-bit value can last over 584,942 years.

The remainder of this chapter is organized as follows. Section 4.2 introduces some of the traditional methods used by OSes to keep time. Section 4.3 describes how we replaced the use of RTAI. Section 4.4 explains how we developed a mechanism to compensate for the unreliable time inside the the VM. Section 4.5 covers the process of obtaining time from TSC readings, including the issues relating to 64-bit division. Finally, section 4.6 summarizes the chapter.

4.2 Traditional OS Timekeeping Methods

For most of its existence, Linux (and most other operating systems) have relied on the timer interrupt to keep track of time. This is a hardware timer that is scheduled a certain number of times per second, depending on the kernel version. The setting of this timer (also known as the HZ number of the kernel) represents the smallest unit of time that the kernel can use for scheduling. Earlier versions of Linux used 100 Hz (every 10 milliseconds), but newer versions use either 1000 Hz (once per millisecond) or 250 Hz (every 4 milliseconds).

Very recently (since kernel version 2.6.15), the option to run a "tickless" kernel has been introduced, which no longer uses the periodic time interrupt mechanism. The details of tickless kernels are a separate topic, and the majority of Linux kernels still use the old timer-interrupt scheme.

Since the Pentium (i586), the x86 architecture has included the Time Stamp Counter (TSC). This is a 64-bit counter that keeps track of the number of CPU cycles since system boot. It can be read via the `rdtsc` ASM instruction and can be used to calculate time intervals with very high accuracy (assuming one knows the exact CPU frequency). The accuracy of this timer is a function of CPU frequency: assuming a CPU speed of 1 GHz, the TSC can obtain nanosecond resolution. A major drawback of the TSC is that there is no reliable way to determine the exact CPU frequency—it has to be calibrated against other available timers. Also, in a laptop setting where the CPU frequency can change dynamically as a power-saving feature, TSC-based timekeeping is unreliable.

While these traditional timekeeping methods work well on physical machines, they become unreliable in a virtual setting. The VM has to compete for CPU cycles not only with the host but also with other VMs that are running on the same host, resulting in frequent speed-ups and slow-downs. For the most part, these average out over time, but they cause time to be unreliable at the sub-second level. Since we require accuracy within a few milliseconds, the VM internal time is insufficient for the watermark engine. For more details about the issues involved in VMware timekeeping, please refer to [41].

4.3 Replacing RTAI

In the previous version, RTAI was used to schedule the `wm_send` function in the watermark kernel module (`ipt_ctdWMencodePb`) at precise 1-millisecond intervals. This function is responsible for checking the queue of packets and sending them out at their intended time.

As mentioned above, RTAI does not work inside a VM, so we had to find a suitable replacement. The Linux kernel provides soft-IRQ timers that can be used to schedule events at the granularity of the timer interrupts (based on the kernel's HZ number). We used a 2.6.10 kernel, which has a default HZ of 1000, providing 1-millisecond resolution.

While RTAI provides a periodic timer feature, the kernel timers operate in one-shot mode. However, they can be easily rescheduled via the `mod_timer` function (which is called at the beginning of the `wm_send` function in our case).

We were able to successfully replace RTAI with a kernel timer that fires once per millisecond and executes the `wm_send` function.

4.4 Compensating for Unreliable VM Time

As explained in Section 4.2 and in [41], VM time (also called apparent time) is unreliable during small intervals, so even though we schedule the function to execute once per millisecond, it may be off this schedule by a significant amount. We cannot fix this property of the kernel timers, but in order to account for these variations, we need to know how much actual time passes in between each call.

The only way to obtain this information is by accessing a time source external to the VM, and the host machine's TSC is the best candidate for this [41]. By default, VMware emulates the TSC, providing values that are consistent with VM time (and therefore useless to us). However, by setting `monitor_control.virtual_rdtsc = false` in the VM configuration, any TSC accesses within the VM will be passed on to the host hardware directly.

In order to use the host TSC readings to measure time accurately, we need to know the exact frequency of the host CPU. This frequency can be closely approximated to the nearest kHz by calibrating the TSC measurement against the other available system clocks (which are reliable on the host system). For this purpose, we wrote a small calibration program that runs on the host, starting with the reported CPU frequency (from `/proc/cpuinfo`) and searching a range of values below and above this frequency. By default, it searches the reported frequency plus/minus 5,000 kHz, first in steps of 100 kHz, and then in steps of 1 kHz in the optimal 100 kHz range.

This calibration results in a very close approximation of the host CPU frequency, and this allows us to determine the time from the TSC with very little drift. A typical drift value we observed after calibration was 19 microseconds over a one-minute period, a drift of only 9.9864 seconds per year.

At this point, we have all of the tools to compensate for the irregularity of VM time, but the process of obtaining clock time from the TSC readings is also non-trivial.

4.5 Obtaining Time from TSC Readings

Since the TSC counts CPU instructions and not units of time, we cannot directly obtain a time value from it. To calculate the clock time that has passed between two TSC readings, we use the following series of equations:

$$msec_{tsc} = \frac{tsc - tsc_{prev}}{CPU_{kHz}} \tag{1}$$

$$msec_{frac} = \frac{msec_{mod}}{CPU_{kHz}}$$

$$sec_{whole} = \frac{msec_{tsc}}{1{,}000}$$

$$msec_{whole} = sec_{mod}$$

$$usec = 1{,}000(msec_{whole} + msec_{frac})$$

$$usec = usec + 1{,}000{,}000(sec_{whole})$$

The $msec_{mod}$ and sec_{mod} variables refer to the remainder of the previous division operations (those used to calculate $msec_{tsc}$ and sec_{whole}, respectively). All operations are on integers, except for those relating to $msec_{frac}$, which are decimal operations. At the end of these

calculations, the *usec* variable contains a very accurate number of microseconds that have passed since the last TSC reading.

These somewhat cumbersome calculations are necessary to obtain the required accuracy. Initially, we were simply dividing tsc – tsc_{prev} by the CPU frequency in MHz, but this resulted in a very inaccurate number of microseconds passed. However, dividing by the frequency in kHz and discarding the remainder also results in an inaccurate number of microseconds (since the calculation is only accurate to the millisecond).

The major problem we encountered during the implementation of these calculations was that the TSC values are 64 bits wide, and even the difference between two TSC readings requires more than 32 bits in most situations. In our case, the host CPU runs at 2.8 GHz, meaning that a 32-bit value can only cover a period of 1.54 seconds before overflowing.

Storing 64-bit values within kernel code is not a problem using the `unsigned long long` type, but division and multiplication are not natively supported. When compiling userspace code, GCC links against libgcc, which provides a software implementation of these operations. However, the kernel is not linked against this library, leaving it up to us to implement 64-bit division. We considered several "hack" implementations, including bit-shifting to divide by powers of two. However, we realized that we would need a better implementation to obtain not only an accurate result but a remainder as well. Luckily, we were able to copy the 64-bit division function almost verbatim from libgcc (`__udivmoddi4.c`). It's a full binary long division implementation that provides an exact result and remainder.

For the periodic calls of the `wm_send` function, calculating the number of microseconds between calls is enough (this is used to decrement a TTL value measured in microseconds). However, other parts of the watermark engine require absolute timestamps, which it was previously obtaining from the `do_gettimeofday` function. This function relies on the unreliable VM time, so we decided to implement our own function, `tsc_gettimeofday`. When the kernel module is initialized, we obtain a baseline by querying the VM time and TSC at the same time. The VM time may not correspond to the actual time, but this is acceptable since it will result in a constant offset.

Every time `tsc_gettimeofday` is called, it calculates the difference between the current TSC reading and that obtained at initialization. It then uses the formulas above to calculate the number of seconds and microseconds passed since initialization, creating a timeval structure from this information. Finally, it adds this timeval to the one obtained at initialization and returns the result as the current time.

4.6 Conclusion

In porting a time-sensitive application from a physical to a virtualized VMware environment, we encountered three main challenges: 1) we had to replace RTAI with kernel soft timers, 2) compensate for the inaccurate time within the VM, and 3) properly handle 64-bit division in a 32-bit kernel. After solving these problems successfully, the watermarking engine now works under VMware.

5 Botmaster Traceback via Hybrid Length and Time Watermarking

Botmasters can currently operate with impunity due to a lack of reliable traceback mechanisms. However, if the botmaster's risk of being caught is increased, he would be hesitant to create and operate botnets. Therefore, even an imperfect botmaster traceback capability could effectively deter botmasters. Unfortunately, current botmasters have all the potential gains from operating botnets with a minimal risk of being caught. Therefore, the botnet problem cannot be solved until we develop a reliable method for identifying and locating botmasters across the Internet. This chapter presents a substantial first step towards achieving the goal of botmaster traceback.

Tracking and locating the botmaster of a discovered botnet is very challenging. first, the botmaster only needs to be online briefly to issue commands or check the bots' status. As a result, any botmaster traceback has to occur in real-time. Second, the botmaster usually does not connect directly to the botnet C&C server and he can easily launder his connection through various stepping stones. Third, the botmaster can protect his C&C traffic with strong encryption. For example, Agobot has built-in SSL/TLS support. finally, the C&C traffic from the botmaster is typically low-volume. As a result, a successful botmaster trace-back approach must be effective on low-volume, encrypted traffic across multiple stepping stones.

To the best of our knowledge, no existing traceback methods can effectively track a botmaster across the Internet in real-time. For example, methods [5, 14, 43, 44, 45, 46, 47] have been shown to be able to trace encrypted traffic across various stepping stones and proxies, but they need a large amount of traffic (at least hundreds of packets) to be effective. During a typical session, each bot exchanges only a few dozen packets with the botmaster. Due to this low traffic volume, the above techniques are not suitable for botmaster traceback.

In this chapter, we address the botmaster traceback problem with a novel packet flow watermarking technique. Our goal is to develop a practical solution that can be used to trace low-volume botnet C&C traffic in real-time even if it is encrypted and laundered through multiple intermediate hosts (e.g., IRC servers, stepping stones, proxies). We assume that the tracer has control of a single rogue bot in the target botnet, and this bot can send messages in response to a the query from the botmaster. To trace the response traffic back to the botmaster, the rogue bot transparently injects a unique watermark into its response. If the injected watermark can survive the various transformations (e.g., encryption/decryption, proxying) of the botnet C&C traffic, we can trace the watermark and locate the botmaster via monitoring nodes across the Internet. To embed the watermark, we adjust the lengths of randomly selected pairs of packets such that the length difference between each packet pair will fall within a certain range. To track encrypted botnet traffic that mixes messages from multiple bots, we developed a hybrid length-timing watermarking method. Compared to previous approaches [43, 44, 45], our two proposed methods require far less traffic volume to encode high-entropy watermarks. We empirically validated the effectiveness of our watermarking algorithms using real-

time experiments on live IRC traffic through PlanetLab nodes and public IRC servers across different continents. Both of our watermarking approaches achieved a virtually 100% watermark detection rate and a 10^{-5} false positive rate with only a few dozen packets. To the best of our knowledge, this is the first approach that has the potential to allow real-time botmaster traceback across the Internet.

The remainder of this chapter is structured as follows: Section 5.1 introduces the botmaster traceback model. Section 5.2 presents the design and analysis of our flow watermarking schemes. Section 5.3 describes our experiments and their results, while section 5.4 discusses limitations and future work. Finally, Section 6 summarizes our findings in this chapter.

5.1 Botmaster Traceback Model

According to [23, 28, 39], most botnets currently in the wild are IRC-based. Therefore, we will focus on tracing the botmaster in the context of IRC-based botnets.

5.1.1 Botnets and Stepping Stones

Bots have been covered extensively in the existing literature, for example [2, 9, 11, 22, 28] provide good overviews. The typical bot lifecycle starts with exploitation, followed by download and installation of the bot software. At this point, the bot contacts the central C&C server run by the botmaster, where he can execute commands and receive responses from his botnet.

Botmasters rarely connect directly to their C&C servers since this would reveal their true IP address and approximate location. Instead, they use a chain of stepping stone proxies

that anonymously relay traffic. Popular proxy software used for this purpose is SSH, SOCKS, and IRC BNCs (such as psyBNC). Since the stepping stones are controlled by the attacker, they do not have an audit trail in place or other means of tracing the true source of traffic. However, there are two properties of stepping stones that can be exploited for tracing purposes: 1) the content of the message (the application-layer payload) is never modified and 2) messages are passed on immediately due to the interactive nature of IRC. Consequently, the relative lengths of messages and their timings are preserved, even if encryption is used. In the case of encryption, the message lengths are rounded up to the nearest multiple of the block size. This inherent length and timing preservation is the foundation of our botmaster traceback approach.

5.1.2 Tracking the Botmaster by Watermarking Botnet traffic

Our botmaster traceback approach exploits the fact that the communication between the IRC-based bots and the botmaster is bidirectional and interactive. Whenever the botmaster issues commands to a bot, the response traffic will eventually return to the botmaster after being laundered and possibly transformed. Therefore, if we can watermark the response traffic from a bot to the botmaster, we can eventually trace and locate the botmaster. Since the response traffic we are tracking may be mixed with other IRC traffic, we need to be able to isolate the target traffic. With unencrypted traffic, this can be achieved by content inspection, but encrypted traffic presents a challenge which we address with our hybrid length-timing algorithm.

