

Efficient and Effective Response to Computer Security Incidents and Cybercrime

System-level defenses against USB-based attacks

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

by

Ing. Dipl.-Ing. Sebastian Neuner, BSc.

Registration Number 1227217

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.Do. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl

The dissertation has been reviewed by:

Engin Kirda

Stefan Brunthaler

Vienna, 1st June, 2019

Sebastian Neuner

Efficient and Effective Response to Computer Security Incidents and Cybercrime

System-level defenses against USB-based attacks

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Ing. Dipl.-Ing. Sebastian Neuner, BSc.

Matrikelnummer 1227217

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv.Doiz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl

Diese Dissertation haben begutachtet:

Engin Kirda

Stefan Brunthaler

Wien, 1. Juni 2019

Sebastian Neuner

Declaration of Authorship

Ing. Dipl.-Ing. Sebastian Neuner, BSc.
Dr.-Kaserer-Weg 4
6170 Zirl
Austria

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 1st June, 2019

Sebastian Neuner

Acknowledgements

I could not have written this thesis without the support of many, many people.

First and foremost, I wish to thank my advisor Edgar Weippl, for making this thesis possible and providing me with the freedom to pursue my research ideas. I would also like to thank my former colleagues at SBA Research– especially Martin Schmiedecker, who guided me through the early stages of my scientific journey; Markus Klemen, for making my master’s degree possible; and both Stefan Brunthaler and Kristoffer Kleine for in-depth technical discussions and joint rants on a regular basis.

At TU Wien, there is also one person that helped me during my master’s as well as during my Ph.D. studies: Reinhard Pichler, the vice dean for the master’s program in computer science.

Furthermore, I would like to thank Engin Kirda, who invited me to work with him and his research team at Boston’s Northeastern University. Those months of research guided me towards focusing my further research on USB security.

It is tough to work on a thesis on top of a fulltime job, especially if you must fly to another country for different matters several times a year. All of this would not have been possible without the support of my amazing colleagues at Google.

Among the many people that supported me and worked with me on hard problems, one person deserves an extraordinary mention: Artemios Vogiatzis, who helped and guided me through many decisions and most of my scientific work, even if that meant working until 5 am. He always had advice at hand and listened to complaints, and I am honored to consider him a friend now.

In my personal life, I consider myself one of the happiest people on earth for having a family that provided me with never-ending support. Without them, I would not be the person I am now, I would not work as a security engineer at a company I always wanted to work for, and none of my studies would have been possible. I thank my family: my mother, Christine, my father, Fritz, and both of my brothers, Clemens and Florian.

Finally, it all boils down to one person in my life who has always been there when I needed her. I dedicated my bachelor’s thesis to my girlfriend and my master’s thesis to my fiancée, and I am more than happy that I can dedicate this doctoral thesis to my wife, Patricia. Thank you for everything. I love you.

Abstract

Security of computer systems is designed and engineered over the years using castle analogies and perimeter defenses. Network ports are well-monitored and protected entry points for system security and network-based defenses steadily improve over time. In contrast, USB ports remain more often than not an unmonitored entry point for malevolent actors. In this setting, malicious or even criminal actions can be launched internally, beyond the reach of network perimeter defenses, rendering the operating systems that interface the USB ports as the last line of defense. Even if such actions are detected in first place, incident response teams are overwhelmed with the amount of information that must be analyzed to detect, mitigate and recover from the attacks.

In this thesis, we design, engineer, and evaluate holistic operating system-level defenses against USB-based attacks. We focus on three attack surfaces: (i) data leakage through unmanaged USB storage media files; (ii) information hiding in filesystem metadata; and (iii) system compromise through fake USB devices with modified firmware.

The contributions of this thesis significantly improve both the resilience of modern operating systems against USB attacks and the response time once an incident arises in an enterprise environment. We also invent automated defense techniques that can proactively protect end users without involving them in the trust decision.

