2017-03

Detecting target data in network traffic

Haycraft, Aaron

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DETECTING TARGET DATA IN NETWORK TRAFFIC

by

Aaron Haycraft

March 2017

Thesis Co-Advisors: Michael McCarrin
Robert Beverly

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2017	Master's Thesis 09-29-2014 to 03-21-2017

4. TITLE AND SUBTITLE

DETECTING TARGET DATA IN NETWORK TRAFFIC

5. FUNDING NUMBERS

6. AUTHOR(S)

Aaron Haycraft

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Data exfiltration over a network poses a threat to confidential information. Due to the possibility of malicious insiders, this threat is especially difficult to mitigate. Our goal is to contribute to the development of a method to detect exfiltration of many targeted files without incurring the full cost of reassembling flows. One strategy for accomplishing this would be to implement an approximate matching scheme that attempts to determine whether a file is being transmitted over the network by analyzing the quantity of payload data that matches fragments of the targeted file. Our work establishes the basic feasibility of such an approach by matching Transmission Control Protocol (TCP) payloads of traffic containing exfiltrated data against a database of MD5 hashes, each representing a fragment of our target data. We tested against a database of 415 million fragment hashes, where the length of the fragments was chosen to be smaller than the payload size expected for most common Maximum Transmission Units (MTUs), and we simulated exfiltration by sending a sample of our targeted data across the network along with other non-target files representing "noise." We demonstrate that under these conditions, we are able to detect the targeted content with a recall of 98.3% and precision of 99.1%.

14. SUBJECT TERMS	15. NUMBER OF PAGES
exfiltration, information, flows, hashes	73
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DETECTING TARGET DATA IN NETWORK TRAFFIC

Aaron Haycraft
Civilian, Federal Reserve Bank of San Francisco
B.S., California State University, Monterey Bay, 2014

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2017

Approved by: Michael McCarrin
Thesis Co-Advisor

Robert Beverly
Thesis Co-Advisor

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Data exfiltration over a network poses a threat to confidential information. Due to the possibility of malicious insiders, this threat is especially difficult to mitigate. Our goal is to contribute to the development of a method to detect exfiltration of many targeted files without incurring the full cost of reassembling flows. One strategy for accomplishing this would be to implement an approximate matching scheme that attempts to determine whether a file is being transmitted over the network by analyzing the quantity of payload data that matches fragments of the targeted file. Our work establishes the basic feasibility of such an approach by matching Transmission Control Protocol (TCP) payloads of traffic containing exfiltrated data against a database of MD5 hashes, each representing a fragment of our target data. We tested against a database of 415 million fragment hashes, where the length of the fragments was chosen to be smaller than the payload size expected for most common Maximum Transmission Units (MTUs), and we simulated exfiltration by sending a sample of our targeted data across the network along with other non-target files representing "noise." We demonstrate that under these conditions, we are able to detect the targeted content with a recall of 98.3% and precision of 99.1%.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

DoD	Department of Defense
IP	Internet Protocol
MASINT	measurement and signature intelligence
MTU	Maximum Transmission Unit
NIDS	Network Intrusion Detection System
NPS	Naval Postgraduate School
SRWBR	short range wide band radio
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USG	United States government
USN	U.S. Navy

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to first thank my advisors, Michael McCarrin and Dr. Beverly, for their hard work and dedication for assisting me in completing this work. This was a fun and rewarding project where I was able to use knowledge and skills that I learned throughout my time at NPS.

I would like to thank all of the professors I had during my tenure at NPS. I truly learned so much in the two years that I was here and you made learning memorable and enjoyable.

I would also like to thank my family and friends, especially my SFS cohort. Without your love and support, I would not have been able to complete this thesis. You all have been wonderful, kind, and caring. I hope that all future endeavors you take part in will be joyous!

I would finally like to thank the National Science Foundation and the Scholarship for Service program. Without this program, this thesis would not have been possible. I am eternally grateful to them for the opportunity I was given to complete my Master's degree.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Data exfiltration is the "unauthorized transfer of sensitive information from a target's network to a location which a threat actor controls" [1]. Recent data leaks, such as Edward Snowden's release of classified data [2], the Office of Personnel Management's loss of sensitive records in 2015 [3], or the Sony hack of 2014 [4], have brought increased attention to the potential damage that can be caused by malicious data exfiltration. Each of these incidents highlights the potential consequences of compromise and data loss to government, private organizations and private citizens. Arguably, the Office of Personnel Management incident was the most damaging, with the disclosure of personal information for over 22 million government officials, contractors, and their friends and families. In the Sony hack, the primary target was a private company rather than a government organization, and the primary consequence was financial damage caused by the loss of over 100 terabytes of data, including proprietary information. However, collateral damage included the loss of personnel details such as e-mails, salary information for executives, usernames and passwords. These incidents strongly argue for the importance of protecting the sensitive documents of the government and proprietary information or content of private companies.

Data exfiltration can take many forms depending on the quantity of data targeted by the attacker and the system on which the data is stored. Small artifacts, such as passwords or encryption keys, might be removed using low-bandwidth out-of-band methods. For example, cell phones send and receive Global System for Mobile communication (GSM) frequencies to communicate with cell towers and other cell phones. However, there is a possibility that transmissions using these GSM frequencies can be made to perform unauthorized tasks, including the transmission of personal or contact data that exists on the compromised cell phone. Further, Guri et al. were able to use these GSM frequencies to obtain information from a desktop computer by manipulating memory to produce GSM transmissions. A compromised Android cell phone placed within a close range to the desktop was able to capture and demodulate these signals [5].

A sneakernet attack is another data exfiltration method in which data is exfiltrated using a hard drive, flash drive, CD, or other media [6]. An insider might have an opportunity to

copy this data and information from the system to an external media device, such as a flash drive or hard drive.

In order to address problems caused by current vulnerabilities, some companies and agencies may train their employees to better protect their credentials and show them what different attacks look like to better protect the network. Others, might look at implementing more physical or electronic security, whether it be more cameras or guards at work, or more firewalls and data exfiltration detection systems. However, this still does not completely prevent data from leaving the network.

Current network data exfiltration systems rebuild packet streams to observe and read the payload. By building stronger network exfiltration detection systems, we hope to create a system to detect data loss.

1.1 Motivation

There are a large amount of packets that flow through networks constantly. Some systems analyze individual packets using full packet inspection, where each packet's payload will be analyzed to determine if there is anything malicious in the payload. There is also the technique of flow reassembly, which puts the payloads of all packets in a flow back together in the correct order to rebuild the original document. We hope to improve the ability of a network operator to stop data exfiltration attacks. Typical systems for detecting data exfiltration require full packet inspection, which usually involves reassembling packet streams. Packet reassembly can take time, which organizations often do not have enough of. Resources can be overloaded and might not be able to handle the stress of constantly reassembling packet data. Dharmapurikar and Paxson built a robust TCP stream reassembly machine. They were able to store up to 16 million connection records with 512 MB of SDRAM [7]. However, if an adversary could overwhelm the system, that could render the flow reassembly system useless.

We hope to contribute to the eventual development of a method that uses approximate matching to find target data without reconstructing streams of network traffic. Signature-based methods can sometimes struggle with finding files that have been slightly altered. This is because hashes or signatures are built with the idea that they will see the file in

the same state. However, if an adversary modifies the file, even by one byte, that will completely change the hash of the file and may cause a signature-based system to miss the file. Approximate matching has the ability to take pieces from different parts of the file and see if they might match hashes in a database. This method might miss the altered part of a file, but would not miss the other pieces.

Our goal in this thesis is to establish a foundation for developing such a technique by first demonstrating that we can correctly match payload fragments to file fragments. Theoretically, such a system could detect fragments more quickly and with fewer resources than would be required for stream reassembly, though we leave comparison of these performance metrics to future work. In addition, we focus exclusively on in-band data that is transmitted over the network without obfuscation; out-of-band data is not within the scope of this thesis and we do not address encryption or obfuscation of the payloads of the packets captured.

Our method looks at packets individually, and makes a determination when looking at each packet. Each packet's payload is hashed to determine if the hash matches something in the database. If the hash matches, that might give an indication that the file exists in our blacklist. If there are multiple hashes of packet payloads that match to hashes of the same file fragments, then it is highly likely that the file exists in the network traffic. Ultimately an approximate matching system will incorporate a strategy for translating these raw matches into a decision regarding whether the file is or is not present in the traffic; we leave the development of this strategy for future work.

1.2 Contributions

We made the following contributions:

- We built a system that uses open source tools to search for target content in traffic without flow reconstruction.
- We demonstrated our method using a database that contained 991 files and 415 million MD5 hashes in our black list, representing targeted content.
- We demonstrate that we are able to detect the targeted content with a precision of 98.3% and recall of 99.2%.

1.3 Structure of this Thesis

Chapter 2 is Technical Background, describing important concepts in order to better understand the thesis. Chapter 3 is Related Work, which looks at work that has been done in the past in this research area. Chapter 4 is Methodology, Chapter 5 is the cap of the thesis, and Chapter 6 is the Conclusion and Future Work.

CHAPTER 2:
Technical Background

This chapter talks about some key terms and concepts that are important to this thesis. First we describe definitions relevant to network exfiltration. We then describe concepts and terminology associated with target data leaving the network. Finally, we discuss the tools that were used to generate data for this thesis.

2.1 Networking Terms

To facilitate our discussion on data exfiltration, it is useful to begin with a review of some basic networking terminology.

- ***Flow***: An *IP flow* is defined as a "set of packets, observed in the network within some time period, that share a common key" [8]. The flow key consists of the source IP, source port, destination IP, destination port and protocol [8]. Flow level statistics give insight into the number and type of conversations occurring on a network, but not the content of the communications. For this reason the amount of data required to describe flows is fairly small and storage-efficient compared to storing full-packet captures.
- ***Flow Reassembly***: *Flow reassembly*, also called stream reassembly, is the process of completely restoring an entire TCP, or other data stream, to the correct order, often to inspect application-layer data. For example, a file might be reassembled at the end of a download [7].
- ***Maximum Transmission Unit***: The *maximum transmission unit*, or MTU, for a packet is the "maximum sized datagram that can be transmitted through the next network" [9]. If there is a difference in MTU between the source and destination, the MTU will be reduced to the smaller of the two and packets will then be fragmented or dropped [10]. The common MTU size for an Ethernet packet is 1,500 bytes [11].

2.2 Network Exfiltration Concepts

We briefly describe some basic concepts that are related to data exfiltration.