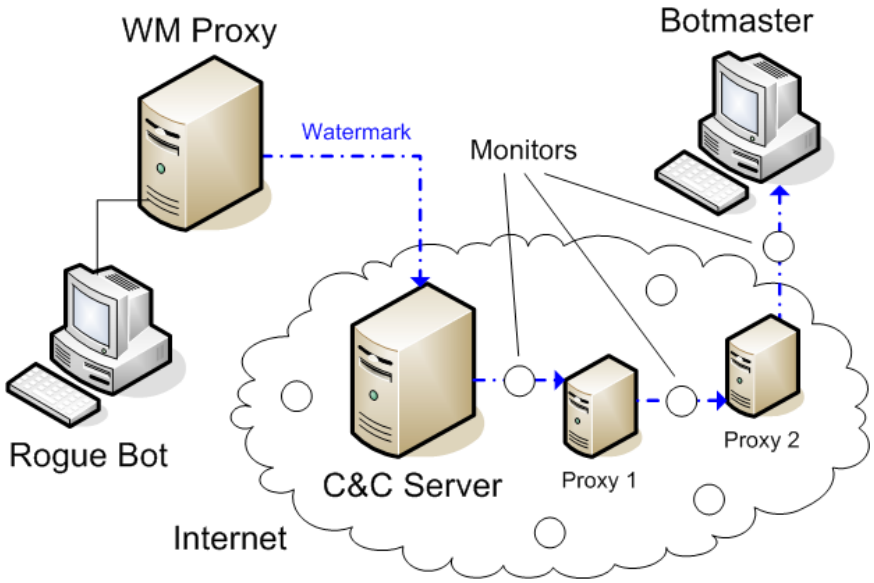

Figure 4: Botmaster traceback by watermarking botnet responses

Figure 4 shows the overall watermarking traceback model. We assume that we control a rogue bot, which could be a honeypot host that has been compromised and has joined a botnet. The rogue bot watermarks its outgoing `PRIVMSG` traffic in response to commands from the botmaster. As with any traceback approach, our watermark tracing scheme needs support from the network. Specifically, we assume there are cooperating monitor nodes across the Internet, which will inspect the passing traffic for the specified watermark and report back to us whenever they find it. Note that our approach does not require a global monitoring capability. If there are uncooperative or unmonitored areas, we would lose one or more links along the traceback path. However, we can pick up the trail again once the watermarked traffic re-enters a monitored area. In general, this appears to be the best possible approach in the absence of a global monitoring capability. We assume that the tracer can securely share the desired watermark with all monitor nodes prior to sending the watermarked traffic. This enables the monitors to report

'sightings' of the watermark in real-time and requires only a single watermarked flow to complete the trace.

5.2 Length-Based Watermarking Scheme

Our watermarking scheme was specifically designed for a low-traffic, text-based channel such as the one between a bot and its botmaster. This section describes the design and analysis of both the length-only (unencrypted traffic) and the length-timing hybrid algorithms (encrypted traffic). We describe the encoding and decoding formulas for both algorithms and address the issue of false positives and false negatives.

The terms 'message' and 'packet' are used interchangeably since they typically refer to the same unit of information in the context of IRC (each message is no longer than 512 bytes and therefore fits into a single packet). Since trailing whitespace does not show up in IRC chats, we can modify the length of the message without affecting its contents in the recipient's eyes.

5.2.1 Basic Length-Based Watermarking Scheme

5.2.1.1 Watermark Bit Encoding

Given a packet flow f of n packets $P_1,...,P_n$, we want to encode an l-bit watermark $W = w_0,...,w_{l-1}$ using $2l \leq n$ packets. We first use a pseudo-random number generator (PRNG) with seed s to randomly choose $2l$ distinct packets from $P_1,...,P_n$, we then use them to randomly form l packet pairs: $\langle P_{r_i}, P_{e_i} \rangle$ $(i = 0,...,l-1)$ such that $r_i \leq e_i$. We call packet P_{r_i} a *reference packet* and packet P_{e_i} an *encoding packet*. We further use the PRNG to

randomly assign watermark bit wk ($0 \le k \le l - 1$) to packet pair $\langle P_{r_i}, P_{e_i} \rangle$, and we use $\langle r_i, e_i, k \rangle$ to represent that packet pair $\langle P_{r_i}, P_{e_i} \rangle$ is assigned to encode watermark bit w_k.

To encode the watermark bit w_k into packet pair $\langle P_{r_i}, P_{e_i} \rangle$, we modify the length of the encoding packet P_{e_i} by adding whitespace padding to achieve a specific length difference to its corresponding reference packet P_{r_i}. Let l_e and l_r be the packet lengths of the watermark encoding and reference packets respectively, $Z = l_e - l_r$ be the length difference, and $L > 0$ be the bucket size. We define the *watermark bit encoding function* as follows:

$$e(l_r, l_e, L, w) = l_e + [(0.5 + w)\ L - (l_e - l_r)]\ mod\ 2L \tag{2}$$

This equation returns the increased length of watermark encoding packet given the length of the reference packet l_r, the length of the encoding packet l_e, the bucket size L, and the watermark bit to be encoded w.

Therefore,

$$\begin{aligned} & (e(l_r, l_e, L, w) - l_r)\ mod\ 2L \\ &= \{(l_e - l_r) + [(0.5 + w)L - (l_e - l_r)]\ mod\ 2L\}\ mod\ 2L \\ &= \{(0.5 + w)L\}\ mod\ 2L \\ &= (w + 0.5)\ L \end{aligned} \tag{3}$$

This indicates that the packet length difference $Z = l_e - l_r$, after l_e is adjusted by the watermark bit encoding function $e(l_r, l_e, L, w)$, falls within the middle of either an even or odd numbered bucket depending on whether the watermark bit w is even or odd.

5.2.1.2 Watermark Bit Decoding

Assuming the decoder knows the watermarking parameters: PRNG, s, n, l, W and L, the watermark decoder can obtain the exact pseudo-random mapping $\langle r_i, e_i, k\rangle$ as that used by the watermark encoder. We use the following *watermark bit decoding function* to decode watermark bit w_k from the packet lengths of packets P_{r_i} and P_{e_i}

$$d(l_r, l_e, L) = \lfloor \frac{l_e - l_r}{L} \rfloor \ mod\ 2 \tag{4}$$

The equation below proves that any watermark bit w encoded by the encoding function defined in equation (2) will be correctly decoded by the decoding function defined in equation (4).

$$\begin{aligned} & d(l_r, e(l_r, l_e, L, w), L) \\ &= \lfloor \frac{e(l_r, l_e, L, w)}{L} \rfloor \ mod\ 2 \\ &= \lfloor \frac{(l_e - l_r) mod\ 2L + [(0.5+w)L - (l_e - l_r)]}{L} \rfloor \ mod\ 2 \\ &= \lfloor \frac{(0.5+w)L}{L} \rfloor \ mod\ 2 \\ &= w \end{aligned} \tag{5}$$

Assume the lengths of packets P_r and P_e (l_r and l_e) have been increased for $x_r \geq 0$ and $x_e \geq 0$ bytes respectively when they are transmitted over the network (e.g., due to padding of encryption), then $x_e - x_r$ is the distortion over the packet length difference $l_e - l_r$. Then the decoding with such distortion is

$$
\begin{aligned}
& d(l_r + x_r, e(l_r, l_e, L, w) + x_e, L) \\
&= \lfloor \frac{e(l_r, l_e, L, w) - l_r + (x_e - x_r)}{L} \rfloor \bmod 2 \\
&= w + \lfloor 0.5 + \frac{x_e - x_r}{L} \rfloor \bmod 2
\end{aligned}
\tag{6}
$$

Therefore, the decoding with distortion will be correct if and only if

$$
(-0.5 + 2i)L \le x_e - x_r < (0.5 + 2i)L \tag{7}
$$

Specifically, when the magnitude of the distortion $| x_e - x_r | < 0.5L$, the decoding is guaranteed to be correct.

5.2.1.3 Watermark Decoding and Error Tolerance

Given a packet flow f and appropriate watermarking parameters (PRNG, s, n, l, W and L) used by the watermark encoder, the watermark decoder can obtain a l-bit decoded watermark W' using the watermark bit decoding function defined in equation (4). Due to potential distortion of the packet lengths in the packet flow f, the decoded W' could have a few bits different from the encoded watermark W. We introduce a Hamming distance threshold $h \ge 0$ to accommodate such partial corruption of the embedded watermark. Specifically, we will consider that packet flow f contains watermark W if the Hamming distance between W and W': $H(W, W')$ is no bigger than h.

5.2.1.4 Watermark Collision Probability (False Positive Rate)

No matter what watermark W and Hamming distance threshold h we choose, there is always a non-zero possibility that the decoding W' of a random unwatermarked flow

happens to have no more than h Hamming distance to the random watermark W we have chosen. In other words, watermark W is reported found in an unwatermarked flow; we refer to this case as a *watermark collision*.

Intuitively, the longer the watermark and the smaller the Hamming distance threshold, the smaller the probability of a watermark collision. Assume we have randomly chosen a l-bit watermark, and we are decoding l-bits from random unwatermarked flows. Any particular bit decoded from a random unwatermarked flow should have 0.5 probability to match the corresponding bit of the random watermark we have chosen. Therefore, the collision probability of l-bit watermark from random unwatermarked flows with Hamming distance threshold h is

$$\sum_{i=0}^{h} \binom{l}{i} \left(\frac{1}{2}\right)^{l} \tag{8}$$

We have empirically validated the watermark collision probability distribution with the following experiment. We first use a PRNG and a random seed number s to generate 32 packet pairs $\langle r_i, e_i \rangle$ and pseudo-randomly assign each bit of a 32-bit watermark W to the 32 packet pairs, we then encode the 32 bit watermark W into a random packet flow f. Now we try to decode the watermarked flow f with 1,000 wrong seed numbers. Given the pseudo-random nature of our selection of the packet pairs, decoding a watermarked flow with the wrong seed is equivalent of decoding an unwatermarked flow, which can be used to measure the watermark collision probability.

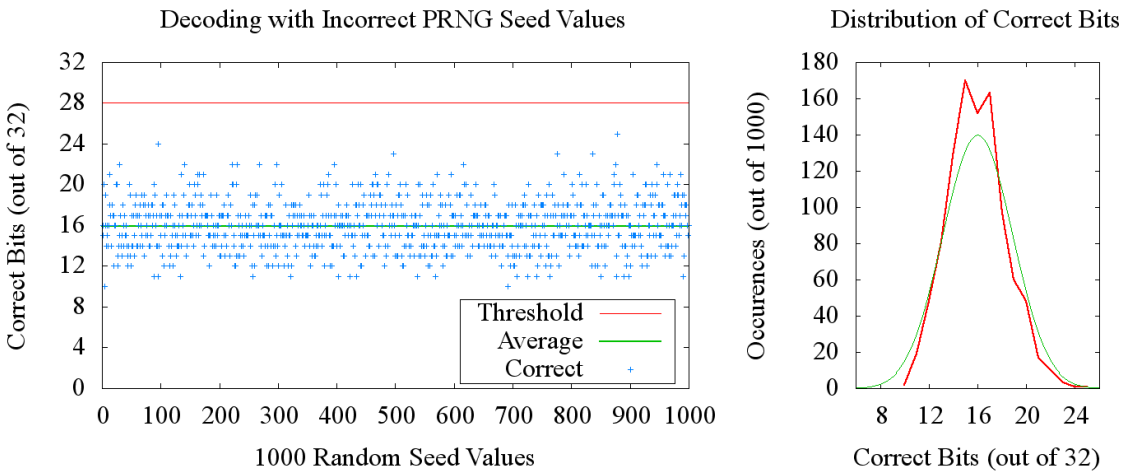

Figure 5: 32-bit watermark collision probability and distribution

The left side of Figure 5 illustrates the number of matched bits from the decoding with each of the 1,000 wrong seed numbers. It shows that the numbers of matched bits are centered around the expected value of 16 bits, which is half of the watermark length. Based on these results and the experimental data in Section 5.3.2, we can choose a Hamming distance threshold of $h = 4$ (28 bits) as shown on the graph, yielding an expected false positive rate (FPR) of 9.64×10^{-6} according to equation (8). The right side of Figure 5 shows the distributions of the measured and the expected number of matched bits. It illustrates that the distribution of the measured number of matched bits is close to the expected binomial distribution with $p = 0.5$ and $n = 32$.

5.2.1.5 Watermark Loss (False Negative)

Our length-only encoding scheme (without the hybrid timing approach) is highly sensitive to having the correct sequence of messages. If any messages are added or deleted in transit, the watermark will be lost in that flow. However, the chance of this

happening is very remote since the encoding takes place at the application layer, on top of TCP. By its nature, TCP guarantees in-order delivery of all packets and their contents, so a non-intentional watermark loss is very unlikely.

In the case of active countermeasures, our scheme can tolerate distortion as long as $| x_e - x_r | < 0.5L$, as described by inequality (7). This property is the result of aiming for the center of each bucket when encoding. However, if an active adversary drops, adds, or reorders messages, the watermark will be lost unless additional redundancy is in place or the length-timing algorithm is used.