Kurzfassung

Die Sicherheit von Computersystemen wird seit jeher nach dem Vorbild von Burgen entworfen und entwickelt – ein Angreifer ist dabei gezwungen Perimeter für Perimeter zu überwinden. Netzwerkanschlüsse sind hervorragend abgesicherte und überwachte Eintrittspunkte in die Sicherheit von Systemen, da sich netzwerkbasierte Sicherheitsstrategien im Laufe der Zeit konstant verbessert haben. Im Gegensatz dazu, bieten USB Anschlüsse häufig unüberwachten Zutritt für Angreifer. Hierbei ist es Angreifern möglich bössartige oder sogar kriminelle Handlungen aus dem Inneren des Netzwerks heraus zu starten, außer Reichweite eines schützenden und überwachten Perimeters. Dieser Umstand macht Betriebssysteme, welche die Daten angeschlossener USB Geräte verarbeiten zur letzten Verteidigungslinie. Jedoch, auch im Fall einer Erkennung einzelner Ereignisse sind sogenannte Computer Security Incident Response Teams (CSIRTs) überschwemmt mit der Flut an Daten, welche ausgewertet werden müssten, um tatsächliche Angriffe zu erkennen, zu mitigieren und die Sicherheit der Systeme und Netzwerke wiederherstellen zu können.

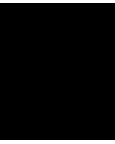
In dieser Arbeit werden allumfassende Verteidigungsstrategien gegen USB-basierte Angriffe auf Betriebssystemebene entworfen, entwickelt als auch evaluiert. Der Fokus liegt dabei auf drei Angriffsflächen: (i) Datenexfiltration durch unverwaltete Dateien auf USB Massenspeichergeräten; (ii) Datenverschleierung in Dateisystemmetadaten; als auch (iii) Kompromittierung von Systemen durch USB Geräte mit veränderter Firmware.

Die hier vorliegende Arbeit verbessert die Widerstandsfähigkeit moderner Betriebssysteme gegen USB-basierte Attacken und die Erkennung von derartigen Ereignissen in einem Unternehmensnetzwerk signifikant. Zudem wird eine automatisierte Methode zur Verhinderung dieser Art von Attacken beschrieben, welche den Benutzer nicht in die Sicherheitsentscheidung miteinbezieht.

Contents

Abstract	ix
Kurzfassung	xi
Contents	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Aims	3
1.3 Methodology	4
1.4 Main Results	5
1.5 Structure	7
2 Enhanced File Whitelisting for Incident Response	9
2.1 Background	10
2.2 Improvements to the Forensic Process	11
2.3 Evaluation	15
2.4 Results	18
2.5 Conclusion	22
3 Improving Incident Response with Sub-file Hashes from Open Sources	25
3.1 Background	26
3.2 P2P Networks for Hash Values	27
3.3 Evaluation	31
3.4 Results	32
3.5 Discussion	35
3.6 Conclusion	36
4 Filesystem Timestamp Steganography Detection	37
4.1 Background	38
4.2 Timestamps in Modern Filesystems	40
4.3 Steganography Based on File Timestamps	43
4.4 Evaluation of the TOMS System	47
4.5 Experimental System Validation	52
	xiii

4.6	Steganographic Capabilities of TOMS	55
4.7	Implications for Forensics Analysis	66
4.8	Conclusions	73
5	System-level Detection of Keystroke Injection Attacks	75
5.1	Background	76
5.2	A Novel Approach for Detection of USB-based Keypress Injection Attacks	81
5.3	Proof-of-concept Implementation and Evaluation	88
5.4	Enriching Incident Response for BadUSB-like Attacks	94
5.5	Conclusions	96
6	Conclusions and Future Work	99
	List of Figures	103
	List of Tables	105
	Bibliography	107



Introduction

1.1 Motivation

The idea of a computer virus can be traced back to the age of the first computer itself, as described in 1966 in the seminal work of John von Neumann [153]. Two decades later, Cohen proved the undecidability of computer viruses and, hence, the impossibility of completely secure computer systems [34]. The first virus was just an experimental code that displayed a message and moved between mainframe computers [29]. However, the motivations of virus authors soon shifted from curiosity-driven academic experiments to malevolent intentions [61, 43, 75].

Nowadays, viruses are just one category of the wide range of malicious software (malware). Not only the malware intentions shift towards nefarious operations but the sophistication of their attacks is increasing. So does the cost of defending against them [137, 28].

According to the recent cybercrime reports, malware is a prevalent issue [45, 124, 134, 108]. The monetary loss globally is reported as high as 600 billion USA dollars in 2017 and projected to reach 6 trillion by 2021 [79, 96, 64]. Specifically highlighted is *ransomware*, i.e., malicious software asking for ransom to release captured files. More than 4,000 attacks by ransomware are reported every day [152]. The most severe example, “WannaCry”, infected more than 300,000 computer systems across 150 countries in just a few days [143, 145]. Such observations led to a need for detection of information technology bypasses on multiple levels, such as the network level, the software level but also as deep as a device’s firmware level [26].