- ***Attacker***: Attackers are typically divided into two main categories: threatening insiders and unauthorized intruders. In their risk management approach to insider threat, Bishop et al. observe that threatening insiders can have access to machines on a corporate network; they can also have tokens, or other credentials that will enable them to access different areas of a building; finally, they can have intimate knowledge of certain classified or confidential projects [12]. Moreover, Bishop et al. state that there are three main characteristics that an insider might have: access, knowledge, and trust. Access is usually a factor because an employee has to be able to access their workspace. Knowledge might be a factor, especially if there is classified data involved. Trust is another key factor implied by privileged access, such as authorization to access a secure perimeter [12]. For example, system administrators have power with respect to the network that they administer. Attackers who lack the special privileges of insiders are typically considered external, unauthorized intruders. Unauthorized intruders may use a variety of attacks to gain access to protected resources, ranging from social engineering to exploitation of vulnerable services. Once access has been achieved to an exploited network, intruders often attempt to exfiltrate sensitive data [13].
- ***Intrusion Detection***: An *intrusion detection system* is defined by Dorothy Denning as a system that "aims to detect a wide range of security violations ranging from attempted break-ins by outsiders to system penetrations and abuses by insiders" [14]. Intrusion detection focuses on trying to identify potential threats to a network. Whether the intrusions are found by scanning for known signatures (see Section 2.3.2), or by identifying statistical anomalies, the most common procedure is to generate an alert that is sent to an administrator for review. Depending on what the system protocol is, an administrator could review the alert and take an action, or the system could initiate an automated response to the alert that is generated by the intrusion detection system.
- ***Exfiltration***: Trend Micro defines *data exfiltration* as the "unauthorized transfer of sensitive information from a target's network to a location which a threat actor controls" [1]. Giani et al. argue that techniques for detecting data exfiltration require the "ability to make a distinction between legitimate and malicious information communication" [15]. A *data exfiltration detection system* determines whether or not sensitive information has left an information system. Often this is accomplished by

consulting a black list of files that should not be allowed to leave the network. A *data exfiltration prevention system* operates in a similar manner but will also attempt to prevent unauthorized information from leaving the system.

- ***Firewall***: A *firewall* is a device or program designed to establish a perimeter around a network and create a single point of entry where security policies can be enforced and auditing can be performed [16]. Security policies for firewalls are enforced by the rulesets that are used. Each firewall contains a ruleset that determines what will happen when each packet is seen. These rulesets can be based on protocol, time to live, IP address, port number, and other different features.

  There are two different types of firewalls: stateful and stateless [17]. Stateless firewalls are meant to filter out policy violations that can be easily detected without reassembling packets, such as unauthorized connections to services like FTP and SSH connections, as well as RDP and MSSQL. Stateful firewalls are designed to complete more advanced tasks than stateless firewalls [18]. They are capable of observing an entire traffic stream. For example, if the stateful firewall sees a SYN-ACK packet, when it never saw a SYN packet, it can be configured to drop that packet. The stateless firewall would let the SYN-ACK packet pass because it does not maintain a record of previous packets.
- ***Deep Packet Inspection***: Firewalls capable of *deep packet inspection* are able to examine the application layer payloads of packets entering and leaving the network [19]. Such systems may use signatures, statistics or heuristics against payload content to attempt to enforce network policy. They may also be capable of flow reconstruction—i.e., the process of rebuilding the payloads across multiple packets in a flow.

2.3 Target Data Detection Concepts

The need to distinguish targeted content from surrounding data and locate it quickly is not confined to the analysis of network traffic. This task is frequently also used in digital forensics analysis performed on secondary storage devices. Some of the terms and techniques described in this section are adopted from that problem domain.

2.3.1 Target Data

We define *target data* as the data we attempt to detect in our traffic. In our experiment, this data comes from the Govdocs data corpus and is intended to represent intellectual property or sensitive documents that are not authorized for transmission over the network [20].

2.3.2 Signature

A *signature* is a string derived from some digital object, often for the purpose of creating a compressed representation of that object. *Signatures* are designed such that if two signatures are identical, there is a high probability that their objects are also identical. Ideally, no other pair of documents should have the same digital signature.

2.3.3 Signature-based Detection

Signature-based detection is used in intrusion detection systems, like Snort and Suricata [21], [22]. Packets and flows are collected from the network and inspected to determine if any data will match a signature in a ruleset. If the packet's content matches a signature in the ruleset, an alert will be generated. Rule-based detection focuses on having content of packet match a certain signature, whether based on IP, port number, or certain bytes in the payload's content. All of these items and more make up a signature that Snort and Suricata would look for.

In our work, we implement a kind of signature-based detection based on checking hashes of packets' payloads to see if they match against our database. In contrast to traditional rule-based methods, approximate matching focuses on finding similarities between the content of the packet payload and the target data. When knowing that multiple pieces of files exist in the payload, we could use the data gathered to match hashes of the file pieces to hashes in a database.

2.3.4 Cryptographic Hash

A *cryptographic hash* is a one-way function that takes arbitrary length input and generates a fixed-length output. Because the length of the output is constant, and the hashing function can be applied to an entire file, cryptographic hashes are a compact way of representing content. One application of a cryptographic hash is to determine a file's authenticity. For

example, hashes are provided online by many software vendors for their customers. It is possible to compare the hashes on the vendor's website with those hashes of the files that you download. Another helpful application of cryptographic hashes is digital signatures. As we will demonstrate in Section 4.5, hashes allow us to build compact signatures, which enables us to store many different hashes in a database.

The most common types of hashes are MD5, SHA1, and SHA256. MD5 hashes are 16 bytes long, SHA1 hash is 40 bytes long, and SHA256 is 64 bytes long. It will take more time for an SHA256 hash to be generated than an MD5 hash because of the size of the complexity of the algorithm. A comparison of the speed of these algorithms run on a MacBook Pro, with a 2.3 Ghz Intel Core i7 processor, and 16 GB of RAM, is shown in Table 2.1. The numbers shown are in kilobytes per second.

Block Size	16 Bytes	64 Bytes	256 Bytes	1024 Bytes	8192 Bytes
MD5	42.35	121.72	265.83	379.52	430.10
SHA1	44.81	131.57	285.54	412.20	456.88
SHA256	33.63	80.67	146.01	178.94	191.52

Table 2.1: Speed of three hash functions in megabytes per second when run repeatedly on different block sizes. Note that SHA256 is considerably slower than MD5.

The main hashing function that was used in this thesis was MD5. Even though MD5 is not secure, our work does not depend on the security features that an MD5 hash would provide. We are using it to provide us with a simple signature that we can store in a database.

2.3.5 File Target Detection with Hashing

A use of cryptographic hashes is to create signatures for content so it can be quickly identified. There are a few advantages of cryptographic signatures. They are lightweight and fast. They also allow the investigator to search for contraband without possessing illegal files themselves. Signatures created from entire files also have some limitations. One of the limitations of signatures created from an entire file is that if even one byte changes, the entire signature will change, and a slight variation might not be found. This is especially problematic for certain file formats, including Microsoft Word documents, which update time stamps when the date is saved. This change is transparent to the user.

2.3.6 Approximate Matching

Approximate matching is defined in the NIST Special Publication 800-168 as "a generic term describing any technique designed to identify similarities between two digital artifacts" [23]. An *artifact* is defined as a byte sequence of arbitrary length that exists in a file. Whereas any cryptographic hash that is used to compare the two artifacts will only detect a full match or no match at all, approximate matching produces a result that indicates the degree of similarity between two digital artifacts.

File fragments are pieces of files [24]. Hard drives, cell phones, network traffic files and other digital media, have a lot of data that has to be processed. By splitting this data into pieces, these pieces can be placed into a database or data store. These pieces of files can also provide insight as to where the data is in relation to the drive, network traffic, or other media used.

These pieces of files are then hashed and stored into a database. These hashes are of different fragment sizes are important because of the division of the files. By having these files in pieces, the hashes can be used to try and find pieces of files on systems or in internet traffic. Our searches can move faster with fragment hashes because we can quickly determine if a piece of a file's hash is present in a database or data store.

Rabin Fingerprinting

Michael Rabin in 1981 released his idea for fingerprinting of files using random polynomials. Rabin states that he will "use irreducible polynomials to 'fingerprint' files so that any unauthorized change will be detected with a very high probability" [25]. Rabin's fingerprinting algorithm offers an efficient technique for dividing a file into variable sized blocks. The block boundaries are calculated using a method of division by random polynomials, such that the block boundaries remain the same even if material is inserted or deleted in the file. This allows fingerprints to match even if modifications have caused the alignment to change.

Shingling

Shingling is the process by which we hashed overlapping blocks of data. With this experiment, we hashed a block of data, moved one byte over, then hashed another similar block of

data. Shingling is commonly used when trying to find similarities between two documents. This term was first mentioned in Broder's "Syntactic Clustering of the Web" in 1997 [26].

sdhash

sdhash is a "tool that allows two arbitrary blobs of data to be compared for similarity based on common strings of binary data" [27]. When building hashes, sdhash will use a sliding hash, which is similar to Broder's shingling method. However, a limited number of shingles are selected for the fingerprints. These shingles are selected based off of statistical analysis that will attempt to guess which shingles are most likely to be correlated with the data that is being fingerprinted.

Hash-based Carving

Hash-based carving is a technique that is used to detect the presence of target data on secondary media storage by evaluating the hashes of individual data blocks. It can locate known files by "recognizing a target file on a piece of searched media by hashing same-sized blocks of data from both the file and the media and looking for hash matches" [28]. This work provides inspiration for the databases that are used later in the thesis. These databases contain hashes of data blocks that look for pieces of data.

2.4 Tools

This section discusses a few of the tools that we used in this thesis.

2.4.1 bulk_extractor

Bulk_extractor is a forensic analysis tool designed for directly extracting artifacts of forensic value from storage media or disk images. The program is capable of looking at hard drive images, directories that contain many files, network traffic, cell phone images, and other types of digital media [29]. bulk_extractor ignores file system structure and runs directly on the underlying data blocks. This strategy allows it to process data on multiple cores at once, speeding up the evidence extraction process.

There are different scanners that can be used with bulk_extractor. Scanners include e-mail, PDF, RAR, and hashdb. These scanners will look for specific data when looking at the data that is given. We make use of the hashdb scanner in this thesis. See Section 2.4.2.

2.4.2 hashdb

Hashdb is a key-value store for storing block hashes. Its main capabilities include creating hash databases of MD5 block hashes, importing block hash values, scanning a hash database for matching hash values, and providing source information for hash values [30].

Hashdb is designed to support block hashing instead of full file hashing. This is useful because there are instances where you might only find fragments of a file instead of the entire thing. A major design goal that motivated the creation of hashdb was to develop a key value store that could store billions of hashes and respond to queries quickly enough to make hash-based carving practical. Hashdb can also be used to analyze network traffic and embedded content in other documents.

There are hashdb libraries for the Python and C++ programming languages. These can be used in programs to use hashdb instead of using the command line.

CHAPTER 3:
Related Work

Much of the prior work in preventing data exfiltration has focused on detecting attempts to transfer documents over the network. We briefly describe three broad categories of data exfiltration strategies: naive exfiltration, obfuscation exfiltration, and encrypted exfiltration. We then discuss three strategies for preventing document exfiltration: behavior-based approaches, content-based methods, and air-gapping. These concepts while different, look to achieve the same goal. Behavior-based approaches look at rule-based systems, anomaly detection, and a combination of rule-based systems and anomaly detection. Content-based methods dive deep into the data itself to try and find evidence of exfiltration. An air-gap is a form of isolated network that prevents connected devices from accessing the external networks, such as the internet.