5.2.2 Hybrid Length-Timing Watermarking for Encrypted traffic

By their nature, IRC-based botnets have many bots on one channel at once, many of them joining, parting, or sending data to the botmaster simultaneously. In this case, the watermarked messages from our rogue bot will be mixed with unwatermarked messages from other bots. We call these unwatermarked messages from others *chaff* messages. In order to reliably decode the embedded watermark, we need to filter out chaff messages as much as possible.

When the C&C traffic is unencrypted, it is easy for the watermark decoder to filter out chaff based on the sender nicks in the messages. However, if the traffic is encrypted (e.g., using SSL/TLS), we cannot rely on content inspection to identify chaff messages. To address this new challenge in filtering out chaff, we propose to use another dimension of information – the packet timing – to filter out chaff.

The basic idea is to send the watermark encoding packets at a specific time (e.g., t_i). Assuming the network jitter is limited, we can narrow the range of potential packets used for decoding to $[t_{e_i}-\frac{\delta}{2}, t_{e_i}+\frac{\delta}{2}]$. If $\delta > 0$ is small, then the chances that some chaff packet happens to fall within the range $[t_{e_i}-\frac{\delta}{2}, t_{e_i}+\frac{\delta}{2}]$ is small. This means we can decode the watermark correctly even if there are substantial encrypted chaff packets.

5.2.2.1 Watermark Encoding

The watermark bit encoding process is exactly the same as that of the basic length-based watermarking scheme. The difference is that now we send out each watermarked packet P_{e_i} at a precise time. Specifically, we use the watermark bit encoding function defined in equation (2) to adjust the length of the watermark encoding packet P_{e_i}. We use a pseudo-random number generator PRNG and seed s_t to generate the random time t_{e_i} at which P_{e_i} will be sent out.

An implicit requirement for the hybrid length-timing watermarking scheme is that we need to know when each watermark encoding packet P_{e_i} will be available. In our watermark tracing model, the tracer owns a rogue bot who can determine what to send out and when to send it. Since we have full control over the outgoing traffic, we can use the hybrid length-timing scheme to watermark the traffic in real-time.

5.2.2.2 Watermark Decoding

When we decode the encrypted botnet traffic, we do not know which packet is a watermark encoding packet P_{e_i}. However, given the PRNG and s_t we do know the approximate time t_{e_i} at which the watermark encoding packet P_{e_i} should arrive. We then use all packets in the time interval $[t_{e_i}-\frac{\delta}{2}, t_{e_i}+\frac{\delta}{2}]$ to decode. Specifically, we use the sum of the lengths of all the packets in the time interval $[t_{e_i}-\frac{\delta}{2}, t_{e_i}+\frac{\delta}{2}]$ as the length of the watermark encoding packet and apply that to the watermark bit decoding function (4).

Due to network delay jitter and/or active timing perturbation by the adversary, the exact arrival time of watermark encoding packet P_{e_i} may be different from t_{e_i}. Fortunately, the decoding can self-synchronize with the encoding by leveraging an intrinsic property of our hybrid length-timing watermarking scheme. Specifically, if the decoding of a watermarked flow uses the wrong offset or wrong seeds (s and s_t), then the decoded l-bit watermark W' will have about $\frac{l}{2}$ bits matched with the true watermark W on average. This gives us an easy way to determine if we are using the correct offset, and we can try a range of possible offsets and pick the best decoding result (see Section 5.3.2.2).

5.3 Implementation and Experiment

To validate the practicality of our watermarking scheme, we implemented both the length-only algorithm (unencrypted traffic) and the length-timing hybrid algorithm (encrypted traffic). To let our watermarking proxy interact with a realistic but benign IRC

bot, we obtained a sanitized version of Agobot from its source code, containing only benign IRC communication features. We ran the sanitized Agobot on a local machine to generate benign IRC traffic to test the effectiveness of our watermarking scheme across public IRC servers and PlanetLab nodes. At no time did we send malicious traffic to anyone in the course of our experiments.

5.3.1 Length-Only Algorithm (Unencrypted traffic)

We implemented the length-only algorithm in a modified open-source IRC proxy server and ran a series of experiments using the sanitized Agobot and public Internet IRC servers. We were able to recover the watermark successfully from unencrypted traffic in all ten of our runs.

5.3.1.1 Modified IRC Bouncer

To achieve greater flexibility, we added our watermarking functionality to an existing IRC bouncer (BNC) package, psyBNC. Having the watermarking implemented on a proxy server allows us to use it on all bots conforming to the standard IRC protocol. It eliminates the need to have access to a bot's source code to add the watermarking functionality: outgoing traffic is modified by the BNC after the bot sends it.

In order for psyBNC to act as a transparent proxy, it needs to be configured identically to the bot. The information required consists of the C&C server's hostname, the port, and an IRC nick consistent with the bot's naming scheme. This information can be gathered by running the bot and monitoring the outgoing network traffic. In order to trick the bot into

connecting to the BNC rather than to the real C&C host, we also need to update our local DNS cache so that a lookup of the C&C server's hostname resolves to the IP of our BNC.

Once it has been configured with this information, the BNC is completely transparent to the bot: when it starts up, the bot is automatically signed into the real C&C server by the BNC. The bot now joins the botnet channel as if it were directly connected and then waits for the botmaster's instructions. All `PRIVMSG` traffic from the bot to the C&C server (and by extension, to the botmaster) is watermarked by the transparent BNC in between.

5.3.1.2 Experiment and Results

To test our watermarking scheme, we devised an experiment that emulates the conditions of an Internet-wide botnet as closely as possible. To simulate the botmaster and stepping stones, we used PlanetLab nodes in California and Germany. We used a live, public IRC server in Arizona to act as a C&C host, creating a uniquely-named channel for our experiments. Our channel consisted of two IRC users: the Test Bot was running a copy of the sanitized Agobot and the Botmaster was acting as the botmaster (see Figure 6). As the diagram indicates, all traffic sent by the Test Bot passes through the psyBNC server (WM Proxy) where the watermark is injected. The distances involved in this setup are considerable: the watermarked traffic traverses literally half the globe (12 time zones) before reaching its ultimate destination in Germany, with a combined round-trip time of 292 milliseconds on average (at the time of our experiment).

The objective is to be able to decode the full watermark in the traffic captured at the Stepping Stone and Botmaster. Since only `PRIVMSG` traffic from the Test Bot is

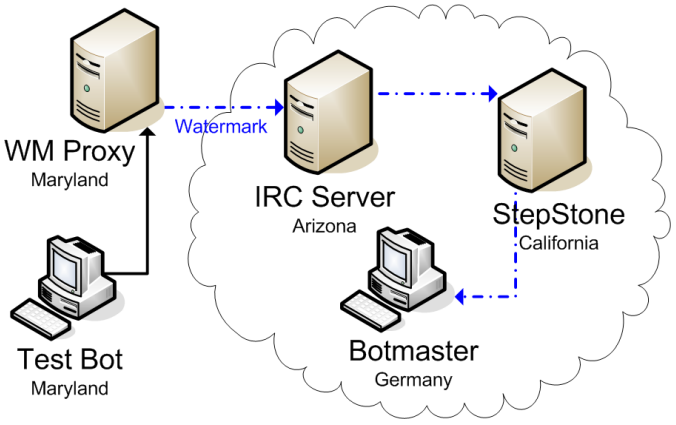

Figure 6: Experiment setup for unencrypted traffic

watermarked, all other traffic (chaff) must be filtered out before decoding. Most of this chaff consists of messages from other users on the channel, PING/PONG exchanges, and JOIN/PART notifications from the channel. There could be additional chaff on the same connection if the botmaster is logged into multiple channels on the same IRC server. However, filtering out the chaff is trivial in the absence of encryption since all IRC messages contain the sender's nick. Therefore, we can easily isolate the watermarked packets based on the Test Bot's nick.

During our experiments, the psyBNC proxy was configured to inject a 32-bit watermark into a 64-packet stream. To generate traffic from the Test Bot, the Botmaster logged in and issued the `commands.list` command, causing the bot to send a list of all valid bot commands and their descriptions. We captured all traffic leaving the WM Proxy, arriving at the Stepping Stone, and arriving at the Botmaster. In ten runs with different (random) 32-bit watermarks, we were able to correct decode the full 32-bit watermark at all three monitoring locations: the WM Proxy in Maryland, the Stepping Stone in California, and Botmaster in Germany.

5.3.2 Hybrid Length-Timing Algorithm (Encrypted traffic)

To test the hybrid length-timing algorithm, we implemented a simple IRC bot that sends length-watermarked messages out at specific intervals. We used a "chaff bot" on the channel to generate controlled amounts of chaff. We were able to recover the watermark with a high success rate, even when high amounts of chaff were present.

5.3.2.1 Hybrid Length-Timing Encoder

We implemented the hybrid encoding algorithm as a Perl program which reads in a previously length-only watermarked stream of messages and sends them out at specific times. To achieve highly precise timing, we used the `Time::HiRes` Perl package, which provides microsecond-resolution timers. At startup, the program uses the Mersenne Twister PRNG (via the `Math::Random::MT` package) to generate a list of departure times for all messages to be sent. Each message is sent at a randomly chosen time between 2 and 2.35 seconds after the previous message. The 2-second minimum spacing avoids IRC server packet throttling (more details are discussed in Section 5.3.2.3).

5.3.2.2 Hybrid Length-Timing Decoder

The hybrid decoding script was also written in Perl, relying on the PCAP library to provide a standardized network traffic capture mechanism (via the `Net::Pcap` module). The program reads in a stream of packets (either from a live interface or from a PCAP file), then performs a sliding-window offset self-synchronization process to determine the time *t1* of the first watermarked packet. To find the correct *t1*, the program steps through a range of possible values determined by the *offset*, *max*, and *step* parameters. It starts

with *t1* = *offset*, incrementing *t1* by *step* until *t1* = (*offset* + *max*). It decodes the full watermark sequence for each *t1*, recording the number of bits matching the sought watermark *W*. It then chooses the *t1* that produced the highest number of matching bits. If there are multiple *t1* values resulting in the same number of matching bits, it uses the lowest value for *t1*. Figure 7 illustrates the synchronization process, showing that the correct *t1* is near 6 seconds: 5.92 sec has 32 correct bits. For all incorrect *t1* values, the decoding rate was significantly lower, averaging 14.84 correct bits.

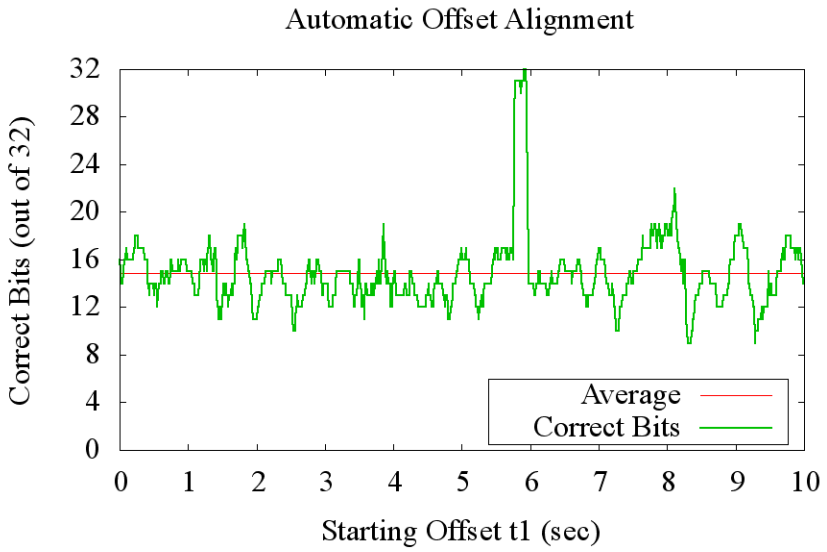

Figure 7: Offset self-synchronization via sliding window alignment

5.3.2.3 Experiment and Results

The experiment setup in this case was similar to the unencrypted experiment described in Section 5.3.1. The three main differences were: 1) a single Source computer producing watermarked traffic on its own replaced the Test Bot and WM Proxy; 2) the connection between the Botmaster and the IRC server (via StepStone) was encrypted using

SSL/TLS; and 3) we used a different IRC server because the one in Arizona does not support SSL/TLS connections. The IRC server in this case happens to be located in Germany, but not in the same place as the Botmaster. Please refer to Figure 8 for the full experiment setup. In this configuration, the distances involved are even greater, with the watermarked traffic traversing the equivalent of the entire globe (24 time zones). The combined round-trip time from Source to Botmaster was 482 milliseconds (on average) at the time of our experiment.

To handle encryption, the parameters for the length-only algorithm were adjusted to ensure that the bucket size matched or exceeded the encryption block size. Most SSL/TLS connections use a block size of 128 bits (16 bytes), though 192 and 256 bits are also common. To ensure that each added bucket also causes another encrypted block to be added to the message, the bucket size has to be greater than or equal to the block size. For our experiment, we used a bucket size of 16 bytes, which was sufficient for the 128-bit block size used in the SSL/TLS connection. For compatibility with the larger block sizes (192 and 256 bits), a bucket size of 32 bytes can be used.