The problem posed by advanced malware intensifies when considering intrusion surfaces beyond the established ones. A common entry point into a network is breaking through its border router connecting with the Internet [154]. An attacker can deliver the malicious payload by creating advanced pieces of malware that evade modern system mitigations, including antivirus software, operating-system-level defenses (e.g., Address Space Layout

Randomization (ASLR) [159] and Data Execution Prevention (DEP) [100]), and binary-level defenses (e.g., Control Flow Integrity (CFI) [1]).

Universal Serial Bus (USB) devices are an even greater threat, especially in the era of bring-your-own-device (BYOD) in enterprise environments. Such devices are blindly trusted and plugged-in by their users wherever possible. Hence, they become the perfect intermediary for payload delivery [149]. Modern USB-based attacks are not bound to delivering malware via their storage media. The “BadUSB” attack modifies (reflashes) the firmware of a USB device to attack without having stored anything on the storage media [119]. The attack is carried out by the malicious firmware that acts as a USB keyboard and automatically injects keystrokes into the victim’s system. This way, antivirus protection and other common defenses are not able to cope with BadUSB [86].

“Stuxnet”, the most sophisticated worm to date, demonstrated the ineffectiveness of “airgapped systems” as a security defense [104, 82]. This malware was delivered into Iranian uranium enrichment centrifuges by contractors, closing the airgap to highly-secured and critical systems by using reflashed USB devices [84]. Other cases where the airgaps were closed by trusted parties include the incidents of IBM and the International Space Station (ISS). In the former case, IBM shipped USB devices for configuring its enterprise storage solutions; these otherwise trusted devices were shown to deliver malware to IBM customers [30]. In the latter case, ISS systems were infected by a Russian cosmonaut carrying an unchecked USB device containing malware from our planet all the way up to the ISS in orbit [57].

USB devices are not only used for active exploitation by leading technology companies, spaceship crew, and state actors. In some cases, these actors are targeted themselves. In mid 2017, a hacker group called “The Shadow Brokers” revealed exploits and other software which they have stolen during their (to date) most successful raid: Classified documents, highly sophisticated file-less malware and exploit frameworks of the National Security Agency of the USA (NSA) [138, 37, 58]. Seven of those stolen assets have been chained together and utilized by the ransomware “EternalRocks” only a few weeks after their release [117, 99]. Agency internal teams, similar to computer security incident response teams (CSIRTs) traced the attack back to an NSA insider [141]. Considering this incident and modern steganographic techniques, USB based attacks become an even more prevalent threat. In this case, data exfiltration from highly secured environments, including agencies themselves [115, 90, 156, 55].

A USB port is truly universal in consumer-grade digital products used in smart homes and the devices that form the Internet of Things (IoT). This port is used for charging, for updating their software, and for interfacing their peripherals. All these consumer IoT devices exhibit fierce competition, decreasing times-to-market, and lack a regulatory framework for their operation, maintenance (including security updates), and overall lifecycle management. Such constrained devices are exposed to maliciously-reflashed USB devices, forming an entry point to our private home and beyond. In this modus operandi, we must avoid transforming the USB port to the “Universal Security Bypass” in the enterprise and the consumer IoT worlds.

It is of utmost importance to enhance the cybersecurity incident response and cybercrime defense capabilities. In this context, the main research question we address in this thesis is: How can we be more efficient and effective in detecting and responding to malicious actions and events?

1.2 Aims

In this thesis, we design, engineer, and evaluate operating system-level defenses against USB-based attacks. We follow a holistic approach to address the information security engineering challenge, taking into account the timeliness, cost, and risk dimensions to prioritize the focus areas, as depicted in Figure 1.1.