3.1 Exfiltration Strategies

An adversary exfiltrating data can use a variety of techniques. Though an exhaustive survey of these techniques is outside the scope of this thesis, we discuss three basic categories of strategy. We call the most basic class of strategies "Naive exfiltration." This refers to exfiltration attempts that do not attempt to prevent detection. Obfuscated exfiltration attempts to send data that has an altered form in order to circumvent simple matching schemes. Finally, as the name suggests, encrypted exfiltration uses encryption to avoid detection and prevent third-party access to the data.

3.1.1 Naive Exfiltration

This thesis focused entirely on what we refer to as "naive exfiltration." Data was sent over HTTP, unobfuscated and unencrypted. Sending data over unencrypted FTP connection is another example of this kind of exfiltration. Any traffic sent this way would have been able to be captured and read by the network administrators. Because the only alteration to the data is caused by the process of being sent over the network itself, it is possible to predict differences between on-disk format and the format in the traffic based on knowledge of the network.

3.1.2 Obfuscated Exfiltration

Obfuscated exfiltrated data can be a little more difficult to read. By flipping a few bytes in a file, replacing similar characters with a text search (O to a 0 for example), the exfiltrator can completely change the hash. While the file would still read the same, the hash would be different, and might not be found.

Another example of obfuscated data is changing the file extension to defeat rudimentary checks for unauthorized file types. Finally, compressing a file is another simple way to obfuscate its content. When a file is compressed, that will change the hash of the file and most or all fragments of the file. The technique we test would detect data where the file extension has been changed, and may detect some data where a find and replace operation has been run. However, it is unlikely to detect compressed versions of the content unless the compression was ineffective.

3.1.3 Encrypted Exfiltration

Exfiltrating encrypted data is a good way for an adversary to exfiltrate data without having it be seen. Sending files over HTTPS, SCP, FTPS, or sending even encrypting a zip file, significantly increases the difficulty of detecting the file. The only way to decrypt these files would be to have the encryption key, or have the ability to break SSL encryption for data going over HTTPS, for example. Once encryption is broken, then the files could be run against the database to determine whether or not it should be allowed off the network.

3.2 Behavior-based Approaches for Preventing Document Exfiltration

Behavior-based approaches look at rule-based systems, anomaly detection, and a combination of rule-based systems and anomaly detection. Rule-based systems, such as firewalls and Snort, use rules to observe packet data and make a determination on what to do with that packet if it meets a certain criteria. Anomaly detection systems, such as Bro, attempt to gauge what normal and abnormal traffic is while observing traffic on a network. The combination of rule-based systems and anomaly detection systems can be effective if implemented properly. Giani et al. used both rules and heuristics to look for data exfiltration off of their network [15].

3.2.1 Rule-based Systems

Rule-based systems follow a certain set of rules that are created by a system administrator, or the manager of the system. These systems will look to see if the data passing through them matches a certain condition. If the condition is met, then the system will perform a previously specified action. Otherwise, the data flow will continue as normal.

Firewalls

The first firewall was created in 1983, at Stanford by Brian Reid [31]. The purpose of this firewall was to safely connect to ARPAnet. In 1988, Reid wrote a procedure manual for the firewall he created in 1983 [31]. In 1990, Bill Cheswick at AT&T Bell Labs began working on a more advanced type of internet gateway. Cheswick's work focused on FTP and telnet services [32]. Earlier firewalls were stateless, meaning that packets passing through the firewall had to meet certain requirements and pass through a ruleset. Stateful firewalls came a few years later. Stateful firewalls collect data about each packet. Eventually, the firewall will make a determination about the packet and the state associated with it. With stateful firewalls potentially being compromised with denial of service attacks, an improvement was made to try and use a third generation firewall. This new type of firewall could determine if there was a port or service that was being constantly attacked. This firewall could look at a packet and see if it was acting like normal traffic for the type of service that was running on that port. Currently, the most common type of firewall is a Next Generation Firewall (NGFW). This firewall does more deep packet inspection than the third generation firewall. It will still look at application layer data, but has many more components, such as intrusion detection systems and web application firewalls [31].

Snort

Snort is an intrusion detection and prevention system. It has the capability to both detect and prevent exfiltration. It has three main modes: sniffer mode, packet logger mode, and network intrusion detection system mode. Sniffer mode just sniffs the traffic to see what exists in the traffic itself. Packet logger mode will log the packets Snort sniffs. Network intrusion detection system mode, or NIDS, will implement a ruleset and attempt to block traffic depending on if something in the packet's data matches up with a ruleset. NIDS mode can also act like a firewall, where it can drop packets without them being logged depending

on how the rules and settings are configured on the system. Snort uses signature-based detection, to where it looks for the packet to have a certain IP, port, or content. Then, a decision will be made on what to do with the packet if the packet matches something in the ruleset.

Snort signatures are capable of looking for anything in network traffic. They can be customized with rules that can look for certain byte patterns. For example, PDF files should not leave the network. In Snort, the signature can be written to look for certain byte patterns that are in all PDF files. This can be useful when trying to identify certain files in network traffic.

Snort allows a ruleset to be implemented for each instance. Each Snort rule in this ruleset is customizable. In addition, Snort offers a set of default rules that looks for activity happening on non-standard ports, ports that should be closed, or ports that are known as not having that much traffic flow on them. The rulesets can be tailored to your network's setup.

3.2.2 Anomaly Detection

Anomaly detection focuses on observing what type of data, how much data, and when the data is transferred on the network. Anomaly detection systems will focus on the traffic itself, while rule-based systems flag alerts based off of found content. Anomaly detection will look for oddities and strange patterns in the traffic. A great example of a network anomaly detection system is Bro.

Overview of Bro

The goal of Bro is to try and analyze a large amount of traffic to find anomalies. Bro uses its own proprietary language to run its system. This proprietary language can also be used to customize what data is collected and where it is stored. This system will then collect logs that can be parsed later.

How Bro Works

Bro starts by capturing network traffic with libpcap. When capturing network traffic, a filter can be applied to only look for certain protocols. Paxson et al. only looked for traffic on Finger, FTP, or telnet. After the traffic is captured, the traffic is then passed onto the

event engine. This engine will ensure that all packet headers are properly assembled before continuing. Once the engine checks the packets, an instance will be created if something does not appear to be correct in the packet header. If the packet has a proper header, the engine will then associate the packet with the correct TCP or UDP flow. After the packet is done being processed by the engine, the engine will then look to see if any events were generated. The events will then be processed by the engine with a policy script that will determine what happens with each individual event [33].

Advantages of Bro

Bro is helpful because it tries to look for anomalies in traffic, which could be something like bursty traffic. When bursty traffic is seen, that could be an indicator that a large file or multiple files are being exfiltrated off of the network. This is considered an anomaly. Bro would also look at the amount of traffic to see if there is less than normal data flowing through the network. If there is not enough traffic flowing, that could indicate that something is wrong with the network. That is also an anomaly. These cases will raise a red flag in Bro and will alert an administrator as to what is going on. It is then possible to use the Bro logs to try and find what is going on and why Bro was logging those items. By determining the IP that sent the data, the network administrator can then find who the IP belongs to, as long as the network is documented thoroughly. The administrator can then determine what went wrong and how serious the danger was.

Bro is also helpful in the amount of data that it logs. Bro will look at to DNS traffic, HTTP requests, and if any other connections attempted to be made over FTP, SSH and other major services. These logs will include the times and IPs that made the connections. These can be useful when trying to identify a potential intrusion or compromise.

3.2.3 Combined Approaches

A "combined approach" means what its name implies, a combination of multiple approaches to ensure data is not exfiltrated off of networks. This approach looks at using both rule-based systems along with anomaly detection systems. By using a combined approach, that provides more opportunities for a system to find potential problems that might arise on a network.

Giani's Work

Giani et al. analyzed the different ways that data could be exfiltrated, and whether or not the speeds that were observed would be sufficient. They looked at methods like CD/DVD burning, using a T1 internet line, a T3 internet line, cable or DSL line, and printing out the data [15]. The T3 internet line was the quickest at exfiltrating data. However, when they were performing the data exfiltration, they had to ensure that they were covert in their mission. Middle ground must be achieved where they could exfiltrate the most data, but still not be caught or found.

This is considered a combined approach because the authors looked at many different factors when considering how the data should be exfiltrated and whether or not this data would be recognized. They looked at the statistics to determine what method would be the fastest, then looked at what the best method to avoid signature-based detection systems and anomaly detection systems.

3.3 Content-based Methods for Preventing Document Exfiltration

Content-based methods dive deep into the data itself to try and find evidence of exfiltration. These methods use content-based signatures, packet reassembly, and approximate matching. Content-based signatures try to identify certain byte patterns and will trigger signatures when those byte patterns are found. Packet reassembly will be used by some firewalls to reassemble the data before it is delivered to the destination. This is useful when trying to find potential exfiltrated files that should be found as a 100% match.

3.3.1 Content Inspection with Exact Matching

When inspecting content for exact matching, an exact match of a piece of data has been found that was being searched for. That is helpful when needing to identify whether or not a specific piece of data was found. This concept however does have a drawback. If the file that is being searched for has been altered, even by one byte, that would cause the data to not be found by the system.

Content-based Signatures

An example where content-based signatures can be used is with Snort. As mentioned earlier in Section 3.2.1, Snort allows a ruleset to be implemented for each instance. An example ruleset might contain special content looking for certain things, such as the starting bytes for a RAR file. Every RAR file they look for has the exact same header. Once the header is found, that packet will not leave their network. Because their example setup had RAR files storing important things, that made it easier to detect whether or not that file was going to leave the network. They decided to not have any RAR files pass through the network. Whenever a piece of a RAR file was seen, Snort dropped the packet.

This prevents insider data exfiltration with RAR files only. Due to the commonality of four bytes in a RAR file header, this will allow Snort to look for something specific each time. Only one signature has to be built and constructed. The RAR files will never be permitted to leave the network because of the signature that is in place.

Packet Reassembly

Glavlit, developed in 2006, provides reassembly and inspection of data at the application layer, but requires all authorized transmissions to be vetted manually in advance [34]. The guard overseeing the traffic that will be run through the Glavlit program will divide each file seen in network traffic into 1024 byte chunks and generate a hash for each of them using SHA-1. Once the hash is created for the fragment of the file, the guard will then verify that the file is known good and allow the file to pass. If the guard determines that the file is known as not good, then the file will not be permitted to leave the network. For their experiment, the guard was receiving 3000 requests per second. The guard was able to process the packets as quickly as the server was getting the packets to the guard [34]. The performance shows that the throughput was slightly over 11Mbps when the file size was above 10KB.

In this instance, the guard was a machine that was able to automatically analyze the packets in real-time. Initially, there were some concerns about the speed of the traffic that was passing through the guard. However, at the end of the experiment, the speed of traffic that flowed through the guard equaled that of the traffic that was not being sent through the guard.