For the experiments, the Source produced a stream of 64 packets, containing a randomly generated 32-bit watermark. The Chaff Bot produced a controlled amount of background traffic, spacing the packets at random intervals between 1 and 6 seconds (at least 1 second to avoid throttling). In addition to our Control run (no chaff), we ran five different chaff levels (Chaff 1 to 5). The number refers to the maximum time between packets (not including the minimum 1-second spacing). For example, for the Chaff 1 run, packets were sent at a random time between 1 and 2 seconds. Thus, one packet was sent on

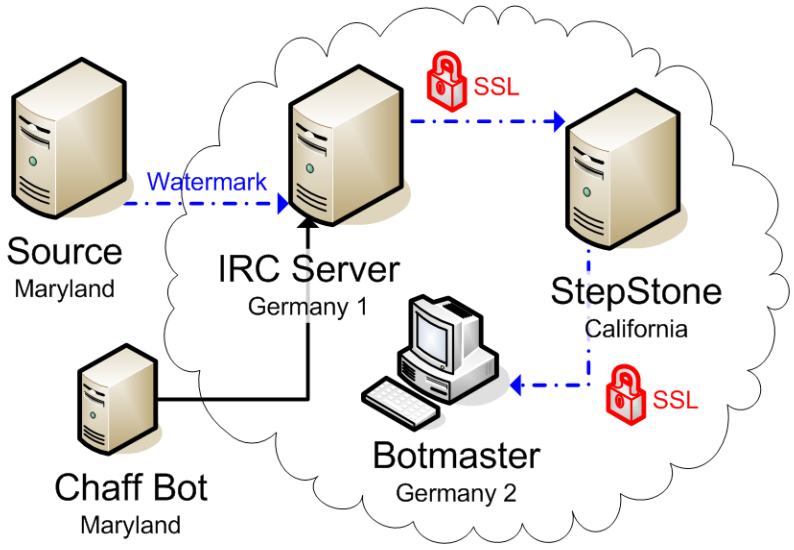

Figure 8: Experiment setup for encrypted traffic

average every 1.5 seconds, resulting in a chaff rate of approximately 1/1.5 = 0.667 packets/ sec. The Chaff 2 run had a chaff rate of about 1/2 = 0.5 packets/sec, and so on.

We captured network traffic in three places: 1) traffic from Source and Chaff Bot to IRC Server; 2) traffic arriving at StepStone from IRC Server; and 3) traffic arriving at Botmaster from StepStone. traffic in all three locations includes both watermark and chaff packets. We decoded the traffic at each location, recording the number of matching bits. For decoding, we used a value of 200 milliseconds for the timing window size δ and a sliding offset range from 0 to 10 seconds. This δ value was large enough to account for possible jitter along the stepping stone chain but small enough to make it unlikely that a chaff packet appears within δ of an encoding packet. We also measured the actual chaff rate based on the departure times of each chaff packet, and these were very close to the expected rates based on an even distribution of random departure times. We repeated this process three times for each chaff level, resulting in a total of 18 trials. Our experiment

results are summarized in Table 1, with each column representing the average number of recovered watermark bits (out of 32) from three runs.

Table 1: Averaged experiment results for encrypted traffic

Monitoring Location	Chaff 1	Chaff 2	Chaff 3	Chaff 4	Chaff 5	Control
Chaff Rate (packets/sec)	0.6719	0.4976	0.4274	0.3236	0.2872	no chaff
Source - Maryland	29.67	30.33	29.67	30.33	30.33	32
StepStone - California	31	32	31.67	31.67	32	32
Botmaster - Germany	31	31.67	32	31.67	31.67	32

We had near-perfect decoding along the stepping-stone chain for all chaff rates of 0.5 packets/sec and below. Only when the chaff rate rose above 0.5 packets/sec did the chaff start having a slight impact, bringing the decoding rate down to an average of 31 bits. The overall average decoding rate at the StepStone and Botmaster was 31.69 bits, or 99.05 percent. The lowest recorded decoding rate during our experiments was 28 bits, so we can use a Hamming distance threshold of $h = 4$ to obtain a 100 percent true positive rate (TPR) and a false positive rate (FPR) of 9.64×10^{-6}.

The most surprising result is that in all cases where chaff was present, the decoding rate was worse at the Source than downstream at the StepStone and Botmaster. After examining the network traces in detail, we realized that this behavior was due to the presence of traffic queuing and throttling on the IRC Server. To avoid flooding, IRC servers are configured to enforce minimum packet spacings, and most will throttle traffic at 0.5 to 1 packets/sec. To confirm this behavior, we sent packets to the IRC Server in Germany at random intervals of 100 to 300 milliseconds. For the first 5 seconds, packets were passed on immediately, but after that the throttling kicked in, limiting the server's

outgoing rate to 1 packet/sec. After about 2 minutes, the server's packet queue became full with backlogged packets, and it disconnected our client. Figure 9 illustrates the effect of throttling on the packet arrival times, including the 5-second "grace period" at the beginning.

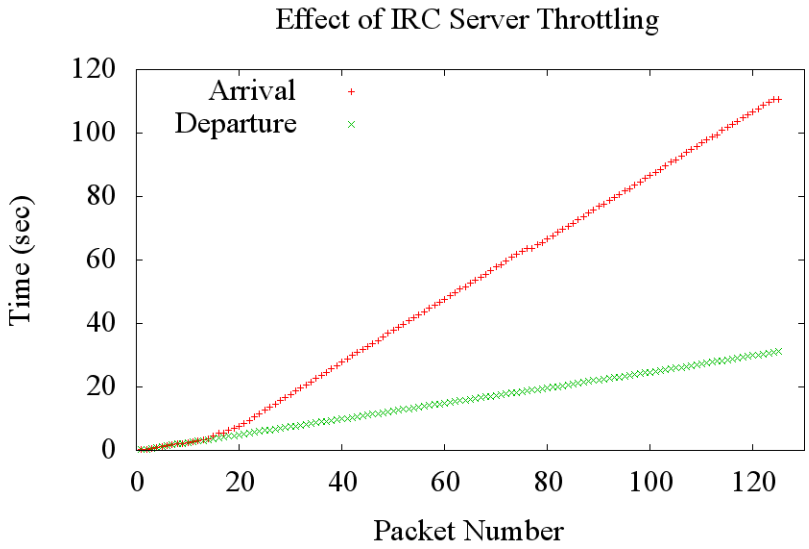

Figure 9: IRC server throttling causes packets to be spaced apart further upon arrival

In the context of our hybrid encoding scheme, IRC message queuing is highly beneficial because it dramatically reduces the chances that chaff and encoding packets will appear close to each other. At the Source, packets appear at the exact intervals they are sent, which could be less than δ and therefore affect decoding. However, this interval will be increased due to queuing by the IRC server. By the time the packets reach the StepStone and Botmaster, they no longer affect decoding because they are more than δ apart. In our experiments, we observed that the IRC server introduced a distance of at about 130

milliseconds between packets due to queuing. Since our δ value was 200 milliseconds, this made it unlikely that two packets would arrive in the same slot.

5.4 Discussion and Future Work

Our experiments show that our watermarking scheme is well-suited to IRC-based botnets, which are still the predominant type in the wild [23, 28, 39]. Even when encryption is present in the botmaster's chain of stepping stones, our watermark can be recovered with a high degree of accuracy.

While peer-to-peer (P2P) botnets have started appearing recently [19], most in-the-wild botnets are still based on the centralized topology. Within that realm, the majority of botnets are based on the IRC protocol [23, 28, 39], but HTTP botnets have been gaining some ground recently. Unfortunately, HTTP-based botnets present a much higher level of traceback difficulty: the messages do not get passed from the bot to the botmaster in real-time. They are typically stored on the C&C server until the botmaster retrieves them in bulk, usually over an encrypted connection such as SSH. Due to this, any approach that relies on properties of individual packets (such as length and timing) will be unsuccessful.

When SSH is used as the final hop in a chain of stepping stones, it presents unique challenges. In this case, the botmaster uses SSH to log into a stepping stone, launches a commandline-based IRC client on that host, and uses this IRC client to connect to his botnet (possibly via more stepping stones). In this capacity, SSH is not acting as a proxy, passing on messages verbatim like psyBNC or SOCKS. Instead, it transfers the

"graphical" screen updates of the running IRC client, which is not necessarily correlated to the incoming IRC messages. This situation is challenging for our approach because the application-layer content is transformed, altering the relative lengths of packets. We are working on this problem, but we have been unable to explore it in detail. Notice that if SSH is used in a tunneling capacity (such as port forwarding or a SOCKS proxy) in the middle of a stepping stone chain, this limitation does not apply.

As previously discussed in Section 5.1.2, our approach requires at least partial network coverage of distributed monitoring stations. This is a common requirement for network traceback approaches, especially since the coverage does not need to be global. The accuracy of the trace is directly proportional to the number and placement of monitoring nodes.

Our work is a significant step in the direction of live botmaster traceback, but it is in fact only a first step. Our future work in this area includes the exploration of several topics, including optimal deployment of monitoring nodes, SSH traffic on the last hop, further data collection with longer stepping stone chains, and traceback experiments on in-the-wild botnets.

6 Conclusion

Our early work in breaking botnet C&C channels by leveraging botnet design constraints laid the foundation for further exploration of the traceback possibilities of C&C channels. The work on precision timing in a VM environment allowed us to run precision time-based watermarking code on virtual machines rather than requiring a dedicated physical host. These two projects built the background knowledge required to take on one of the most challenging aspects of botnets: real-time botmaster traceback across the Internet.

The key contribution of our work is that it addresses the four major obstacles in botmaster traceback: 1) stepping stones, 2) encryption, 3) flow mixing and 4) a low traffic volume between bot and botmaster. Our watermarking traceback approach is resilient to stepping stones and encryption, and it requires only a small number of packets to embed a high-entropy watermark into a network flow. The watermarked flow can be tracked even when it has been mixed with arbitrary chaff traffic. Due to these characteristics, our approach is uniquely suited for real-time tracing of the low-traffic IRC link between a bot and its botmaster. We believe that this is the first viable technique for performing live botmaster traceback on the Internet.

We validated our watermarking traceback algorithms both analytically and experimentally. In trials on public Internet IRC servers, we were able to achieve virtually a 100

percent TPR with an FPR of less than 10^{-5}. Our method can successfully trace a watermarked IRC flow from an IRC botnet member to the botmaster's true location, even if the watermarked flow 1) is encrypted with SSL/TLS; 2) passes through several stepping stones; and 3) travels tens of thousands of miles around the world.

Appendix A: Implementation Source Code

A.1 Length-Only Encoder Source Files

This section lists the full source code of the files required to build the length-only encoder and decoder programs. The code is written in C and was compiled using GCC. It relies on the GNU Scientific Library (GSL) for the Mersenne Twister PRNG.

A.1.1 ldb.h

```
#include <gsl/gsl_rng.h>

typedef struct {
    short int ref;
    short int enc;
    short int len;
    short int bit;
    short int used;
} bitpair;

void generatePairs(bitpair * ppairs, unsigned long int seed, int packets, int bits);
```

A.1.2 ldb.c

```
#include <string.h>
#include <gsl/gsl_rng.h>

#include "ldb.h"

void generatePairs(bitpair * ppairs, unsigned long int seed, int packets, int bits) {

    int i, rand1, rand2, high, low;

    // initialize the array of pairs
    for(i = 0; i < packets; i++) {
        bitpair * bp = &ppairs[i];
        bp->ref = -1;
        bp->enc = -1;
        bp->len = -1;
        bp->bit = -1;
        bp->used = 0;
    }

    // initialize the PRNG
    gsl_rng * r = gsl_rng_alloc(gsl_rng_mt19937);
```

```
    gsl_rng_set(r, seed);

    // pick two random numbers per encoding bit
    for(i = 0; i < bits; i++) {
        do {
            rand1 = gsl_rng_uniform_int(r, packets);
        } while(ppairs[rand1].used != 0);
        ppairs[rand1].used = 1; // IMPORTANT: set this first so we don't pick rand1 ==
rand2

        do {
            rand2 = gsl_rng_uniform_int(r, packets);
        } while(ppairs[rand2].used != 0);
        ppairs[rand2].used = 1;

        // check which random number is lower
        if(rand1 > rand2) {
            high = rand1;
            low = rand2;
        } else if(rand1 < rand2) {
            high = rand2;
            low = rand1;
        } else {
            // ERROR!
        }

        // initialize both data structures (bp->used is already set)
        bitpair * bp = &ppairs[low];
        bp->ref = low;
        bp->enc = high;
        bp->bit = i;

        bp = &ppairs[high];
        bp->ref = low;
        bp->enc = high;
        bp->bit = i;

        printf("Bit %02d: Choosing pair (ref %d, enc %d)\n", bp->bit, bp->ref, bp->enc);
    }

    return;
}
```

A.1.3 ldb-encoder.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <error.h>

#include "ldb.h"