The first aim of this thesis is to devise solutions that massively reduce the post-incident file corpus to be analyzed and therefore improve the analysis performance. Cybercriminals

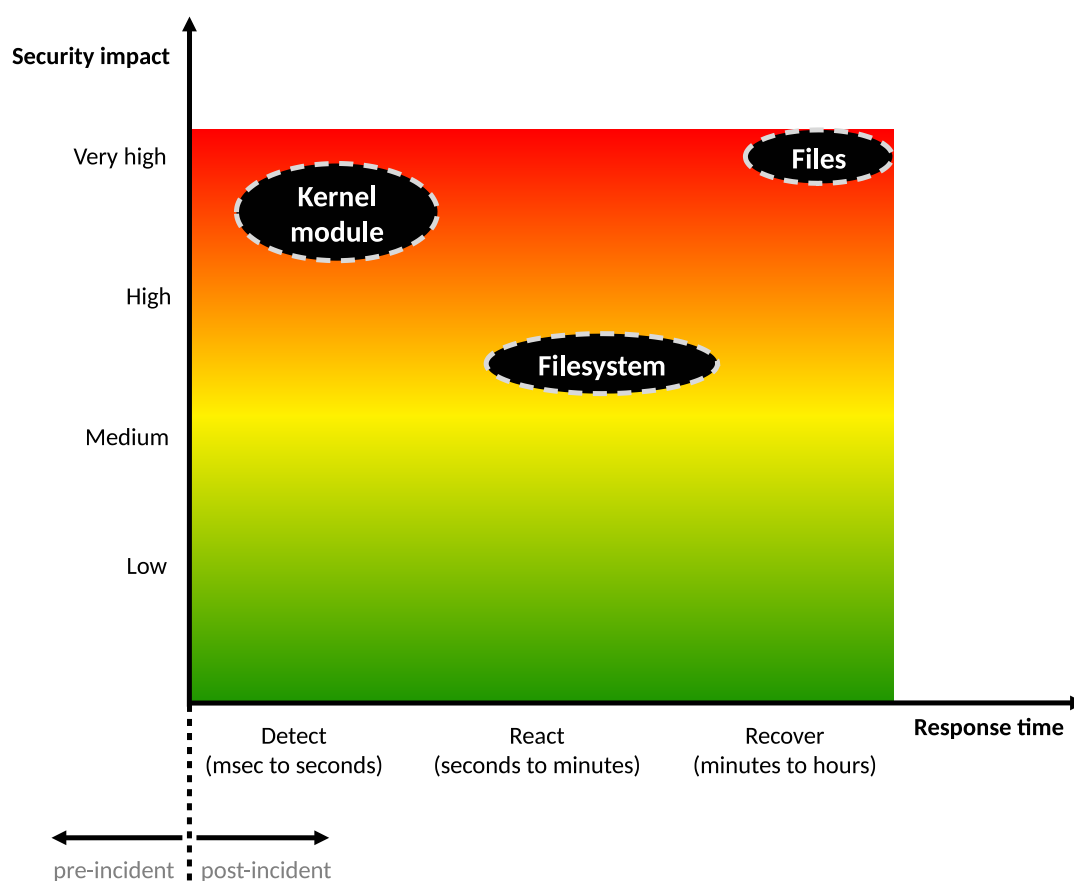


Figure 1.1: Response phase and timeliness vs. estimated security impact; thesis focus points.

use computer files for exploitation, to store malware, and to leak sensitive data via USB storage media files. Commodity hardware with double-digit-terabyte capacities and big data cloud storage systems are easily available with an appropriate budget at hand. This poses a significant challenge to computer security incident response teams and investigators. Proper incident handling requires backup and working copies for verification reasons, which intensifies the problem of analysing the exponentially increasing amounts of data for suspicious files. This volume of data actively hinders the analysis efforts and results in the need of smart processing techniques to exclude unneeded files from further processing.

The second aim of this thesis is to analyze filesystem metadata as a means to create steganographic channel. If this is feasible in first place, we aim to devise smart detection techniques to reveal such malicious manipulations of filesystem metadata. Cybercrimes do not necessarily involve file contents only. Hence, a file-contents-based investigation might not bring insightful results. File metadata is a potential but often neglected carrier to hide information.

The third aim of this thesis is to devise kernel-level defenses against USB reflashing attacks so as to detect and prevent keystroke injection attacks. Also, to aid the CSIRTs efforts during post-incident analysis of a BadUSB-like attack in an enterprise environment, where the amount of possibly infected workstations is unmanageable with established procedures. In the age of bring-your-own-device (BYOD) practice, CSIRTs are not only faced with malware stored on stationary hard disks. More often than not, they must cope with malware on unmanaged, non-stationary devices, especially USB storage devices. In fact, USB devices are a trusted entry point into the enterprise network beyond the reach of managed security solutions. Attacks based on reflashing the firmware of USB devices in combination with the plug-and-play capability of USB are more prevalent, e.g., the BadUSB attack. These attacks leave no file trace and minimal logs at the operating system level. Hence, early detection and indicators of compromise are necessary.