3.3.2 Detecting Exfiltration with Approximate Matching

When using approximate matching to find potentially exfiltrated data, a hash will match in a database or data store. Hashes are small and easy to store. They are easy to compute and in the case of this thesis, not used for security purposes. Hashes are reliable, which helps when comparing them to find potential files that might have left the network.

Finding Targeted Data in Secondary Devices

Roussev's work focused on using the National Software Reference Library, or NSRL, provided by NIST to attempt to compare the files found with those they attempted to exfiltrate with their experiment. Foster used sector hashes of secondary media drives to correlate the hashes of similar sectors. Garfinkel and McCarrin worked on hash-based carving, which detects the presence of target data on secondary media by looking at the individual data blocks.

In 2009, Roussev discussed a tool that would be useful in trying to identify files, hashing. A cryptographic hash function is applied to the files on a file system, or to the file system itself [35]. The output obtained from the cryptographic hash function are hashes. These hashes can then be compared against the National Software Reference Library, or NSRL. This library contains many known operating system and file system specific files. Many of the hashes obtained might match up to those in the NSRL. Those hashes were filtered out to get the hashes that do not match anything in the NSRL database. Those hashes could potentially match other files, such as Word documents and PDFs from different computers. By hashing the entire file, there is an opportunity to find if a file is on multiple systems.

In 2012, Foster attempted to find documents from a data corpus on a system [36]. Her work focused on finding documents by looking at sector hashes. When generating hashes for each of the sectors used in the experiment, databases were generated to compare the hashes with other data in different corpora to see if there were any similarities between the data. The govdocs corpus, as well as a few other data sets, were used. If the sector hashes did not match others in the data set, that indicated a unique sector of data might have been found. Otherwise, the sectors that appeared multiple times were ignored.

In order to test her experiment, Foster had to create multiple databases using different programs. When these databases were created, five different databases were built, one for

each program. This was to best determine what database would be used. It was determined that the query rate for everything below 100 million rows when calculating transactions per second worked best with SQLite. SQLite also worked best when looking at query speeds as well. These databases showed which type of program was best in order to build the various databases that need to be queried to find the data.

Foster's work is important because of its ability to look at databases that involve many rows of hashes. These hashes were used to find documents that existed in a real data corpus. These rows of hashes could be queried against a known data set of files to determine what matches and what does not. By determining the uniqueness of the data set, it could be determined that the data found was either interesting or not interesting. The more unique the data, the more interesting it becomes. This work is important because of the amount of hashes that were stored in the databases. There were many hashes stored, and that is something that we needed for this thesis.

As mentioned in Section 2.3.6, "Hash-based carving" was developed to detect the presence of target data on secondary storage media by evaluating the hashes of individual data blocks [28]. Hash-based carving finds target data by dividing each targeted file into sector-sized fragments and storing these in a database (Garfinkel and McCarrin use the hashdb database, which was built for this purpose [30]). This database stores hashes of fragments of files that can be compared to sectors on hard disks to determine if a file is or was present, even if the fragments are not stored contiguously. Our scheme will apply this approach to network traffic.

There are two other software applications that are involved with hashes: ssdeep and sdhash. ssdeep focuses on fuzzy hashing, which looks at trying to find similar sequences of bytes, but not necessarily in a similar order. sdhash is a tool that focuses on comparing two blobs of data to look for strings in common between the two blobs. Both of these software applications while important, do not pertain to this thesis.

Application to Detecting Document Exfiltration

Two systems have been created within the past few years that claim to help with detecting data exfiltration: the PROOFS system and a max-hashing system. The PROOFS system looks at signatures for digital objects, and stores digital objects from many machines on a

network on this one system. The max-hashing system proposed by Larbanet et al. observed traffic over a 40/100GbE Ethernet card to try and detect MP4 files specifically going over a network.

PROOFS, or Proactive Object Fingerprinting and Storage, is a system developed by Shields et al. to create and store the object signatures for digital objects [37]. This system is connected to many machines on a network. Anything that is done on any of the systems will be logged with an object signature on the PROOFS system. This is to monitor and track what each machine on the network does. This signature contains the metadata associated with each action. The signature also contains multiple digital fingerprints that can be used to search and track the data at a later point in time.

In order to effectively keep track of the data on the network, they created a dictionary containing fingerprints of the objects [37]. In order to create the fingerprints, tokens from documents must be pulled. Shields et al. used Oracle's Outside In program to remove the text from Word documents and PDF files. The text is then filtered with common stop words, symbols and numbers being removed. The processing is done by creating tokens based off of the types of documents that are being fingerprinted. The compressed fingerprints take up 375 bytes worth of data [37]. Once the fingerprints are created, the signatures will be created as well. The signatures are a combination of the fingerprints that are stored and the metadata that is sent from the system. File writes, file creations and other things can determine what the metadata will contain. To best run their experiments, they used the Enron data set. This dataset contained information and documents that they could use in order to create dictionaries to determine if this data could be found. After the trial run, it showed a success rate of about 96%, meaning that they were able to successfully identify 96% of the files that they knew were inside the dictionary.

There are some limitations with the system. The system is probabilistic, meaning that the system will take an educated guess whether or not the file is there. Shields et al. explains that they need other forensics tools to ensure that all forensic evidence is used. The authors also mention that this tool is not meant to replace any tool, but to be used in addition to other tools [37].

Within the past year, Larbanet et al. recently created a data exfiltration detection program that will look at traffic that is flowing over a 40/100GbE Ethernet card [38]. Their goal was

to find MP4 files in their network traffic at high speed. In order to accomplish their task, they computed the fingerprints of all their files they wanted to check against inside of the network traffic. They then decided to check for TCP and UDP flows. By checking for IP addresses and the ports in the traffic, they were able to successfully identify and match up certain flows with each other that they could use later. This would give them a better idea of how many flows went back and forth between two IP addresses. If they were able to determine IP addresses and ports belong to the same flow, the CPU was then alerted to start searching through the fingerprints of the data to try and find matches.

The max-hashing algorithm system proposed by Larbanet et al. in 2015 approaches the problem by computing the hash value of each fixed-size, small window in a data set, then keeping only the one with the maximum value [38]. Larbanet et al. when completing their work only used FTP connections to download their files. They were able to use those FTP connections to their advantage and collect traffic on their own Linux servers to see if they were truly able to find their files going across their network connections. For memory transferred from the Direct Memory Access to the GPU memory, the rate was about 50Gbps. The speed depended on the transmitted block size [38]. The fingerprint generation was tested at random. There were 768 megabytes worth of files randomly chosen and there were "four fingerprints per 1536 B blocks" [38]. The kernel was able to process at a rate of 119.9Gbps and generate 40 million fingerprints per second.

This thesis is different than their project in one key aspect: this thesis focuses on trying to identify fragments of files within HTTP traffic. We are taking the hashes of the payloads of HTTP traffic. We will attempt to look for these hashes in the hashdb databases that we create. This is similar to the fingerprinting technique that Larbanet et al. uses in their experiment.

3.4 Air-Gapping

As mentioned earlier, an air-gap is a form of isolated network that prevents a computer from accessing any outside networks [39]. By having computers strictly isolated from networks and other computers, this strategy forces exfiltrators to rely on out-of-band channels such as external media. A USB flash drive, CD, or floppy disk are examples of what could be used to get new data to an air-gapped machine. This makes exfiltrating data off of

these machines more difficult, by forcing malicious actors to have physical access to the air-gapped computer.

CHAPTER 4:
Methodology

This chapter covers our methods and techniques. We conducted several experiments to detect data exfiltration by comparing the hashes of packets' payloads to a database of blacklisted file fragments. The Govdocs data set was used to simulate the blacklisted content. The files from Govdocs were hosted on a server, and downloaded over HTTP. In addition to the blacklisted files, there was also noise data. This data does not belong to the blacklist. However, this data is important, as it is supposed to represent actual web traffic that could produce false positives when trying to find blacklisted files.

4.1 Hardware

We built our database using a 64-core machine with 512GB of RAM. In addition, there was over 14TB of available space on a network share to use to build the databases. The web server was capable of holding all Govdocs files and allowed for the use of HTTP over port 80. The Mac laptop has 16GB of RAM and a 512GB SSD. Figure 4.4 shows a diagram of the hardware.

4.2 Data Sets

To simulate data leaving the network, we used two subsets of the Govdocs data set, which we label Sample 1 and Sample 2. Both subsets were further divided into target data and noise data. Target data means this is a part of our blacklisted files: files that we hope to find when scanning against our target traffic. Noise data represents authorized data that we expect to find within our network traffic, however, it is traffic that we are not looking for. This is extraneous data that is found that allows us to simulate an actual network.

4.2.1 Govdocs

We used Govdocs as the data set for this thesis [20]. It was established as a file set to use for forensics research that everyone can use. Govdocs contains approximately one million files that are freely distributable. The one million files are divided into 1,000 separate

directories, each containing around 1,000 files. These files are all open source and can be used by anyone. To build the corpus, Garfinkel et al. performed basic searches of the .gov domain with Google and Yahoo in order to download the one million files [20]. These files include Word documents, Excel spreadsheets, PDF files, text documents, HTML webpages, and many other data types.

4.2.2 Govdocs Sample 1

Sample 1 contained the first 100 files from Govdocs directory 002, which contains 990 files. These files were chosen because they were the first 100 files in the directory. Out of the 100 files downloaded, the first 50 of those files were selected as target data and stored in our database, the second 50 files were noise data. Noise data consists of files that we downloaded that we know are not a part of the target data set that we are looking for. Sample 1 contains 6.4 MB of target data, and 13.6 MB of noise data. The amount of target data is about half that of the noise data for the first sample. There were 50 files chosen as it was a small sample size that we could test and obtain fast results.

Table 4.1, shows statistics about the 100 different files in Sample 1. The columns indicate how many of each file type were downloaded for the target data and the noise data. Because only 100 files were used in Sample 1 and our selection method did not ensure a uniform distribution across file types, there was not a wide variety of file type. Most of the target data was HTML files (45 out of the 50). With respect to the noise data, 38 out of 50 files were text documents.

The one UNK file that was found was actually a text document that was a court ruling, but which did not have a .txt extension. This document could have probably been interpreted as a text document, as it could be opened with a text editor. This file was interpreted as UNK when it became a part of Govdocs.

Extension	Target	Noise	Extension	Target	Noise
pdf	2	2	txt	3	37
html	45	1	UNK	0	1
jpg	0	8	text	0	1

Table 4.1: The distribution of file extensions in the Govdocs Sample 1 corpus. The Govdocs Sample 1 corpus contains mostly human readable documents and images. Adobe PDF, HTML, text, and JPG files make up the dataset.

4.2.3 Govdocs Sample 2

Sample 2 contained one 990-file directory, directory 002, and one 991-file directory, directory 013. We used the files from these directories to create our databases. The set of 990 files was our target data, the other 991 files were noise data. Sample 2 contains 387 MB of target data, and 536 MB of noise data. There was a 149 MB increase in noise data compared to the target data.