#define MTU 510
#define BUCKET_SIZE 16
#define PACKET_COUNT 64
#define WM_MAX_LENGTH 32
#define PRNG_SEED 1234
#define PAD_CHAR ' '

int main(int argc, char * argv[]) {
    int i, wm_len;
    char * watermark;

    bitpair * ppairs = (bitpair *) malloc(sizeof(bitpair) * PACKET_COUNT);
```

```
    if(argc != 3) {
        printf("\nUsage: %s wm_bits wm_file\n\n\twm_bits - watermark bits to
encode\n\twm_file - input file containing strings to be watermarked\n\n", argv[0]);
        exit(1);
    }

    watermark = argv[1];
    wm_len = strlen(watermark);
    if(wm_len > WM_MAX_LENGTH) {
        error(0, 0, "The given watermark is longer than the maximum number of WM bits
(%d).", WM_MAX_LENGTH);
        exit(1);
    }

    generatePairs(ppairs, PRNG_SEED, PACKET_COUNT, wm_len);

    // Read input strings line-by-line
    // Append the proper number of watermarking characters
    FILE *file = fopen (argv[2], "r");
    if(file != NULL) {
        char line[MTU + 1];
        int len, prev_len = 0;

        for(i = 0; i <= PACKET_COUNT && fgets(line, MTU, file) != NULL; i++) {
            len = strlen(line) - 1; // subtract 1 due to newline

            bitpair * bp = &ppairs[i];
            char bit_char = (char) watermark[bp->bit];
            int bit = (int) (bit_char - 48);

            // only pad if this is an encoding bit
            if(i == bp->enc) {

                bp->len = len; // store length of packet
                prev_len = ppairs[bp->ref].len; // get lenth of reference packet

                // incremental approach
                while(
                    (((abs(len - prev_len) / BUCKET_SIZE) % 2) != bit ||
                    abs(len - prev_len) % BUCKET_SIZE != 0) &&
                    len < MTU - 2) {
                    line[len++] = PAD_CHAR;
                }

                if(len >= MTU - 2) {
                    error(0, 0, "WARNING: MTU exceeded!");
                }

                line[len] = '\n';
                line[len+1] = '\0';

                //printf("Bit %02d (enc, val = %d): Packet %d, length = %d, diff = %d\n",
bp->bit, bit, i, len, len - prev_len);
                //printf("Bit %02d (enc, val = %d): Packet %d, length = %d, diff = %d: ",
bp->bit, bit, i, len, len - prev_len);
                fputs(line, stdout);

            // do not modify content if this is a reference bit
            } else if(i == bp->ref) {
                bp->len = len; // store length of packet

                line[len] = '\n';
                line[len+1] = '\0';

                //printf("Bit %02d (ref): Packet %d, length = %d\n", bp->bit, i, len);
                //printf("Bit %02d (ref, val = %d): Packet %d, length = %d: ", bp->bit,
bit, i, len);
```

```
                fputs(line, stdout);
            }

            //fprintf(stderr, "%d\n", len); // dump (encoded) lengths to STDERR
        }
        fclose(file);

    } else {
        perror(argv[2]);
    }

    free(ppairs);
    return 0;
}
```

A.1.4 ldb-decoder.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <error.h>

#include "ldb.h"

#define MTU 510
#define BUCKET_SIZE 16
#define PACKET_COUNT 64
#define WM_MAX_LENGTH 32
#define PRNG_SEED 1234
#define PAD_CHAR ' '

int main(int argc, char * argv[]) {
    int i, wm_len;
    char * watermark;
    char recovered[WM_MAX_LENGTH+1];

    bitpair * ppairs = (bitpair *) malloc(sizeof(bitpair) * PACKET_COUNT);

    if(argc < 3) {
        printf("\nUsage: %s wm_bits wm_file [-v]\n\n\twm_bits - watermark bits to
decode\n\twm_file - input file containing watermarked strings\n\t-v - verbose output
(recovered WM rather than number of correct bits)\n\n", argv[0]);
        exit(1);
    }

    watermark = argv[1];
    wm_len = strlen(watermark);
    if(wm_len > WM_MAX_LENGTH) {
        error(0, 0, "The given watermark is longer than the maximum number of WM bits
(%d).", WM_MAX_LENGTH);
        exit(1);
    }

    generatePairs(ppairs, PRNG_SEED, PACKET_COUNT, wm_len);

    // Read input strings line-by-line
    FILE *file = fopen (argv[2], "r");
    if(file != NULL) {
        char line[MTU + 1];
        int len, prev_len = 0;

        for(i = 0; i <= PACKET_COUNT && fgets(line, MTU, file) != NULL; i++) {
            len = atoi(line);

            bitpair * bp = &ppairs[i];
```

```
            //char bit_char = (char) watermark[bp->bit];
            //int bit = (int) (bit_char - 48);

            // only pad if this is an encoding bit
            if(i == bp->enc) {

                bp->len = len; // store length of packet
                prev_len = ppairs[bp->ref].len; // get lenth of reference packet

                // get observed bit value
                int bit = (abs(len - prev_len) / BUCKET_SIZE) % 2;

                /*if(len >= MTU - 2) {
                    error(0, 0, "WARNING: MTU exceeded!");
                }*/

                recovered[bp->bit] = (char) (bit + 48);
                //printf("Bit %02d (enc, val = %d): Packet %d, diff = %d\n", bp->bit, bit,
i, len - prev_len);
                //printf("Bit %02d (enc, val = %d): Packet %d, length = %d, diff = %d\n",
bp->bit, bit, i, len, len - prev_len);

            // no action needed if this is a reference bit
            } else if(i == bp->ref) {
                bp->len = len; // store length of packet

                //printf("Bit %02d (ref): Packet %d\n", bp->bit, i);
                //printf("Bit %02d (ref): Packet %d, length = %d\n", bp->bit, i, len);
            }
        }
        fclose(file);

        recovered[wm_len] = '\0';
        //printf("\n Expected watermark (%d bits): %s\n", wm_len, watermark);
        //printf("Recovered watermark (%d bits): %s\n", wm_len, recovered);

        int correct = 0;
        for(i = 0; i < wm_len; i++) {
            if(watermark[i] == recovered[i]) correct++;
        }

        //printf("\nCorrect bits: %d\n\n", correct);
        if(argc > 3 && strcmp(argv[3],"-v") == 0) {
            printf("%s (%d bits correct)\n", recovered, correct);
        } else {
            printf("%d\n", correct);
        }

    } else {
        perror(argv[2]);
    }

    return 0;
}
```

A.1.5 Makefile

```
LIBS=-lgsl -lgslcblas

all: ldb-encoder ldb-decoder

ldb-encoder: ldb-encoder.c ldb.o
    gcc $(LIBS) ldb-encoder.c ldb.o -o ldb-encoder

ldb-decoder: ldb-decoder.c ldb.o
```

```
    gcc $(LIBS) ldb-decoder.c ldb.o -o ldb-decoder

ldb.o: ldb.h ldb.c
    gcc -c ldb.c

clean:
    rm -rf *.o ldb-encoder ldb-decoder
```

A.2 Length-Timing Hybrid Source Files

This section contains the Perl scripts used to implement the automated encoding and decoding using the hybrid length-timing algorithm. They rely on several standard Perl modules that are available through CPAN: `Time::HiRes`, `Math::Random:MT`, `IO::Socket::INET`, `IO::Socket::SSL`, and `Net::PCAP`.

A.2.1 encoder.pl

```
#! /usr/bin/perl

use Time::HiRes qw( ualarm gettimeofday );
use Math::Random::MT qw( srand rand );
use IO::Socket::INET;

# timing constants
my $USEC_PER_MSEC = 1000;
my $MSEC_PER_SEC = 1000;
my $USEC_PER_SEC = ($USEC_PER_MSEC * $MSEC_PER_SEC);

# IRC parameters
my $IRC_SERVER = 'X.X.X.X';
my $IRC_PORT = 6667;
my $IRC_CHANNEL = '#test14131211';
my $IRC_NICK = 'r0gueb0t';

# encoding algorithm parameters
my $PRNG_SEED = 1234;
my $TIME_MAX_DELAY = (350 * $USEC_PER_MSEC);

my $sock = new IO::Socket::INET->new(PeerPort=> $IRC_PORT, Proto=> 'tcp',
   PeerAddr=> $IRC_SERVER) or die ("Could not create connection: $!\n");

# open the input file (list of messages to send)
my $infile = $ARGV[0];
open IN, "<$infile"  or die("Could not open input file '$infile': $!\n");

# seed the MT PRNG
srand($PRNG_SEED);

# loop through all messages and calculate the scheduled departure times
my @messages = ();
my @sleep = ();
for($i = 0; $line = <IN>; $i++) {
    chomp $line;
    $messages[$i] = "PRIVMSG $IRC_CHANNEL :$line\r\n";
    $sleep[$i] = (int(rand($TIME_MAX_DELAY)) + 2*$USEC_PER_SEC);
```

```
    printf "%d,%.6f,%s\n", ($i+1), ($sleep[$i]/$USEC_PER_SEC), $line;
}
close IN;

# connect to IRC server and join channel
print $sock "USER $IRC_NICK $IRC_NICK 127.0.0.1 :Unknown\r\n";
print $sock "NICK $IRC_NICK\r\n";
sleep 2;
print $sock "JOIN $IRC_CHANNEL\r\n";
print "Joined channel successfully, starting in 8 seconds...\n";
sleep 8;

# define the SIGALRM handler
my $pc = 0; # init the global packet counter
$SIG{ALRM} = sub {
    ualarm $sleep[$pc+1]; # schedule the next SIGALRM
    my ($sec, $usec) = gettimeofday();
    my $len = length($messages[$pc]);
    print "$pc,".($len + 105).",$sec.$usec,$messages[$pc]";
    syswrite $sock, $messages[$pc], $len;
    $pc++;
};
ualarm 1; # schedule SIGALRM in 1 usec (the first delay value is discarded)

# IMPORTANT: busy-loop so we can catch alarm signals
while($pc <= $#messages) {
    sleep 1;
}

# clean up
print $sock "PART $IRC_CHANNEL\r\n";
sleep 2;

close $sock;
```