1.3 Methodology

Throughout the thesis, we follow a multi-step procedure to appropriately address the main research question.

As a first step, we perform a **literature review**. We review the published literature and state of the art.

The second step is the **defense design**. We devise a holistic security solution that appropriately defends against the identified security threats and matches the engineering constraints of the application domain, as for example the available time-to-analyze for a CSIRT.

In the third step, we **collect and generate datasets** for empirical evaluation and validation of our designs. This step involves large-scale operations, including Internet-wide scanning, longitudinal studies, and laboratory studies with human participants.

The next two steps are the **prototype design**, where a proof-of-concept for the devised defense is designed to perform validation and evaluation experiments, and the actual **prototype implementation**.

The final step is the **security evaluation**, where the devised solution is evaluated under different conditions in regards to correct operation, scalability, and performance.

1.4 Main Results

We proposed a novel smart whitelisting technique to exclude known files from further processing of post-incident corpora during an incident response analysis. Our proposal involves the utilization of available databases, such as the NIST National Software Reference Library and files, which are cross-correlated from multiple sources within the investigation corpus, to filter known files. Our method improves the investigation process, not only in terms of reduced processing time but also in terms of data volume to process. The theoretical approach was further evaluated using a prototype implementation and real-world corpus compiled in the course of this research. Our findings were published in [111, 112].

A second proposal towards this direction was to enrich the incident response analysis with the *sub-file* hashes contained in data fragments that are publicly-available on Torrents of globally-spanning peer-to-peer (P2P) networks. We built and released databases containing over 6 *billion* sub-file hashes from more than 4.8 million torrents. Utilizing those datasets, the identification of more than 5 *petabytes* of data (64 million files) within a forensic corpus is possible. Our findings were published in [113].

We analyzed how modern operating systems represent and store file timestamps in filesystem metadata. We identified that the sub-second part of these timestamps contains unused information that can be manipulated without affecting normal system operations. Hence, one can hide information in them and evade information leakage detection. We described how an information channel of steganographic strength can be built using USB disks, utilizing encoding, error correction, and encryption. We devised a proof-of-concept implementation of this system. We also performed a real-world study in different environments, analyzing the channel capacity and proposing appropriate detection techniques. Our findings were published in [115] and [116].

We proposed a new technique to fast triage computer systems in an enterprise environment after a security incident involving USB devices. An information collection, analysis, and visualization aid for authorized personnel was also developed to aid the analysis. Our findings were published in [114].

We proposed USBlock, a highly-efficient and effective kernel-level defense against malicious USB devices. USBlock can detect fake keyboard devices in only a few milliseconds and by analyzing only a few (fake) keystrokes. This allows to detect and protect against such attacks *automatically*, at the system level, without requiring any user participation in the security decision. This constitutes a unique characteristic of USBlock compared to

published literature. An extensive user study validated the soundness of our proposal. Our findings were published in [114].

The results of the research comprising this Ph.D. thesis were published in the proceedings of international peer-reviewed journals and conferences:

1. **S. Neuner**, A. G. Voyiatzis, S. Fotopoulos, C. Mulliner, and E. R. Weippl. US-Block: Blocking USB-based Keypress Injection Attacks. In *32nd Annual Conference on Data and Applications Security and Privacy* (DBSec 2018), IFIP WG 11.3, 2018.
2. **S. Neuner**, A. G. Voyiatzis, M. Schmiedecker, and E. R. Weippl. Timestamp hiccups: detecting manipulated filesystem timestamps. In *12th International Conference on Availability, Reliability and Security* (ARES 2017), ACM ICPS, pages 33:1–33:6, 2017.
3. **S. Neuner**, A. G. Voyiatzis, M. Schmiedecker, S. Brunthaler, S. Katzenbeisser, and E. R. Weippl. Time is on my side: Steganography in filesystem metadata. *Digital Investigation*, 18(7):76–86, 2016.
4. **S. Neuner**, M. Schmiedecker, and E. R. Weippl. PeekTorrent: Leveraging P2P Hash Values for Digital Forensics. *Digital Investigation*, 18(7):149–156, 2016.
5. **S. Neuner**, M. Schmiedecker, and E. R. Weippl. Effectiveness of file-based deduplication in digital forensics. *Security and Communication Networks*, 15(9):2876–2885, 2016.
6. **S. Neuner**, M. Mulazzani, S. Schrittwieser, and E. R. Weippl. Gradually improving the forensic process. In *10th International Conference on Availability, Reliability and Security* (ARES 2015), pages 404–410. IEEE, 2015.