Table 4.2 columns show the differences in the 1,981 files in Govdocs Sample 2. The most common types of files that were downloaded for the target and noise data were PDF files, TXT documents and HTML webpages. Through our block level analysis, we find that most files have correct extensions that match the file type, but some files do not. Many of the text files found contained HTML data, meaning that they could also be interpreted as webpages.

Extension	Target	Noise	Extension	Target	Noise
pdf	242	271	log	3	3
txt	160	128	wp	1	0
html	292	241	UNK	5	2
tex	2	1	jpg	97	81
gls	1	0	xml	10	12
text	3	3	rtf	0	1
f	0	1	doc	64	71
xls	21	26	ppt	54	53
gif	12	57	ps	10	13
dbase3	2	1	csv	6	14
gz	4	9	java	1	3

Table 4.2: The distribution of file extensions in the Govdocs Sample 2 corpus. The Govdocs Sample 2 corpus contains mostly human readable documents and images. Adobe PDF, Microsoft Office, HTML, text files and graphical image files make up the majority of the corpus.

4.3 Byte Alignment

When files are transferred over a network, they must be divided into fragments so that no packet can exceed the network MTU. This can present problems for observers attempting to monitor the network for unauthorized exfiltration because it is difficult to match small pieces of content with whole blacklisted files. In addition, the first packet in every stream contains an HTTP header. The size of the header can vary. The location of the first byte of a file being downloaded depends on header size. The inclusion of the HTTP header

makes finding the start of data more difficult. Because the HTTP header exists at the start of a traffic stream to download a file, the HTTP header caused problems trying to find the fragments of files.

```
HTTP/1.1 200 OK
Date: Tue, 24 May 2016 19:52:32 GMT
Server: Apache/2.4.7 (Ubuntu)
Last-Modified: Fri, 06 Feb 2009 01:52:52 GMT
ETag: "7fe0-46236475ee500"
Accept-Ranges: bytes
Content-Length: 32736
Vary: Accept-Encoding
Connection: close
Content-Type: text/html
```

This is an example of an HTTP header. We chose HTTP because it represents one reasonable channel for data exfiltration, even on restricted networks.

We solve these problems by dividing our blacklisted files into fragments in advance, and matching these fragments to the fragments in the network traffic. To match the fragments correctly, we had to align the bytes of data with the bytes in the network packets. This would allow us to match the most hashes to a packet's payload. Since the varying HTTP header size makes it difficult to predict where a fragment of payload will begin or end in the file being transmitted without actually parsing the header itself, we populated our database with all possible alignments in each blacklisted file. This is why a small step size of 1 byte was used. To populate our blacklist database, we began at the start of each blacklist file, hashed a fragment, then moved over one byte, hashed that piece of data, etc. This guaranteed that all fragments would be present in our database. The hashdb utility provides options to allow the user to control how fragments are ingested, as necessary. See Figure 4.1 for a diagram that describes our ingestion process.

For this thesis, a byte alignment of 1 was necessary to find the target files in the traffic. Because of the nature of our data, with TCP and HTTP headers included of arbitrary length,

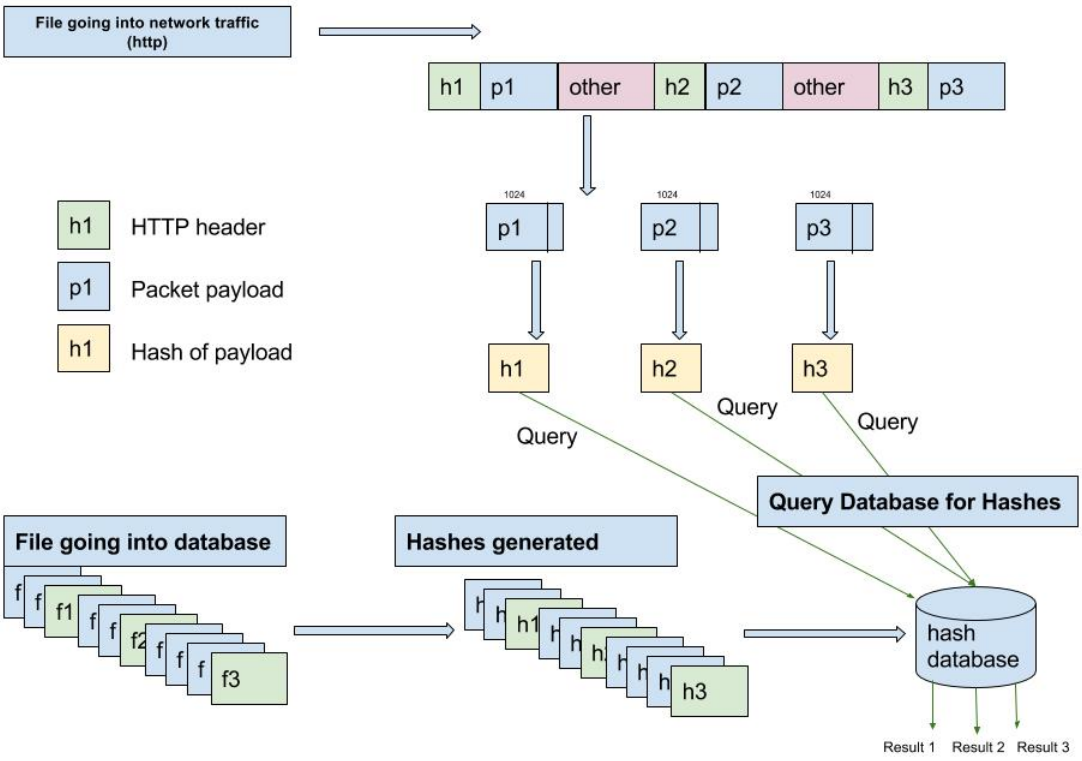

Figure 4.1: Diagram of how the files were converted into hashes.
The file going into network traffic was split up into pieces, due to the MTU of the network. We took the first 1024 bytes of payload from these Internet packets and took an MD5 hash of them. We then looked at h1, h2 and h3 shaded in yellow to see if they were in our database. We split the file into pieces, and created MD5 hashes of the pieces of files. The hashes were 1024 bytes each. These generated hashes were then placed into a hash database, where the hashes of packet payload could be compared.

having a byte alignment of 1 was the best option to make sure that we found the most hashes inside the database. If the byte alignment was increased up to 2, a problem might occur in that the bytes would be shifted just enough to cause problems and we would not find as many hashes that match. Even if the fragments were misaligned by one byte, the fragments would appear as not found in the network traffic, even though we knew that the files existed in the traffic. See Figure 4.2 to see how data blocks are hashed.

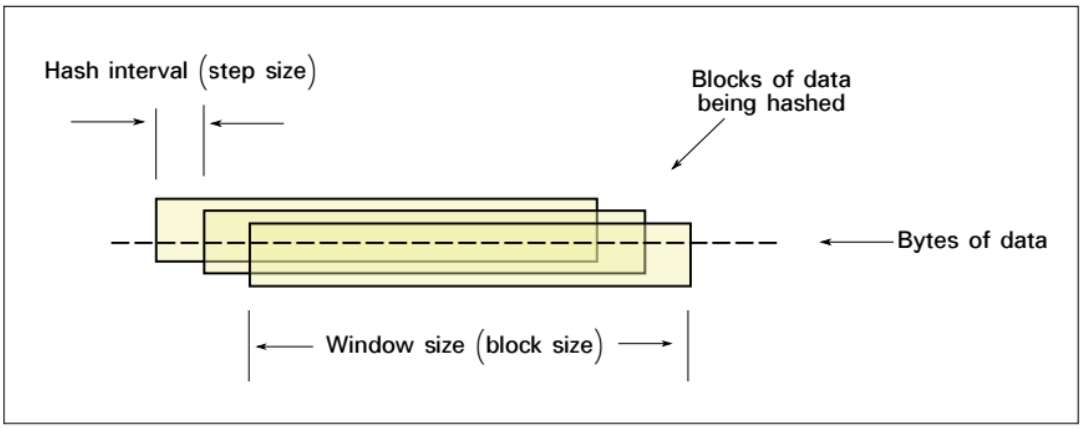

Figure 4.2: Figure 4 from hashdb user manual, showing how data blocks are hashed. Source [30].

4.4 Choosing a Block Size

We found when trying to download files that most packets had a TCP payload of 1448 bytes. This was because the overall size of the packet was 1514 bytes. In addition to the payload, we also took the Ethernet header, IP header, and TCP header into account. In our experiment, the Ethernet, IP, and TCP headers typically added up to 66 bytes. There were instances where the header size was 74 bytes. This mostly happened in ACK packets. We chose our block size for our blacklist based on the typical payload size. Initially, the block size was 1448 bytes, the size of most packets' payloads that we were looking for.

We attempted multiple different experiments, where we attempted to hash along 1024, 1200, and 1448 byte boundaries. When hashdb took a 1448 byte hash at every byte inside the directory, it created 6, 678, 056 hashes in the database to compare it to. When hashdb took a 1200 byte hash at every byte inside the directory, it created 6, 695, 346 hashes in the database to compare it to. When hashdb took a 1024 byte hash at every byte inside the directory, it created 6, 712, 048 hashes in the database to compare it to.

When hashing, we did not have to match on 1448 byte boundaries. By having a smaller boundary, this would allow our experiment to work on different networks that might have smaller payload sizes than the network that we were using. When hashing along boundaries, we could not hash the first 1024 bytes of data, then simply move on to the next 1024 bytes,

and so on. The data could not be aligned properly. Due to the variation of the HTTP header and MTU, the hash of the first payload would not be able to match any hash that we created. Table 4.3 shows how many hashes were created in each database.

Experiment	Files	Block Size	Hashes
1	50	1448	6678056
1	50	1200	6695346
1	50	1024	6712048
2	990	1024	412551239

Table 4.3: Hashes in each hashdb database created.

Explained are the amount of hashes that are in each of the databases created for this thesis. The table explains each experiment run, how many files are in each database, the size of the data that is hashed and the amount of hashes in the database.

4.5 Building the Target Databases

We created the databases using a combination of hashdb and bulk_extractor. As mentioned earlier, (see Section 2.4.1) bulk_extractor is capable of processing large amounts of data at once. We can take advantage of this by using hashdb as a scanner feature with a bulk_extractor instance. Using the hashdb scanner, bulk_extractor can add hashes of the data from bulk_extractor's run to a specified hashdb database.

Multiple options are necessary to correctly configure bulk_extractor. The "-E" option followed by a scanner tells bulk_extractor to use no other scanner than the one that you have provided. The "-S" option allows the user to pass configuration options to individual scanners. Figure 4.3 shows an example command line invocation of bulk_extractor. The "-S hashdb_mode=import" is used to import the data as hashes. The "-S hashdb_byte_alignment=1" and "-S hashdb_step_size=1" are used to indicate the size of byte_alignment and step_size. The step_size is the interval at which data is hashed. The byte_alignment determines what byte size the data is hashed along. When the step_size is 1, the byte_alignment also must be 1. When working with the byte_alignment, that value has to be divisible by the step_size. In many instances, the byte_alignment is large to reduce the space required to store the hashes in the database. The "-S hashdb_block_size=1024" option determines how many bytes we are using to create each MD5 hash, for example 1,024 bytes. The "-R <input directory>" option is to provide bulk_extractor with a directory that

it can use to process data. The "-o <output directory>" option is to give the output directory for the hashdb database.

```
bulk_extractor -E hashdb -S hashdb_mode=import -S hashdb_byte_alignment=1
-S hashdb_step_size=1 -S hashdb_block_size=1024 -R <input directory>
-o <output directory>
```

Figure 4.3: Instance of bulk_extractor run.