A.2.2 decoder.pl

```
#! /usr/bin/perl

use Time::HiRes qw( usleep gettimeofday );
use Math::Random::MT qw( srand rand );
use IO::Socket::INET;
use IO::Socket::SSL;
use Net::Pcap;

# timing constants
my $USEC_PER_MSEC = 1000;
my $MSEC_PER_SEC = 1000;
my $USEC_PER_SEC = ($USEC_PER_MSEC * $MSEC_PER_SEC);

# IRC parameters
my $IRC_SERVER = 'X.X.X.X';
my $IRC_PORT = 6667;
my $IRC_CHANNEL = '#test14131211';
my $IRC_NICK = 'd3c0d3r';

# en/decoding algorithm parameters
my $TIME_MAX_DELAY = (350 * $USEC_PER_MSEC);
my $ENC_PACKET_COUNT = 64;
my $EXTRA_CAP_TIME = 5; # seconds to add to the expected capture time
my $DECODE_TEMP_FILE = "decode-temp.txt";

# use passed-in WM if available
my $WM = "01010010110001010110011111000101";
if(defined($ARGV[1])) {
```

```
    $WM = $ARGV[1];
}

# use passed-in PRNG seed if available
my $PRNG_SEED = 1234;
if(defined($ARGV[2])) {
    $PRNG_SEED = int($ARGV[2]);
}

# default capture device for PCAP
my $PCAP_DEV = 'eth0';

my $err;
my $start_sec;
my $start_usec;

# master index of received packets
my @packets;

my @expected;
srand($PRNG_SEED); # seed the MT PRNG
rand($TIME_MAX_DELAY); # IMPORTANT: discard the first rand() result!
$expected[0] = 0; # t1 is always zero (relative)
for($i = 1; $i < $ENC_PACKET_COUNT; $i++) {
    $expected[$i] = $expected[$i-1] + (int(rand($TIME_MAX_DELAY)) + 2*$USEC_PER_SEC);
}
my $capture_time = int($expected[$#expected] / $USEC_PER_SEC) + $EXTRA_CAP_TIME;

# Use PCAP to capture all traffic on our IRC connection
# Mostly based on http://www.perlmonks.org/index.pl?node_id=170648
# IMPORTANT: variables below must be global
my $pcap;
my $sock;
my $pcap_dump;

if(!defined($ARGV[0])) {
    $sock = new IO::Socket::SSL->new(PeerPort=> $IRC_PORT, Proto=> 'tcp',
    PeerAddr=> $IRC_SERVER) or die ("Could not create connection: $!\n");

    my $localport = $sock->sockport;
    print "Established connection using local port $localport\n";

    # connect to IRC server and join channel
    print $sock "USER $IRC_NICK $IRC_NICK 127.0.0.1 :Unknown\r\n";
    print $sock "NICK $IRC_NICK\r\n";
    sleep 2;
    print $sock "JOIN $IRC_CHANNEL\r\n";
    print "Joined channel successfully...\n";
    sleep 1;

    #   Use network device passed in program arguments or if no
    #   argument is passed, determine an appropriate network
    #   device for packet sniffing using the
    #   Net::Pcap::lookupdev method
    my $dev = $PCAP_DEV;
    unless (defined $dev) {
        $dev = Net::Pcap::lookupdev(\$err);
        if (defined $err) {
            die 'Unable to determine network device for monitoring - ', $err;
        }
    }

    #   Look up network address information about network
    #   device using Net::Pcap::lookupnet - This also acts as a
    #   check on bogus network device arguments that may be
    #   passed to the program as an argument
    my ($address, $netmask);
```

```
    if (Net::Pcap::lookupnet($dev, \$address, \$netmask, \$err)) {
        die 'Unable to look up device information for ', $dev, ' - ', $err;
    }

    #   Create packet capture object on device
    $pcap = Net::Pcap::open_live($dev, 1500, 0, 0, \$err);

    # Compile and set packet filter for packet capture
    my $filter;
    Net::Pcap::compile(
        $pcap,
        \$filter,
        "(src $IRC_SERVER) && (dst port $localport)",
        0,
        $netmask
    ) && die 'Unable to compile packet capture filter';
    Net::Pcap::setfilter($pcap, $filter) && die 'Unable to set packet capture filter';

    my $timestamp = `date +"%Y%m%d-%H%M"`; chomp $timestamp;
    $pcap_dump = Net::Pcap::dump_open($pcap, "decoder-ssl-$timestamp.pcap");
    print "Starting PCAP capture for $capture_time seconds...\n";

} else {
    $pcap = Net::Pcap::open_offline($ARGV[0], \$err);
    print "Reading from PCAP file $ARGV[0]...\n";
}
unless (defined $pcap) {
    die 'Unable to create packet capture on device ', $dev, ' - ', $err;
}

#   Set callback function and initiate packet capture loop
Net::Pcap::loop($pcap, -1, \&receive, ''); # || die 'Unable to perform packet capture';

# clean up (if live capture was used)
if(!defined($ARGV[0])) {
    Net::Pcap::dump_flush($pcap_dump);
    Net::Pcap::dump_close($pcap_dump);

    print $sock "PART $IRC_CHANNEL\r\n";
    sleep 2;

    close $sock;
}

Net::Pcap::close($pcap);

# perform sliding-window offset alignment, collect possible decodings
my @possible = ();
my %results = {};
my $delta = 100 * $USEC_PER_MSEC;
my $offset = 0 * $USEC_PER_SEC;
my $max = 10 * $USEC_PER_SEC;
my $step = 10 * $USEC_PER_MSEC;

for($t1 = $offset; $t1 <= $offset + $max; $t1 += $step) {
    @possible = (); # IMPORTANT: reset possible counts
    #print "Processing decoding (t1 = $t1):\n"; # DEBUG
    for($pc = 0; $pc < $ENC_PACKET_COUNT; $pc++) {
        for $packet (@packets) {
            ($len, $time) = split /,/, $packet;
            if($time >= ($t1 + $expected[$pc] - $delta) && $time <= ($t1 + $expected[$pc]
+ $delta)) {
                $possible[$pc] += $len;
                #printf "Packet len = %d is in range [%d, %d]\n", $len, ($t1 +
$expected[$pc]), ($t1 + $expected[$pc] + $delta); # DEBUG
            }
        }
```

```
    }

    # write length values to temporary file
    open TEMP, ">$DECODE_TEMP_FILE";
    #print "Possible decoding (t1 = $t1):\n"; # DEBUG
    for($i = 0; $i < $ENC_PACKET_COUNT; $i++) {
        $len = $possible[$i];
        if($len eq "") { $len = "0"; }
        #printf "%d: len %d\n", $i, $len; # DEBUG
        print TEMP "$len\n";
    }
    close TEMP;

    # run decoding, store result
    $result = `./ldb-decoder $WM $DECODE_TEMP_FILE`;
    chomp $result;
    $results{$t1} = $result;

    #print "Matching bits: $result\n"; # DEBUG
    #print "-----------------\n"; # DEBUG

    #printf "OFFSET %.3f %d\n",($t1/$USEC_PER_SEC),$result; # DEBUG
}

# pick the best result
my $wkey = -1, my $wval = -1;
for $k (keys %results) {
    if($results{$k} > $wval) {
        $wval = $results{$k};
        $wkey = $k;
    }
}
print "\nWinner: t1 = $wkey, $wval bits match\n";

# Extract the encoding packets based on the best offset
my $t1 = $wkey;
my @enc_packets = ();
for($pc = 0; $pc < $ENC_PACKET_COUNT; $pc++) {
    for $packet (@packets) {
        ($len, $time) = split /,/, $packet;
        if($time >= ($t1 + $expected[$pc] - $delta) && $time <= ($t1 + $expected[$pc] +
$delta)) {
            $enc_packets[$pc] = $packet;
            #printf "Packet len = %d is in range [%d, %d]\n", $len, ($t1 +
$expected[$pc]), ($t1 + $expected[$pc] + $delta); # DEBUG
        }
    }
}

print "Round 1: Encoding Packets:\n";
my $difftotal = 0;
my $diffcount = 0;
for($i = 0; $i < $ENC_PACKET_COUNT; $i++) {
    my ($len, $time) = split /,/, $enc_packets[$i];
    my $diff = ($time - ($expected[$i] + $t1));
    # only factor into average if value is reasonable
    if($len > 0 && $time > 0) {
        $difftotal += $diff;
        $diffcount++;
    }
    printf "%d: len = %d, time = %d (exp %d, diff %.3f)\n", $i, $len, $time, $expected[$i]
+ $t1, $diff / 1000;
}

# Adjust t1 based on the average difference in arrival times
# if($diffcount > 0) {
#   my $diffavg = int($difftotal/$diffcount);
```

```
#   print "\nAverage diff: $diffavg\n";
#   $t1 += $diffavg;
#   print "Adjusted t1 = $t1\n\n";
# }

# Re-run the decoding with adjusted t1, smaller delta
#my $delta = 30 * $USEC_PER_MSEC;
@possible = (); # IMPORTANT: reset possible counts
#print "Processing decoding (t1 = $t1):\n"; # DEBUG
for($pc = 0; $pc < $ENC_PACKET_COUNT; $pc++) {
    for $packet (@packets) {
        ($len, $time) = split /,/, $packet;
        if($time >= ($t1 + $expected[$pc] - $delta) && $time <= ($t1 + $expected[$pc] +
$delta)) {
            $possible[$pc] += $len;
            #printf "Packet len = %d is in range [%d, %d]\n", $len, ($t1 +
$expected[$pc]), ($t1 + $expected[$pc] + $delta); # DEBUG
        }
    }
}

# write length values to temporary file
open TEMP, ">$DECODE_TEMP_FILE";
print "\nOptimal decoding (t1 = $t1):\n"; # DEBUG
for($i = 0; $i < $ENC_PACKET_COUNT; $i++) {
    #printf "%d: len %d\n", $i, $possible[$i]; # DEBUG
    print TEMP $possible[$i]."\n";
}
close TEMP;

# run decoding, store result
$result = `./ldb-decoder $WM $DECODE_TEMP_FILE -v`;
chomp $result;
$results{$t1} = $result;

print "Recovered WM: $result\n";
print "-----------------\n";

# Re-extract the encoding packets based on the adjusted t1
my @enc_packets = ();
for($pc = 0; $pc < $ENC_PACKET_COUNT; $pc++) {
    for $packet (@packets) {
        ($len, $time) = split /,/, $packet;
        if($time >= ($t1 + $expected[$pc] - $delta) && $time <= ($t1 + $expected[$pc] +
$delta)) {
            $enc_packets[$pc] = $packet;
            #printf "Packet len = %d is in range [%d, %d]\n", $len, ($t1 +
$expected[$pc]), ($t1 + $expected[$pc] + $delta); # DEBUG
        }
    }
}

print "Round 2: Encoding Packets (optimal t1):\n";
for($i = 0; $i < $ENC_PACKET_COUNT; $i++) {
    my ($len, $time) = split /,/, $enc_packets[$i];
    my $diff = ($time - ($expected[$i] + $t1));
    printf "%d: len = %d, time = %d (exp %d, diff %.3f)\n", $i, $len, $time, $expected[$i]
+ $t1, $diff / 1000;
}

sub receive {
    my ($user_data, $header, $packet) = @_;

    # save the packet to a PCAP file
    if(defined($pcap_dump)) { Net::Pcap::dump($pcap_dump, $header, $packet); }

    printf "len = %d, time = %d.%06d\n", $header->{'len'}, $header->{'tv_sec'}, $header-
```

```
>{'tv_usec'};

    # store the number of usec elapsed since the first packet
    if(defined($packets[0])) {
        my $diff = time_diff($start_sec, $start_usec, $header->{'tv_sec'}, $header-
>{'tv_usec'});
        #printf "len = %d, diff = %d\n", $header->{'len'}, $diff; # DEBUG
        #printf "%d,%s\n", $header->{'len'}, $header->{'tv_sec'} * $USEC_PER_SEC +
$header->{'tv_usec'}; # DEBUG
        push @packets, (sprintf "%d,%d", $header->{'len'}, $diff);
    } else {
        $start_sec = $header->{'tv_sec'};
        $start_usec = $header->{'tv_usec'};
        push @packets, (sprintf "%d,%d", $header->{'len'}, 0);
    }

    if(!defined($ARGV[0])) {
        my ($sec, $usec) = gettimeofday();
        $elapsed = time_diff($start_sec, $start_usec, $sec, $usec) / $USEC_PER_SEC;
        #print "elapsed = $elapsed\n"; # DEBUG
        if($elapsed > $capture_time) {
            print "Stopping capture after $elapsed seconds\n";
            Net::Pcap::breakloop($pcap);
        }
    }
}

# returns the difference in usec between two timevals (assumes b > a)
sub time_diff {
    my ($a_sec, $a_usec, $b_sec, $b_usec) = @_;
    my $diff = 0;
    $diff += ($b_sec - $a_sec) * $USEC_PER_SEC;
    $diff += ($b_usec - $a_usec);
    return $diff;
}
```

A.2.3 chaffbot.pl

```
#! /usr/bin/perl

use Time::HiRes qw( usleep );
use Math::Random::MT;
use IO::Socket::INET;

# timing constants
my $USEC_PER_MSEC = 1000;
my $MSEC_PER_SEC = 1000;
my $USEC_PER_SEC = ($USEC_PER_MSEC * $MSEC_PER_SEC);

# IRC parameters
my $IRC_SERVER = 'X.X.X.X';
my $IRC_PORT = 6667;
my $IRC_CHANNEL = '#test14131211';
my $IRC_NICK = 'chatter'.(int(rand(99)) + 1);

# encoding algorithm parameters
my $PRNG_SEED = 1234;
my $MIN_DELAY = $USEC_PER_SEC;
my $MAX_DELAY = int($ARGV[0]) * $USEC_PER_SEC;
#my $MIN_DELAY = 5;
#my $MAX_DELAY = 10;

my $sock = new IO::Socket::INET->new(PeerPort=> $IRC_PORT, Proto=> 'tcp',
   PeerAddr=> $IRC_SERVER) or die ("Could not create connection: $!\n");
```

```
# seed the MT PRNG
Math::Random::MT::srand(time);

# connect to IRC server and join channel
print $sock "USER $IRC_NICK $IRC_NICK 127.0.0.1 :Unknown\r\n";
print $sock "NICK $IRC_NICK\r\n";
sleep 2;
print $sock "JOIN $IRC_CHANNEL\r\n";
print "Joined channel $IRC_CHANNEL successfully as $IRC_NICK...\n";
sleep 1;

# define the SIGINT handler
$SIG{INT} = sub {
    # clean up
    print "Caught SIGINT, leaving channel and cleaning up...\n";
    print $sock "PART $IRC_CHANNEL\r\n";
    sleep 2;
    close $sock;
    exit(0);
};

$MTU = 60;
@chars = ('A'..'Z', 'a'..'z', '0'..'9');
srand(time);

while(1) {
    #$sleep = int(rand($MAX_DELAY - $MIN_DELAY + 1)) + $MIN_DELAY;
    $sleep = int(Math::Random::MT::rand($MAX_DELAY + 1)) + $MIN_DELAY;
    #$msg = "The magic number is $sleep";
    $max = int(Math::Random::MT::rand($MTU + 1)) + 1;
    $msg = join("", @chars[ map { rand @chars } ( 1..$max ) ]);
    $irc = "PRIVMSG $IRC_CHANNEL :$msg\r\n";
    #sleep $sleep;
    usleep $sleep;
    print "$sleep,$msg\n";
    syswrite $sock, $irc, length($irc);
}
```

A.3 Experiment Command Lists

This section lists the sequence of commands used to run the experiments. There are multiple shell sessions involved in each experiment, so each group of commands is prefixed by the name of the session (which is indicative of its role and physical location). The host names have been changed from their true DNS names.