The following contributions were also made in the course of my studies:

7. P. Kieseberg, **S. Neuner**, S. Schrittwieser, M. Schmiedecker, and E. R. Weippl. Real-time Forensics through Endpoint Visibility. In *International Conference on Digital Forensics & Cyber Crime*, 2017.
8. G. Merzdovnik, M. Huber, D. Buhov, N. Nikiiforakis, **S. Neuner**, M. Schmiedecker, and E. R. Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *IEEE European Symposium on Security and Privacy* (EuroS&P 2017), 2017.
9. M. Schmiedecker and **S. Neuner**. On Reducing Bottlenecks in Digital Forensics. *ERCIM News*, 106:54–54, July 2016.
10. **S. Neuner**, V. van der Veen, M. Lindorfer, M. Huber, G. Merzdovnik, M. Mulazzani, and E. R. Weippl. Enter Sandbox: Android Sandbox Comparison. In *Mobile Security Technologies*, 2014.

11. C. Kadluba, M. Mulazzani, L. Zechner, **S. Neuner**, and E. R. Weippl. Windows installer security. In *ASE International Conference on Privacy, Security, Risk and Trust*, 2014.
12. M. Mulazzani, **S. Neuner**, P. Kieseberg, M. Huber, S. Schrittwieser, and E. R. Weippl. Quantifying windows file slack size and stability. In *IFIP International Conference on Digital Forensics*, 2013.

Whenever possible, we openly released collected and processed datasets and prototype implementations, following the best practices in open science and ensuring reproducibility of our research. The following list points the interested reader to the repositories:

- A. *PeekaTorrent: Leveraging P2P Hash Values for Digital Forensics*:
<https://peekatorrent.org/>
- B. *Time is on my side: Steganography in filesystem metadata*:
<https://www.sba-research.org/dfrws2016/>
- C. *Timestamp hiccups: detecting manipulated filesystem timestamps*:
<https://www.sba-research.org/ares2017hiccups/>

1.5 Structure

The remainder of this thesis is structured as follows. We note that for the sake of coherence and easy reference, the literature review for each addressed topic is incorporated in the respective chapter.

Chapter 2 presents our novel smart whitelisting technique to exclude known files from further processing during an incident response analysis.

Chapter 3 presents the use of Torrents from globally spanning peer-to-peer-networks as a means to enhance the performance of (sub-) file analysis, during an incident response analysis.

Chapter 4 presents the novel steganographic channel to hide information in filesystem metadata (sub-second file timestamps), its strength against normal filesystem use in different environments, and detection techniques that reduce its information capacity.

Chapter 5 presents our novel kernel-level defense against keystroke-injection attacks from fake USB devices. Also, a method is presented to fast-triage computer systems in enterprise environments after a security incident.

Finally, Chapter 6 presents the conclusions of this thesis and discusses future directions of research.

CHAPTER 2

Enhanced File Whitelisting for Incident Response

Investigators in digital forensics are facing several issues throughout their daily analysis routines. One of the biggest issues is the increasing amount of commodity hardware affordable for everybody. Commodity 3.5" SATA hard drives come with a maximum capacity of up to 8 terabytes per hard drive, while memory cards for smartphones and digital cameras can have up to 256 gigabytes. USB thumb drives have a current maximum capacity of two terabytes. Observing the past trend, Kryder et al. [81] expect this trend to increase even further than posited by Moore's Law [20]. Considering the stated numbers and the average number of devices per household, a forensic investigator would have to deal with tens of terabytes of data. This amount of data has to be securely acquired, processed and stored in several copies. Garfinkel et al. [52] predicted this problem years ago. However, so far this issue was not tackled in-depth in research and neither was it adapted into the forensic process of investigators nor in the corresponding standards.