The time required to constructed the database depends on the number of hashes inserted and the available resources. When using these parameters with bulk_extractor, we were able to build a database of the fifty files from Sample 1. This is the control hashdb database. We were able to use this later in the experiment to know which hashes and files existed in the database.

In order to calculate the number of hashes h that we expect our database to contain, we use the following equation, where n is the number of files, x is the average file size, k is the block size, and y is the step size.

$$h = n\left(\frac{x-k}{y}+1\right)$$

We then built a database containing 990 files, initially using the aforementioned options. However, this database took a long time to build and bulk_extractor timed out because hashing each block repeatedly took more time than bulk_extractor expected. Another reason bulk_extractor timed out was the size of the files that the program was dealing with. By default, a thread in bulk_extractor cannot run for more than 60 minutes. Some of the threads ran longer than 60 minutes due to the amount of data that had to be processed by bulk_extractor. To resolve this problem, we divided the 990 files into 12 folders. This was so bulk_extractor did not have to process as much information as it did with 990 files in an instance. One folder only had one file due to the file's large size. bulk_extractor struggled to process this one file because of its size. It had to be placed in a directory all on its own in order to be processed. When processing each individual file directory, it took a little over an hour on average in order to build the databases for each of the 12 folders. By only opening the hashdb database once in the program, it brought down the program run time

by about 75%.

A master database full of hashes from the 990 file directory had to be created. The smaller databases were already built, the next step was to combine them. There were 12 different databases that were built. Each of these 12 databases had to have their hashes combined into one master database. hashdb has a feature that allows you to copy the hashes from one database into another. This was used 12 times to ensure that each of the 12 directories that were hashed would be inserted into this database. When all of the hashes were combined, it equated to about 415 million hashes in one master database.

```
hashdb add <source hashdb directory> <destination hashdb directory>
```

In order to determine the master database was working properly, we ran test traffic sets through the master database to ensure that all of the files we expected to find we did in fact find. We were able to run Experiment 1's data through the master database as a test, and successfully find all fragments.

4.6 Capturing Traffic

We started with a server that hosted files from Govdocs (mentioned in Section 2.3.1). The next step was to collect the data. We used a Python program to select 100 sequential files from Govdocs. This Python program sets up a web connection to download each of the files given to record the filename, HTTP code, IPv4 of web server, source port, destination port, and size of the file downloaded. While this program was selecting 100 sequential files, we had an instance of tcpdump running to capture the data the Python program was collecting and put the packets into a pcap file. This pcap file contained the packets that were downloaded using the Python program. The first pcap file downloaded was used in Experiment 1.

After we collected a 100 file capture, we then collected the data that we would use for Experiment 2. For our second experiment, we wanted to collect data that would be representative of a network that has many packets flowing through it. The second experiment focused

on downloading two 991 file directories and storing their content in pcap files. Figure 4.4 shows how the files were downloaded onto the Mac laptop.

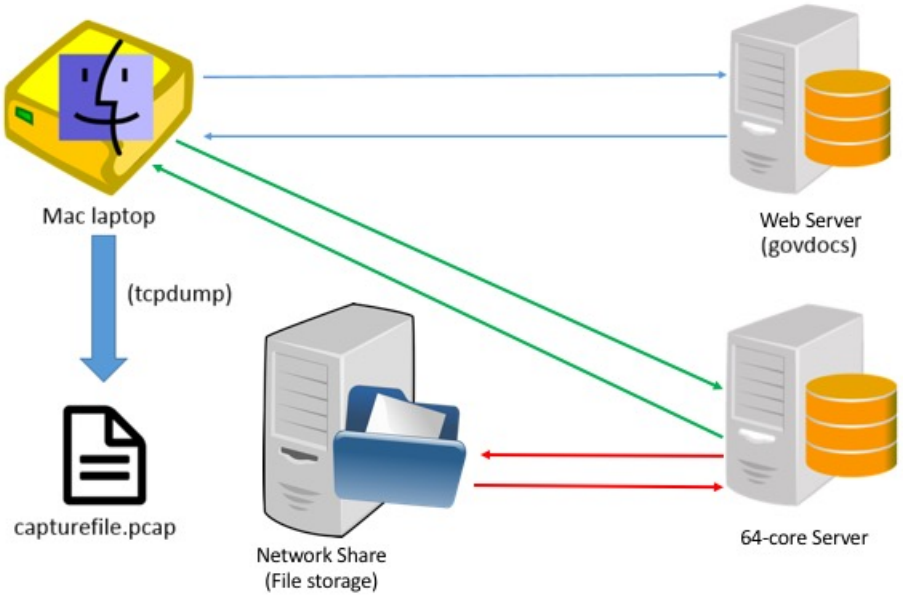

Figure 4.4: Diagram of server setup, how files were downloaded.

We created a Python program to analyze each packet, matching packets with an individual file ID from govdocs to produce a list of packets and the file IDs that are associated with each packet. The program requires three things to input: the pcap file, the file IDs for the different files, and the file generated during the download process explaining each file and what flow tuple it corresponds to. The program looks at each flow tuple and determines which file ID should belong to each packet in the flow tuple. Then, once the files are correctly identified, the output from the program is a list of packets and the file ids that correspond to them. This file is the ground truth; where we know exactly what file is associated with what packet.

4.7 Analyzing Traffic for Target Data

Figure 4.1 displays what has occurred so far in the thesis. The files travel into network traffic via HTTP. Once the files are inside the network traffic, each packet's payload is then pulled out and the first 1,024 bytes of that payload is hashed. At the same time, the database of blacklisted files contains hashes of file pieces. Once the hashes are in the database, the

database is then queried to look up the payload's hash to see if it exists inside the database. The results are then indicated, whether the hash of packet payload existed in the database or not.

For each of the pcaps that we captured, we run the program that we created to get the hashes of each payload for each packet. Once these hashes are created, they will then be looked up in our blacklist database to see if the hash generated by a packet's payload matches up against something in the database. If it does, that might indicate that target data was downloaded. However, if we did not find files that we know should be in the traffic, we investigated to determine why those files were not found. Our program used a "scanning" function that is included with hashdb's API. This opened up a hashdb database and we passed in the hash of the packet's payload to see if it was in the database. If the payload's hash existed, a "1" was output, indicating that a matching hash was found, along with the packet number in which the payload's hash was found. If the payload's hash did not match up with anything in our database, the packet was marked with a "0" for false. In addition, we also output the JSON string of the hash's content that was found. This includes file directory, block size, entropy, and other facts about the file and its hash. This is so we can identify which file was found. In Table 4.4, there is an explanation of the files downloaded, how many packets we are looking for, and how many packets are in the overall traffic.

Experiment	Capture	Files	Target Packets	Total Packets
1	1	100	4575	9635
2	1	200	31284	54822
2	2	200	33691	79873
2	3	200	22890	158451
2	4	200	41956	72481
2	5	199	21327	36981
2	6	200	24626	52186
2	7	200	30963	52495
2	8	200	26342	73342
2	9	200	25502	44323
2	10	183	20576	47303

Table 4.4: The statistics of each experiment and file capture.

Each of the captures we completed is associated with an experiment. We noted how many files were downloaded in each capture, as well as the amount of target packets and total packets in each capture.

If the hashes were found, that means the method we used worked. However, if the hashes were not found, then we needed to go back and investigate why the hashes were not found

and what parameters can be changed in order to find the hashes of the payloads inside the hashdb database.

4.8 Scoring Our Results

We are going to be looking for four things when we look through the traffic: true positives, false positives, true negatives, and false negatives. True positives are those hashes where we know that target data is in the packet and the packet will show as having target data in it. False positives are those hashes where the packet shows up as having something we want in it, but we know does not contain the content in the packet that we are looking for. True negatives are those hashes that do not find a match, and content is not a payload that we are looking for. False negatives are those hashes where we do not find a match, and we know that the hash should be associated with the packet. These categories are summarized as follows:

- Positive: A hash of a packet payload appears in our target database.
- Negative: A hash of a packet payload does not appear in our target database.
- True positive: A hash of a packet payload appears in our target database AND the packet comes from transfer of a target file.
- True negative: A hash of a packet payload does not appear in our target database AND the packet does not come from transfer of a target file.
- False positive: A hash of a packet payload appears in our target database BUT the packet does not come from transfer of a target file.
- False negative: A hash of a packet payload does not appear in our target database BUT the packet does come from the transfer of a target file.

The overall goal is to produce only true positives and true negatives. If there were a high number of false positives, it required some fine tuning to ensure that we were looking to find the target data, not the noise data. If there were a high number of false negatives, that meant that there was a problem with the program that we created.

In order to determine what the true positive, true negative, false positive and false negative values were, the ground truth file and the output from the traffic parser program were put into a Python scoring program. This program is written to read in the two files, then determine

whether or not the packets match between the traffic and the ground truth. The program then determines the positives and negatives, then displays them on the screen. As long as the payload is determined to be true and match a hash in the database, and have a file id be with the same packet, that is considered a true positive.

Once the tests were complete, we focused on analyzing problems and refining our method to bring down the false positives and false negatives that we found within our traffic.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Results and Analysis

In our first experiment, we find that the smallest block size tested (1024 bytes) gave the best results. In our second experiment, we use this block size and perform 10-fold cross validation on Govdocs Sample 2 to determine the effectiveness of our method.

We used folds, or parts of the experiments, to determine the results using the programs and methods described in Chapter 4. The results for each experiment showed how many true positives, true negatives, false positives, and false negatives were associated with each fold.

5.1 Experiment One: Testing the Impact of Hash Block Size on Precision and Recall with Govdocs Sample 1

After examining the results of our experiments with 1448, 1200, and 1024 bytes, we determined that 1024 byte fragments produced the best outcome. In Table 5.1, the precision and recall are noted for each result with different block size.

As indicated in Figure 5.1, the first evaluation of our initial experiment with hashes of 1,448 byte segments yielded 4479 true positives, 4717 true negatives, 25 false positives, and 96 false negatives.

The false positives are attributed to two PDF files that have similar content. The target PDF file was a preliminary report, 002021.pdf, and the PDF file that was generated false positives was a final report, 002086.pdf. Figures 5.2 and 5.3 show the content of the two PDFs. The two sizes for the files are 754KB and 1.2MB respectively. The other file that generated false positives was 002096.pdf. This file is not similar to the other two. There

Block Size	Precision	Recall
1448	97.9%	99.4%
1200	98.0%	99.4%
1024	98.2%	99.4%

Table 5.1: Precision and Recall for each of three potential outcomes.