A.3.1 Length-Only Experiment (Unencrypted Traffic)

```
Experiment Steps for Setup 1:

1. A (IRCC) ---> 1 (psyBNC) ---> S
    A - Germany, 1 - Berkeley

Encoding Parameters:
#define MTU 510
#define BUCKET_SIZE 10
```

```
#define PACKET_COUNT 64
#define WM_MAX_LENGTH 32
#define PRNG_SEED 1234
#define PAD_CHAR ' '


Before First Run:

Berkeley (planetlab.berkeley):
traceroute planetlab.germany
ping -c4 planetlab.germany
(paste into $BASE/planetlab/results/setup1-traceroute-ping.txt)
traceroute irc.server
ping -c4 irc.server
(paste into $BASE/planetlab/results/setup1-traceroute-ping.txt)

Germany (planetlab.germany):
traceroute planetlab.berkeley
ping -c4 planetlab.berkeley
(paste into $BASE/planetlab/results/setup1-traceroute-ping.txt)

Laptop Term1 (traffic monitoring):
traceroute planetlab.germany
ping -c4 planetlab.germany
(paste into $BASE/planetlab/results/setup1-traceroute-ping.txt)

Watermarks for Each Run:
1.  11000101011001111101010011101010O
2.  10100110111111100010101101001010
3.  01001010011110100101000101000001
4.  01101110101001001011011001111001
5.  00001001011001100000010000000100
6.  11111111110000000001001110101000
7.  01100001011100111000010101011101
8.  00111000100101011111110101011000
9.  00011001010001011011111010111111
10. 00100010111010111010110101100000


Each Run:

Laptop Term1 (traffic monitoring):
cd $BASE/planetlab/results/setup1_run$R_32/
sudo tcpdump -i lan -s0 -w roguebot-`date +"%Y%m%d-%H%M"`.pcap host irc.server

Germany (planetlab.germany):
sudo tcpdump -i eth0 -s0 -w ~/`hostname`-`date +"%Y%m%d-%H%M"`.pcap host
planetlab.berkeley & (note JOBID)

** Hit enter on both tcpdump's before proceeding **

Berkeley (planetlab.berkeley):
cd ~/psybnc
./psybnc (note PID)
sudo tcpdump -i eth0 -s0 -w ~/`hostname`-`date +"%Y%m%d-%H%M"`.pcap "host irc.server or
host planetlab.germany"

Laptop Term2 (watermarking psyBNC):
cd $BASE/psybnc-kdevelop
nano wm.txt (insert correct watermark for this run)
./psybnc (note PID)

AgobotVM:
cd /root/agobot3-priv-compiled-linux
./agobot3

Germany (planetlab.germany):
```

```
irssi
> /connect planetlab.berkeley 31337 password
> (Alt-3)
> .login user password
> .commands.list
> .bot.id
> .bot.status
> .mac.logout
Ctrl-Z
fg JOBID
Ctrl-C (cancel tcpdump)

Berkeley (planetlab.berkeley):
Ctrl-C (cancel tcpdump)
kill -9 PID (kill psyBNC relay)

Laptop Term2 (watermarking psyBNC):
kill -9 PID (kill watermarking psyBNC, disconnecting r0gueb0t)

AgobotVM:
Ctrl-C (kill Agobot)

Germany (planetlab.germany):
fg (switch back to irssi)
> /quit (exit irssi)

Laptop Term1 (traffic monitoring):
Ctrl-C (cancel tcpdump)

Laptop Term2 (watermarking psyBNC):
scp -pi ~/.ssh/planetlab_id_rsa gmu_nss@planetlab.berkeley:~/*.pcap
$BASE/planetlab/results/setup1_run$R_32/
scp -pi ~/.ssh/planetlab_id_rsa gmu_nss@planetlab.germany:~/*.pcap
$BASE/planetlab/results/setup1_run$R_32/
```

A.3.2 Length-Timing Hybrid Experiment (Encrypted Traffic)

```
Experiment Steps for Setup 1:

1. A (IRCC) ---> 1 (psyBNC) ---> S  <--- R
    A - Germany, 1 - Berkeley, R - Maryland
    A = Attacker, 1 = Stepping stone, R = Rogue Bot

Encoding Parameters:
#define MTU 510
#define BUCKET_SIZE 16
#define PACKET_COUNT 64
#define WM_MAX_LENGTH 32
#define PRNG_SEED 1234
#define PAD_CHAR ' '


Before First Run:

Berkeley (planetlab.berkeley):
traceroute planetlab.germany
ping -c4 planetlab.germany
(paste into $BASE/encrypted/results/setup1-traceroute-ping.txt)
traceroute irc.server
ping -c4 irc.server
(paste into $BASE/encrypted/results/setup1-traceroute-ping.txt)

Germany (planetlab.germany):
traceroute planetlab.berkeley
ping -c4 planetlab.berkeley
```

```
(paste into $BASE/encrypted/results/setup1-traceroute-ping.txt)

Maryland (local):
traceroute irc.server
ping -c4 irc.server
(paste into $BASE/encrypted/results/setup1-traceroute-ping.txt)

Generate messages.txt files:
Run 1:
1. 01010010110001010110011111000101
2. 11001010110101011100100010011111
3. 11101110110000101110000101101100
4. 01111010101001011111001101101000
5. 11110011000110110011101010101001
N. 11100110100010000110110010010101

Run 2:
1. 01010010110001010110011111000101
2. 01101001000110011010111011110101
3. 01011111010000011110110110010100
4. 00101000010011111010000011001001
5. 10111100010001010011111011010010
N. 01000101101110101011111111001010

Run 3:
1. 11000101101001010111010010011001
2. 10000011001011010100000011100101
3. 00111111111100010000011111010000
4. 10001111101101111100111010100010
5. 11101101101011011101100001001000
N. 10110101011111101101100001111011

for ($NUM = 1..5) {
    ./ldb-encoder $WM{$NUM} messages.txt > messages-ssl-run$R-$NUM.txt
}


Each Run ($NUM = 1..5, N):

Berkeley (planetlab.berkeley):
cd ~/psybnc
./psybnc (note PID)

Germany (planetlab.germany):
irssi
> /connect -ssl planetlab.berkeley 31337 password
> (Alt-3)
Ctrl-Z
sudo tcpdump -i eth0 -s0 -w ~/`hostname`-`date +"%Y%m%d-%H%M"`.pcap src
planetlab.berkeley &

Maryland (local):
sudo date (to cache credentials)
perl chaffbot.pl $NUM > chatterbot-`date +"%Y%m%d-%H%M"`.log & (skip if $NUM is N)
sudo tcpdump -i eth0 -s0 -w roguebot-`date +"%Y%m%d-%H%M"`.pcap 'dst irc.server and
greater 55' &

Local (Root console):
tcpdump -i lan -s0 -w botmaster-ssh-`date +"%Y%m%d-%H%M"`.pcap 'src planetlab.germany &&
src port 22'

Berkeley (planetlab.berkeley):
sudo tcpdump -i eth0 -s0 -w ~/`hostname`-`date +"%Y%m%d-%H%M"`.pcap src irc.server

** Hit enter on all tcpdump's before proceeding **

Maryland (local):
```

```
perl encoder.pl messages-ssl-run$R-$NUM.txt

Germany (planetlab.germany):
fg
(wait for run to finish in irssi)
> /quit
fg
Ctrl-C (cancel tcpdump)

Maryland (local):
fg
Ctrl-C (cancel tcpdump)
fg
Ctrl-C (cancel chatterbot)

Berkeley (planetlab.berkeley):
Ctrl-C (cancel tcpdump)
kill -9 PID (kill psyBNC relay)

Local (Root console):
Ctrl-C (cancel tcpdump)

Laptop Term2 (watermarking psyBNC):
scp -pi ~/.ssh/planetlab_id_rsa gmu_nss@planetlab.berkeley:~/*.pcap
$BASE/encrypted/results/setup1_chatter$NUM_32/
scp -pi ~/.ssh/planetlab_id_rsa gmu_nss@planetlab.germany:~/*.pcap
$BASE/encrypted/results/setup1_chatter$NUM_32/
(copy PCAP file from Maryland)

Decoding:
cd $BASE/encrypted/results/setup1_chatter$NUM_32/
ls -1 results/setup1_chatter$NUM_32/*.pcap | xargs -iFILE echo "time perl decoder.pl FILE
$WM{$NUM} > FILE-decode.log" > run
chmod u+x run
./run
rm -f run
```

Appendix B: Full Experiment Results

This appendix lists the full experiment results that are summarized in Sections 5.3.1.2 (unencrypted traffic) and 5.3.2.3 (encrypted traffic). The numbers in each table are the correct bits out of 32 possible bits.

B.1 Length-Only Experiment (Unencrypted Traffic)

For unencrypted traffic, we performed 10 runs of the same setup: the bot and WM Proxy were located in Maryland, the IRC Server in Arizona, the Stepping Stone in California, and the Botmaster in Germany. We sent a different random 32-bit watermark in each run and were able to decode all watermarks perfectly at our three monitoring stations.

Table 2: Full results for unencrypted traffic

Monitoring Location	Run 1	R2	R3	R4	R5	R6	R7	R8	R9	R10
WM Proxy - Maryland	32	32	32	32	32	32	32	32	32	32
StepStone - California	32	32	32	32	32	32	32	32	32	32
Botmaster - Germany	32	32	32	32	32	32	32	32	32	32

B.2 Length-Timing Hybrid Experiment (Encrypted Traffic)

For encrypted traffic, we performed three runs using six different chaff rates, yielding a total of 18 trials. The setup was similar: the watermark Source was in Maryland, the IRC Server in Germany, the Stepping Stone in California, and the Botmaster in Germany.

Table 3: Run 1 results for encrypted traffic

Monitoring Location	Chaff 1	Chaff 2	Chaff 3	Chaff 4	Chaff 5	Control
Chaff Rate (packets/sec)	0.6639	0.5093	0.4217	0.3313	0.2864	no chaff
Source - Maryland	31	30	29	30	30	32
StepStone - California	30	32	32	32	32	32
Botmaster - Germany	30	32	32	32	32	32

Table 4: Run 2 results for encrypted traffic

Monitoring Location	Chaff 1	Chaff 2	Chaff 3	Chaff 4	Chaff 5	Control
Chaff Rate (packets/sec)	0.6751	0.4855	0.4268	0.3209	0.2609	no chaff
Source - Maryland	30	30	32	31	30	32
StepStone - California	31	32	32	32	32	32
Botmaster - Germany	31	32	32	32	32	32

Table 5: Run 3 results for encrypted traffic

Monitoring Location	Chaff 1	Chaff 2	Chaff 3	Chaff 4	Chaff 5	Control
Chaff Rate (packets/sec)	0.6770	0.4985	0.4340	0.3188	0.3203	no chaff
Source - Maryland	28	31	28	30	31	32
StepStone - California	32	32	31	31	32	32
Botmaster - Germany	32	31	32	31	31	32

B.3 Screenshots

This section shows two representative screenshots from our experiments on unencrypted traffic. The first one (Figure 10) shows the botmaster's IRC window, with the whitespace padding characters not visible. The second one (Figure 11) shows the corresponding network trace in Wireshark, with the trailing whitespace padding being visible in several of the packets.