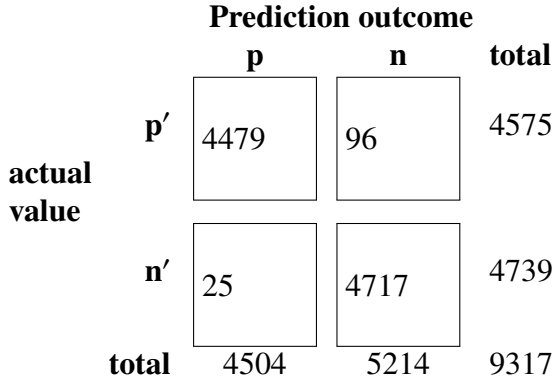

Figure 5.1: Confusion matrix, hashes generated from 1448 byte segments

were 23 false positives from 002086.pdf and 2 from 002096.pdf.

After further investigating, we determined that the false negatives could be explained by two features of the file transfer mechanism we employed. First, HTTP header data inside the first packet of content for each transferred file prevented hashes of that packet from matching hashes in our target database. Because the HTTP header is of arbitrary length, this is something that we had to account for when looking for the hashes of the first packets in a stream. All packets after the HTTP header were found the majority of the time. Second, most of the files did not end exactly on 1448 byte alignment, meaning that many of the final packets in a flow did not match. There were 50 header packets and 46 final packets the program could not find.

Even though we found 46 final packets that the program could not find, that still left 4 packets unaccounted for. All payloads were less than 1448 bytes, as indicated in Table 5.2. However, there were two files, 002028.html and 002007.html, that had file size less than 1448. The header and final packet were the same in the stream. The other two final packets, from files 002008.html and 002047.html, did not appear either. In this instance, both files had a final payload size less than 100 bytes, meaning that our Python script had an issue attempting to identify small packet sizes.

As indicated in Figure 5.4, the second evaluation of the first experiment with hashes of 1,200 byte segments yielded 4483 true positives, 4716 true negatives, 26 false positives, and 92 false negatives.

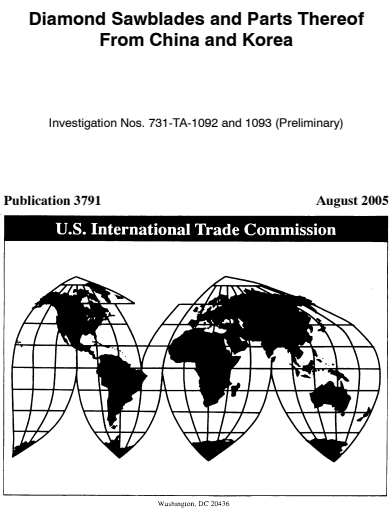

Figure 5.2: First page of 002021.pdf

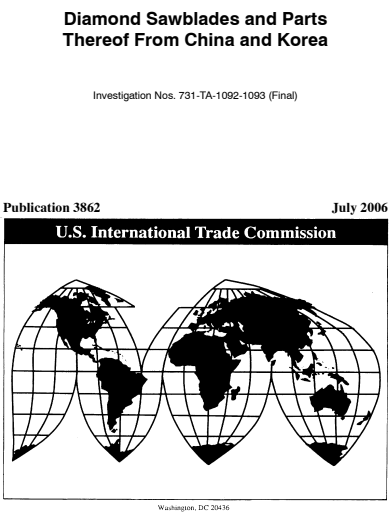

Figure 5.3: First page of 002086.pdf

Block Size	1448 Bytes	1200 Bytes	1024 Bytes
# of Fragments below Block Size	50	46	37

Table 5.2: Fragments below block size.

This table demonstrates the final fragment sizes and their relation to block sizes used.

		Prediction outcome p	Prediction outcome n	total
actual value	p′	4483	92	4575
	n′	26	4716	4742
	total	4509	4808	9317

Figure 5.4: Confusion matrix, hashes generated from 1200 byte segments

Compared with the 1,448 byte segment hashes, the 1,200 byte segment hashes provided us with four fewer false negatives, one more false positive, and one less true negative. Packet numbers 85, 2880, 2918 and 2944 showed as true positives instead of false negatives. This is because these packets were the final ones in their respective streams. Packet 7592 showed

as an additional false positive instead of a true negative. Packet 7592 has a payload size of 1448, but the hash of the first 1,200 bytes of that payload was recognized as belonging to 002021.pdf. There were 24 false positives from 002086.pdf and 2 false positives from 002096.pdf. Therefore, there is an additional false positive added to 002086.pdf and the same amount as found in 002096.pdf.

As indicated in Figure 5.5, the final evaluation of the first experiment with hashes of 1,024 byte segments yielded 4492 true positives, 4715 true negatives, 27 false positives, and 83 false negatives.

		Prediction outcome		
		p	**n**	**total**
actual value	**p′**	4492	83	4575
	n′	27	4715	4742
	total	4519	4798	9317

Figure 5.5: Confusion matrix, hashes generated from 1024 byte segments

Compared with the 1,200 byte segment hashes, the 1,024 byte segment hashes provided us with nine fewer false negatives, one more false positive, and one less true negative. Packet numbers 76, 211, 273, 344, 2033, 2040, 2725, 2871 and 3886 showed as true positives instead of false negatives and packet 8391 showed as an additional false positive instead of a true negative. Packet 8391 has a payload size of 1448, but the hash of the first 1,024 bytes of that payload was recognized as belonging to 002021.pdf. There were 25 false positives from 002086.pdf and 2 false positives from 002096.pdf. There was once again an additional false positive that belonged to 002086.pdf, and the amount in 002096.pdf remained the same. The same two false positives for 002096.pdf were found for each of the three evaluations. This is probably an instance where the data contained in the file matched to content in another one of the files that was in the database. With regards to the false positives of 002086.pdf, 002021.pdf and 002086.pdf are so close to each other, it makes sense that more hashes continue to be found. As mentioned earlier, 002021.pdf was

a preliminary report, while 002086.pdf was a final report, so they both contain a lot of the same information and data.

After running the different evaluations of the first experiment, we were able to bring down our false negatives and increase our true positives. The number of false negatives dropped from 96 to 83 because 13 packets at the end of file transfer flows had payloads that were greater than 1,024 bytes, but less than 1,448 bytes. We also noticed a small increase in the false positives. This is probably because the same quantity of matching data was split into slightly more fragments when the fragment size decreased. All of the false positives came from 002021.pdf.

In addition, we found that our program was consistently having difficulty in finding 4 packets. Our numbers consistently showed that we were off by 4 false negatives every time. There were two factors: small files and small packet payloads. There were two files that were found with an overall size less than 1448 bytes: 002007.html and 002028.html. In those two packet streams, the header and final packet was the same one. For small packet payloads, there was an issue with our program. It had difficulty identifying true negatives on packets smaller than 100 bytes in size: 002008.html had a final payload size of 36 bytes and 002047.html had a final payload size of 19 bytes.

5.2 Experiment 2: 10-fold Validation with Govdocs Sample 2

Results from evaluations of the first experiment led us to conclude that 1,024 byte segments was a reasonable choice of block size. When running these tests, we found that expanding the size of the master database had minimal impact on the runtime of our analysis. Though further testing is required to measure performance, this preliminary observation was encouraging. Table 5.3 shows the amount of positives, both true and false, negatives, both true and false, precision, and recall for each of the folds. Figure 5.6 shows the Govdocs Sample 2 confusion matrix.

The precision averaged 98.3% and the recall averaged 99.2% throughout the 10 folds.

The majority of our false positives were caused by PDFs, HTML webpages, Word documents, PowerPoint presentations, Excel spreadsheets and PostScript files. PDF files contain

Fold	True Positives	True Negatives	False Positives	False Negatives	Total Packets	Precision	Recall
1	31020	22568	333	264	54822	99.0%	99.2%
2	33438	44999	542	253	79873	98.4%	99.2%
3	22672	134473	437	218	158451	98.1%	99.0%
4	41745	29771	473	211	72481	98.9%	99.5%
5	21112	14822	186	215	36981	99.1%	99.0%
6	24416	26647	266	210	52186	98.9%	99.1%
7	30676	20011	890	287	52495	97.2%	99.1%
8	26221	45309	959	211	73342	96.5%	99.2%
9	25271	17759	416	231	44323	98.4%	99.1%
10	20386	25950	195	190	47303	99.1%	99.1%

Table 5.3: The packet results for each fold in Experiment 2.

The precision and recall are fairly consistent for the 10 folds. The only two that were lower than 98% had a high amount of false positives. The recall did not go below 99%. Fold 5 was smaller than the others. The size of the pcap was smaller by 20MB to the next closest capture, indicating that the download did not consist of many large files. Fold 3 had a high amount of packets because there was a 133MB Excel spreadsheet that was downloaded.

		Prediction outcome		
		p	**n**	**total**
actual value	**p′**	276957	2290	279247
	n′	4697	382309	387006
	total	281654	384599	666253

Figure 5.6: Confusion matrix, hashes generated 1024 bytes from all ten capture files

similar data in certain parts of the file, because it is a standardized file format. Microsoft office documents normally have the same content at the start of each file. PostScript files are similar to PDF files. There were about 100 files of the target data that were PDF files in the target data. In the false positives, about 280 noise files were found that incorrectly matched to data in blacklisted files. False positives by file extension totals are found in Table 5.4.

The reason that many fragments of PDF files in the 002 govdocs directory that matched to fragments in the 013 govdocs directory is that multiple PDF files had the same data. There

Extension	False Positives
PDF	2,361
PS	641
DOC	537
PPT	378
XLS	255
HTML	252
CSV	190
JPG	65
PPS	13
TEXT	3
XML	2

Table 5.4: Number of false positives per file extension.

were multiple cases where there were PDF files that were a preliminary report, and then a final report similar to the preliminary report, but not quite the same. Some Microsoft Office documents also contained similar content. By having identical 1,024 byte runs that exist, these files confused our matching algorithm and produced false positives.

Out of all the false negatives, 1,633 packets were pieces of files immediately following HTTP headers, the last packet in a stream, or the header and final packet in the stream, as some files were small and did not have data in more than one packet. The HTTP header caused false negatives because the packet content had to exactly match 1,024 byte hashes in files inside the directory. False negatives by file extension totals are found in Table 5.5.

There were 657 false negatives that appeared that did not match up with the first and last packets in a stream. The majority of these false negatives were from Word documents, PowerPoint presentations, and Excel spreadsheets. When searching Microsoft Office application documents, the bytes 0x00 and 0xff for example, were seen in the first 1024 bytes of most of their false negatives. After hashing the 1024 0x00 and 0xff bytes, it appears that those hashes were not included in the hashdb database that was built. In addition, there were instances of an HTML file, a PDF document, a PostScript document, and two JPG images where there were more than two false negatives. The HTML file had a duplicate packet that was sent, the PDF document had the same 0x20 byte for its first 1024 bytes in the hash, the PostScript document had the same 0x46 byte for its first 1024 bytes for 21 packets, one JPG had the same 0x00 byte for two false negative packets, and the second

Extension	False Negatives
HTML	472
DOC	461
PDF	412
PPT	305
TXT	271
JPG	151
XLS	99
PS	35
XML	19
GIF	17
CSV	10
UNK	10
GZ	6
LOG	5
TEXT	5
DBASE3	4
TEX	3
JAVA	2
WP	2
GLS	1

Table 5.5: False negatives by file extension.