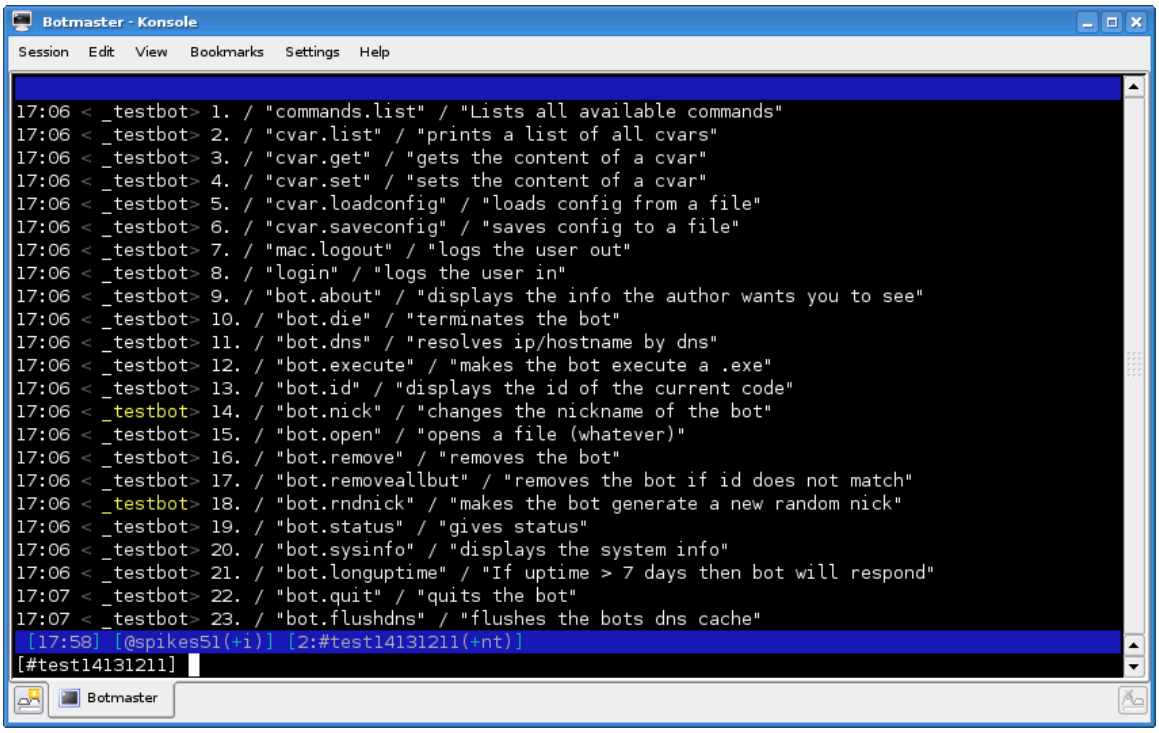

Figure 10: Botmaster's IRC window during an experiment

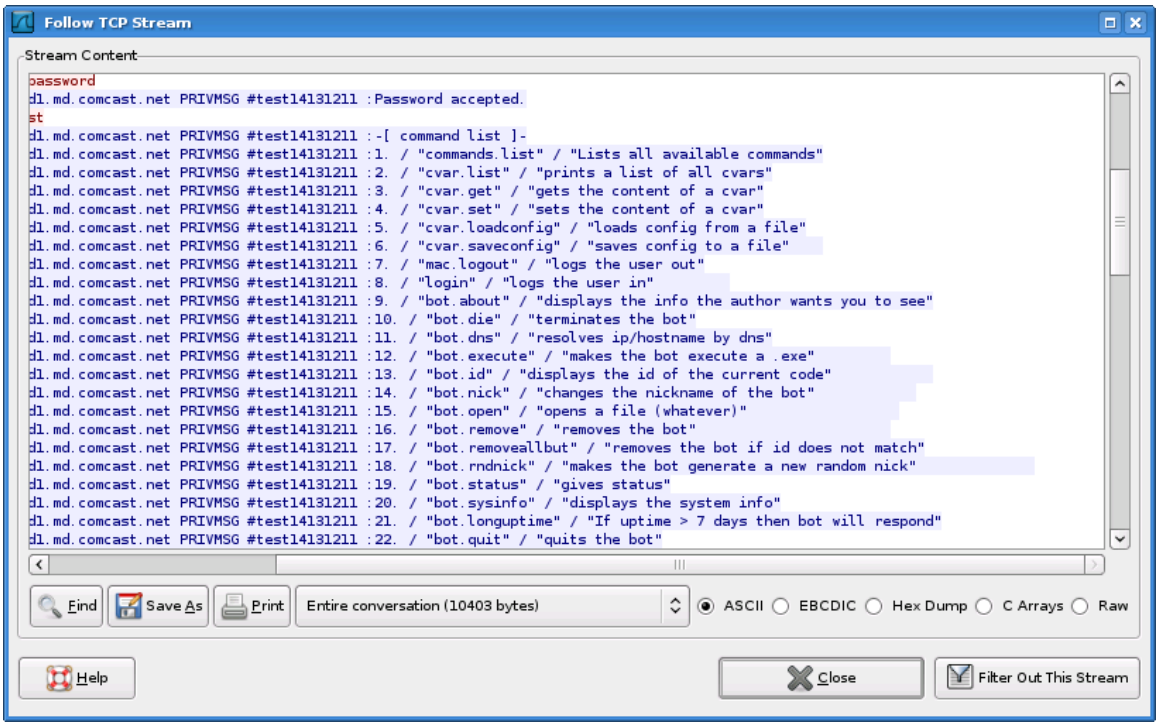

Figure 11: Network traffic capture showing trailing whitespace padding

Bibliography

Bibliography

[1] P. Bächer, T. Holz, M. Kötter, and G. Wicherski, "Know Your Enemy: Tracking Botnets," March 13, 2005, see http://www.honeynet.org/papers/bots/.

[2] P. Barford and V. Yegneswaran, "An Inside Look at Botnets," Special Workshop on Malware Detection, Advances in Information Security, Springer Verlag, 2006.

[3] P. Barford and M. Blodgett, "Toward Botnet Mesocosms," in Proc. First Workshop on Hot Topics in Understanding Botnets (HotBots), Cambridge, MA, April 10, 2007.

[4] J. Binkley and S. Singh, "An Algorithm for Anomaly-based Botnet Detection," in Proc. 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI), San Jose, CA, July 7, 2006, pp. 43-48.

[5] A. Blum, D. Song, and S. Venkataraman, "Detection of Interactive Stepping Stones: Algorithms and Confidence Bounds," in Proc. 7th Symposium on Recent Advances in Intrusion Detection (RAID 2004), Springer, October 2004.

[6] A. Brodsky and D. Brodsky, "A Distributed Content Independent Method for Spam Detection," in Proc. First Workshop on Hot Topics in Understanding Botnets (HotBots), Cambridge, MA, April 10, 2007.

[7] Z. Chi, Z. Zhao, "Detecting and Blocking Malicious traffic Caused by IRC Protocol Based Botnets," in Proc. Network and Parallel Computing (NPC 2007), Dalian, China, September 2007, pp. 485-489.

[8] K. Chiang and L. Lloyd, "A Case Study of the Rustock Rootkit and Spam Bot," in Proc. First Workshop on Hot Topics in Understanding Botnets (HotBots), Cambridge, MA, April 10, 2007.

[9] E. Cooke, F. Jahanian, and D. McPherson, "The Zombie Roundup: Understanding, Detecting, and Disturbing Botnets," in Proc. 1st Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI), Cambridge, MA, July 7, 2005, pp. 39-44.

[10] cURL and libcurl. http://curl.haxx.se/.

[11] D. Dagon, G. Gu, C. Zou, J. Grizzard, S. Dwivedi, W. Lee, and R. Lipton, "A Taxonomy of Botnets," unpublished paper, 2005.

[12] D. Dagon, C. Zou, and W. Lee, “Modeling Botnet Propagation Using Time Zones,” in Proc. 13th Network and Distributed System Security Symposium (NDSS), February 2006.

[13] DeleGate Multi-Purpose Application Gateway, see http://www.delegate.org/.

[14] D. L. Donoho, A. G. Flesia, U. Shankar, V. Paxson, J. Coit and S. Staniford, “Multiscale Stepping Stone Detection: Detecting Pairs of Jittered Interactive Streams by Exploiting Maximum Tolerable Delay,” in Proc. 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002): LNCS2516, Springer, October 2002, pp. 17-35.

[15] J. Evers, “'Bot herders' may have controlled 1.5 million PCs,” http://news.com.com/2102-73503-5906896.html?tag=st.util.print.

[16] F. Freiling, T. Holz, and G. Wicherski, “Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Denial-of-Service Attacks,” in Proc. 10th European Symposium on Research in Computer Security (ESORICS), Milan, Italy, September 12-14, 2005.

[17] J. Goebel and T. Holz, “Rishi: Identify Bot Contaminated Hosts by IRC Nickname Evaluation,” in Proc. First Workshop on Hot Topics in Understanding Botnets (HotBots), Cambridge, MA, April 10, 2007.

[18] M. T. Goodrich, “Efficient Packet Marking for Large-scale IP Traceback,” in Proc. 9th ACM Conference on Computer and Communications Security (CCS 2002), October 2002, pp. 117-126.

[19] J. Grizzard, V. Sharma, C. Nunnery, B. Kang, and D. Dagon, “Peer-to-Peer Botnets: Overview and Case Study,” in Proc. First Workshop on Hot Topics in Under-standing Botnets (HotBots), Cambridge, MA, April 2007.

[20] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, “BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation,” in Proc. 16th USENIX Security Symposium, Boston, MA, August 2007.

[21] G. Gu, J. Zhang, and W. Lee, “BotSniffer: Detecting Botnet Command and Control Channels in Network traffic,” in Proc. 15th Network and Distributed System Security Symposium (NDSS), San Diego, CA, February 2008.

[22] T. Holz, “A Short Visit to the Bot Zoo,” *IEEE Security and Privacy*, 3(3), 2005, pp. 76-79.

[23] N. Ianelli and A. Hackworth, “Botnets as a Vehicle for Online Crime,” in Proc. 18th Annual Forum of Incident Response and Security Teams (FIRST), Baltimore, MD, June 25-30, 2006.

[24] X. Jiang and X. Wang, "'Out-of-the-box' Monitoring of VM-based High-Interaction Honeypots," in Proc. 10th International Symposium on Recent Advances in Intrusion Detection (RAID), Queensland, Australia, September 2007.

[25] A. Karasaridis, B. Rexroad, and D. Hoeflin, "Wide-Scale Botnet Detection and Characterization," in Proc. First Workshop on Hot Topics in Understanding Botnets (HotBots), Cambridge, MA, April 10, 2007.

[26] J. Li, M. Sung, J. Xu, and L. Li, "Large Scale IP Traceback in High-Speed Internet: Practical Techniques and Theoretical Foundation," in Proc. 2004 IEEE Symposium on Security and Privacy, IEEE, 2004.

[27] R. Naraine, "Is the Botnet Battle Already Lost?" http://www.eweek.com/article2/0,1895,2029720,00.asp.

[28] M. Rajab, J. Zarfoss, F. Monrose, and A. Terzis, "A multifaceted approach to understanding the botnet phenomenon," in Proc. 6th ACM SIGCOMM on Internet Measurement, Rio de Janeiro, Brazil, October 25-27, 2006.

[29] A. Ramachandran, N. Feamster, and D. Dagon, "Revealing Botnet Membership Using DNSBL Counter-Intelligence," in Proc. 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI), San Jose, CA, July 7, 2006, pp. 49-54.

[30] P. F. Roberts, "California Man Charged with Botnet Offenses," http://www.eweek.com/article2/0,1759,1881621,00.asp.

[31] P. F. Roberts, "Botnet Operator Pleads Guilty," http://www.eweek.com/article2/0,1759,1914833,00.asp.

[32] P. F. Roberts, "DOJ Indicts Hacker for Hospital Botnet Attack," http://www.eweek.com/article2/0,1759,1925456,00.asp.

[33] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical Network Support for IP Traceback," in Proc. ACM SIGCOMM 2000, Sept. 2000, pp. 295-306.

[34] A. Snoeren, C. Patridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer, "Hash-based IP Traceback," in Proc. ACM SIGCOMM 2001, September 2001, pp. 3-14.

[35] J. Stewart, "Bobax Trojan Analysis," SecureWorks, May 17, 2004, http://secureworks.com/research/threats/bobax.

[36] E. Stinson and J. Mitchell, "Characterizing Bots' Remote Control Behavior," in Proc. 4th International Conference on Detection of Intrusions & Malware and Vulnerability Assessment (DIMVA), Lucerne, Switzerland, July 12-13, 2007.

[37] Stunnel - SSL Wrapper, see http://www.stunnel.org/.

[38] Symantec, “Symantec Internet Security Threat Report - Trends for January 06 - June 06,” Volume X, September 2006.

[39] Trend Micro, “Taxonomy of Botnet Threats,” Trend Micro Enterprise Security Library, November 2006.

[40] UnrealIRCD IRC Server, see http://www.unrealircd.com/.

[41] VMware White Paper, “Timekeeping in VMware Virtual Machines,” 2005, see http://www. vmware.com/pdf/vmware_timekeeping.pdf.

[42] P. Wang, S. Sparks, and C. Zou, “An Advanced Hybrid Peer-to-Peer Botnet,” in Proc. First Workshop on Hot Topics in Understanding Botnets (HotBots), Cambridge, MA, April 10, 2007.

[43] X. Wang, S. Chen, and S. Jajodia, “Network flow Watermarking Attack on Low-Latency Anonymous Communication Systems,” in Proc. 2007 IEEE Symposium on Security and Privacy, May 2007.

[44] X. Wang, S. Chen, and S. Jajodia, “Tracking Anonymous, Peer-to-Peer VoIP Calls on the Internet,” in Proc. 12th ACM Conference on Computer and Communications Security (CCS 2005), October 2005.

[45] X. Wang and D. Reeves, “Robust Correlation of Encrypted Attack traffic Through Stepping Stones by Manipulation of Interpacket Delays,” in Proc. 10th ACM Conference on Computer and Communications Security (CCS 2003), October 2003, pp. 20-29.

[46] X. Wang, D. Reeves, and S. Wu, “Inter-packet Delay Based Correlation for Tracing Encrypted Connections Through Stepping Stones,” in Proc. 7th European Symposium on Research in Computer Security (ESORICS 2002), LNCS-2502, Springer-Verlag, October 2002, pp. 244-263.

[47] K. Yoda and H. Etoh, “Finding a Connection Chain for Tracing Intruders,” in Proc. 6th European Symposium on Research in Computer Security (ESORICS 2000), LNCS-1895, Springer-Verlag, October 2000, pp. 191-205.

[48] Y. Zhang and V. Paxson, “Detecting Stepping Stones,” in Proc. 9th USENIX Security Symposium, USENIX, 2000, pp. 171-184.

[49] C. Zou and R. Cunningham, “Honeypot-Aware Advanced Botnet Construction and Maintenance,” in Proc. International Conference on Dependable Systems and Networks (DSN), Philadelphia, PA, June 25-28, 2006, pp. 199-208.

Curriculum Vitae

Daniel Ramsbrock was born in Bielefeld, Germany and is a dual citizen of Germany and the United States of America. He graduated from Princess Anne High School in Virginia Beach, Virginia, in 2002. He received his Bachelor of Science in Computer Science and his Bachelor of Arts in Criminology and Criminal Justice from the University of Maryland in 2006. In graduate school, he attended the University of Maryland for one semester, then transferred to George Mason University in January 2007. He received his Master of Science in Information Security and Assurance from George Mason University in 2008.