JPG had a duplicate packet that was sent.

There were many false negatives with PDF and HTML files, but those were the first and last packets per traffic stream. There were also a large amount of TXT and JPG files that caused multiple false negatives with the first and last packets in their respective traffic streams.

CHAPTER 6:
Conclusion and Future Work

This chapter discusses our conclusions and proposed future work.

6.1 Conclusion

Our goal was to develop a technique to find target data in network traffic without rebuilding a network stream by hashing the payload of packets. We tested different window sizes to determine the size that allows us to find the most packets that contained target data. The ideal window size that we found was 1,024 bytes. We think this is the ideal size because it maximized the amount of true positives, while also reducing the amount of false negatives.

We were able to identify packets that were in our target data without reconstructing traffic streams. While identifying these packets, we were also able to identify packets accurately that would determine whether data was exfiltrated. Our average precision was 98.3% and our average recall was 99.2%. We were able to develop a program that would generate the results within a few minutes. We were also able to put over 415 million hashes into one database.

We believe this is a feasible approach to finding pieces of files in network traffic over HTTP. We had a high percentage of finding pieces of files that we were hoping to find. This method would not work over HTTPS, unless there was an ability to break SSL. However, there were some issues, such as finding a hash with the same 1,024 bytes throughout the length of payload. When the same byte was present for the first 1,024 bytes of payload, hashdb did not take a hash of that data. That caused a large amount of false negatives.

We were able to work towards our goal of creating a system that would look at a packet's individual payload, take a hash of that payload, then compare the results to a database that contains the hashes of blacklisted files. The program was a prototype implemented in Python and we are leaving the development and testing of a speed-optimized version for future work.

While testing bulk_extractor, we found a bug that prevented users from specifying the

amount of time to give a thread before it timed out. In addition, there was also the testing of new versions of bulk_extractor and hashdb. These programs, which were critical to the completion of this thesis, were constantly used and 98 to confirm they were properly.

6.2 Future Work

Our experiment demonstrates that a simple approach of hashing the beginning of the payload allows us to reliably determine whether packet payloads were part of our target data. However, to show that this method improves over the state of the art, we need to test performance to determine how much throughput this approach can handle. We did not run this system in real-time, and performed the majority of our tests post-mortem, so collecting these metrics remains for future work.

Furthermore, more testing needs to be done to determine how large the target set can be given the available memory resources. We were only able to sample 1,982 files of the one million files govdocs corpus. In the future, experiments could be done with more blacklist files to test the limits of our approach. The hashdb database is capable of handling many more hashes than what we have processed. However, more work needs to be done to make it possible to ingest larger files. Due to bulk_extractor's issue with only having a thread run for 60 minutes, some of the data might not be able to be processed in time. Splitting up the data into chunks and hashing it was the best way to complete this task. Another option would be to update to the latest version of bulk_extractor.

To complete the implementation of an approximate matching scheme, we would need to determine the probability that, giving some number of packet matches against the payload, a targeted file could have been or is in the process of being exfiltrated off of the network. This could be determined in part from the results of our experiment, but a more extensive test using real traffic would make these results more representative. For this reason, our approach should be tested on a more realistic dataset. That would hopefully prove that this system would be able to work in a larger environment than the one we tested in.

Once we understand the probability that a packet match correlates to the presence of a target file, further experiments should be run in which we test different thresholds for number of matches or the proportion of the target file matched. Precision and recall should then be

evaluated again for different threshold values. These results, in addition to the throughput metrics, should then be compared to those of "state of the art" data exfiltration prevention systems. This will help determine whether or not our system would be able to compete against other systems like it.

In our experiment, packets that contained the HTTP header always registered as false negatives. One way to fix this problem is to remove of the HTTP header. The HTTP header is of arbitrary length, so this would involve additional parsing. That would help increase the number of true positives.

If we were able to fix the alignment problem, we could reduce space needed to store files in our database. We are restricted because the hashes have to be overlapped with 1-byte intervals in order to ensure that we captured all possible hashes. A different traffic parser would be necessary in order to complete this task.

Another way the HTTP header problem could solved would be making decisions on groups of packets instead of an individualized approach. This approach would look at how many packets for each file there should be in the network traffic, and then make a decision regarding whether or not the file existed in the traffic, based on the same threshold of matches.

This program solely focused on HTTP traffic, but it could be extended to other protocols. For example, future versions could look at e-mail, FTP, SQL, and SMB. SQL and SMB can be easily encrypted, making it more difficult to determine if files are in a set of traffic, unless the traffic is decrypted. FTP is not encrypted by default, meaning that the data transferred over that protocol can be observed and checked against a blacklist. However, if FTPS, or secure FTP is used, then that data will be encrypted and difficult to observe. Our method should be tested on these protocols in the future to determine its usefulness in identifying pieces of data in network traffic.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] M. Cruz. (2013). Data exfiltration in targeted attacks. [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/data-exfiltration-in-targeted-attacks/

[2] P. Finn and S. Horwitz, "U.S. charges Snowden with espionage," *The Washington Post*, vol. 21, 2013.

[3] E. Nakashima. (2015). Hacks of opm databases compromised 22.1 million people, federal authorities say. [Online]. Available: https://www.washingtonpost.com/news/federal-eye/wp/2015/07/09/hack-of-security-clearance-system-affected-21-5-million-people-federal-authorities-say/

[4] K. Zetter. (2014). Sony got hacked hard: What we know and don't know so far. [Online]. Available: https://www.wired.com/2014/12/sony-hack-what-we-know/

[5] M. Guri, A. Kachlon, O. Hasson, G. Kedma, Y. Mirsky, and Y. Elovici, "Gsmem: data exfiltration from air-gapped computers over gsm frequencies," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 849–864.

[6] P. Boutin, "Sneakernet redux: Walk your data," 2002. [Online]. Available: http://www.wired.com/culture/lifestyle/news/2002/08/54739

[7] S. Dharmapurikar and V. Paxson, "Robust TCP stream reassembly in the presence of adversaries." in *USENIX Security*, 2005.

[8] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2003, pp. 325–336.

[9] J. Postel et al., "RFC 791: Internet protocol," 1981. [Online]. Available: https://tools.ietf.org/html/rfc791

[10] D. Murray, T. Koziniec, K. Lee, and M. Dixon, "Large MTUs and internet performance," in *High Performance Switching and Routing (HPSR), 2012 IEEE 13th International Conference on*. IEEE, 2012, pp. 82–87.

[11] C. Hornig, "RFC 894: Standard for the transmission of IP datagrams over Ethernet networks," 1984. [Online]. Available: https://tools.ietf.org/html/rfc894

[12] M. Bishop, S. Engle, D. A. Frincke, C. Gates, F. L. Greitzer, S. Peisert, and S. Whalen, "A risk management approach to the 'insider threat'," in *Insider threats in cyber security*. Springer, 2010, pp. 115–137.

[13] The Critical Security Controls for Effective Cyber Defense. [Online]. Available: https://www.sans.org/media/critical-security-controls/CSC-5.pdf

[14] D. E. Denning, "An intrusion-detection model," *Software Engineering, IEEE Transactions on*, no. 2, pp. 222–232, 1987.

[15] A. Giani, V. H. Berk, and G. V. Cybenko, "Data exfiltration and covert channels," in *Defense and Security Symposium*. International Society for Optics and Photonics, 2006.

[16] R. Oppliger, "Internet security: Firewalls and bey," *Journal A*, vol. 39, no. 2, pp. 20–27, 1998.

[17] Stateful vs. stateless firewalls. [Online]. Available: http://www.inetdaemon.com/tutorials/information_security/devices/firewalls/stateful_vs_stateless_firewalls.shtml

[18] S. Northcutt et al., "Chapter 3: Stateful firewalls," *Inside Network Perimeter Security, 2nd Edition, Pearson*, pp. 55–86, 2005.

[19] T. Porter, "The perils of deep packet inspection," *Security Focus*, 2005.

[20] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *Digital Investigation*, vol. 6, pp. S2–S11, 2009.

[21] Snort - Network Intrusion Detection & Prevention System. [Online]. Available: https://www.snort.org/

[22] Suricata | Open Source IDS / IPS / NSM Engine. [Online]. Available: https://suricata-ids.org/

[23] F. Breitinger, B. Guttman, M. McCarrin, V. Roussev, and D. White, "Approximate matching: definition and terminology," *NIST Special Publication*, vol. 800, p. 168, 2014.

[24] S. Garfinkel, A. Nelson, D. White, and V. Roussev, "Using purpose-built functions and block hashes to enable small block and sub-file forensics," *Digital Investigation*, vol. 7, pp. S13–S23, 2010.

[25] M. O. Rabin *et al.*, *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[26] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic clustering of the web," *Computer Networks and ISDN Systems*, vol. 29, no. 8, pp. 1157–1166, 1997.

[27] V. Roussev, "Sdhash version 3.4," 2013. [Online]. Available: http://roussev.net/sdhash/sdhash.html

[28] S. L. Garfinkel and M. McCarrin, "Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb," *Digital Investigation*, vol. 14, pp. S95–S105, 2015.

[29] S. Garfinkel, "bulk_extractor," https://github.com/simsong/bulk_extractor, 2016.

[30] B. Allen, "hashdb," 2016. [Online]. Available: https://github.com/NPS-DEEP/hashdb

[31] S. Jarvenpaa and B. Ives, "Digital equipment corporation: The internet company (a)," *Boston (Harvard Business School Case Study)*, 1994.

[32] B. Cheswick, "The design of a secure internet gateway," in *USENIX Summer Conference Proceedings*. Citeseer, 1990.

[33] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23, pp. 2435–2463, 1999.

[34] N. Schear, C. Kintana, Q. Zhang, and A. Vahdat, "Glavlit: Preventing exfiltration at wire speed," in *Proceedings of the 5th ACM Workshop on Hot Topics in Networks (HotNets-V)*, Irvine, CA, November 2006.

[35] V. Roussev, "Hashing and data fingerprinting in digital forensics," *Computing in Science and Engineering*, vol. 7, no. 2, pp. 49–55, 2009.

[36] K. Foster, 2012. [Online]. Available: https://calhoun.nps.edu/handle/10945/17365

[37] C. Shields, O. Frieder, and M. Maloof, "A system for the proactive, continuous, and efficient collection of digital forensic evidence," *Digital Investigation*, vol. 8, pp. S3–S13, 2011.

[38] A. Larbanet, J. Lerebours, and J. David, "Detecting very large sets of referenced files at 40/100 gbe, especially mp4 files," *Digital Investigation*, vol. 14, pp. S85–S94, 2015.

[39] K. Zetter. (2014). Hacker lexicon: What is an air gap? [Online]. Available: https://www.wired.com/2014/12/hacker-lexicon-air-gap/

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California