This chapter not only aims to enhance well-known standards like the RFC 3227 [19] or the NIST SP-800-86 [74] with the recommendations proposed in this chapter, but also aims to proof the proposed performance improvements by reducing the to-be-acquired data in an early stage of the investigation. Those recommendations specifically tackle storage capacity issues during an investigation and therefore also reduce the needed processing power, workload and time to handle the data.

Therefore the contributions of this chapter are as follows:

- We propose an advanced forensic process for digital investigations, taking into account some of the most pressing limitations for investigators.

- We discuss different analysis techniques which scale well and can be used to limit backend storage requirements for analysts.
- We evaluate our process with an exemplary use case and show that the overall storage requirement for that case can be decreased by 78%.
- Conducting an evaluation on a real-world dataset, we show that storage reductions of over a third are possible.

2.1 Background

The usual image acquisition approach in digital forensics as defined in [74, 19] creates two images of a hard drive of interest, while hashing the source and destination multiple times to prove that the integrity of the source hard drive has not been tainted and the underlying data has not been modified. As such, the created images are exactly the same size as the hard drive, which is one of the yet unsolved problems in digital forensics [52]: hard drives with 8 terabytes capacity are nowadays a commodity, and can be readily obtained online and in retail stores.

A major problem with the current forensic process as practiced today is its duration. It can take days to acquire and analyze large data repositories. Even though highly parallelized approaches have been proposed recently [53], they are not yet incorporated in commercial tools or the process itself, where open source products are heading in the right direction as shown by the well-known tool *bulk_extractor*¹. Moreover, the large set of software, data formats and as devices restrict the possibility of having fully automated approaches [25]. As such, the images created are still bit-by-bit exact copies of the hard drive to be analyzed.

File whitelisting is a common approach in digital forensics, during which each file on the hard drive is hashed using e.g. SHA-1 and compared against a list of well-known, benign files. The largest corpus of benign files is the NIST National Software Reference Library (NSRL) with their Reference Data Set (RDS). To build it, NIST installs all kinds of software in virtual machines and monitors the files that are created during installation. This allows the RDS to map each stored hash value to a specific file and furthermore to which software package containing it. The RDS is published quarterly, the most recent version at the time of writing being the RDS 2.49 from June 2015, containing more than 42 million unique hash values.

¹Online at https://github.com/simsong/bulk_extractor

2.2 Improvements to the Forensic Process

This section is split into two parts: In the first part we explain theoretical techniques which can be leveraged to cope with the ever increasing digital forensics investigation case sizes. In the second part, we evaluate an artificial corpus with respect to existing forensic analysis recommendations.

2.2.1 Individual Improvements

The first proposed enhancement of the forensic process is already done in practice: **two physical copies are not always required**. While a working copy and at least one backup copy should always be in place, less stringent rule enforcement is needed when the data source drives are not bound to time requirements to get back to their owners or into production. This is particularly the case for investigations by law enforcement, where the data sources themselves are confiscated and no temporal pressure exists to return them to the owner. We do not have concrete numbers on how this is done in practice, but this can be very effective for reducing storage requirements. It is more like a logical enhancement to the standard processes, since it is not in all cases that the data needs to go back to production systems as soon as possible. Of course, for production systems where downtime is an issue and hard constraints exist that these systems stay online, a second copy is needed for backup. This is of relevance for example all kinds of server systems like e-mail or web servers. Sometimes it can be also sufficient to create an image of the current files in the file system, omitting the free space and possible file slack. This depends on the context of the investigation, and the actual questions to be answered.

Another strategy which is missing so far in the process descriptions is the rigorous use of **file whitelisting**. Files irrelevant to the investigation can be easily excluded in the early stages due to the use of cryptographic hash functions like MD5 or SHA-1, whereas files of particular interest can be identified if they are known a-priori to the investigator. In the forensic community, the most notable example for the former case is the NIST national software reference library (NSRL) with their reference data set (RDS) [98]. It uses default software installations of operating systems and end user software to derive a list of hash values on a file basis. The most recent version of the RDS 2.49 (as of June 2015) contains more than 42 million unique hash values. An example for the latter is PhotoDNA which computes a visual fingerprint for pictures and compares it with known pictures of child abuse. It was developed by Microsoft and Dartmouth University and is used by large software companies like Facebook or Twitter. Most recently, a REST API was introduced to query the PhotoDNA database online². Due to the availability of cheap storage and processing power, we argue that any investigator could and should set up their own list of hash values for files of interest. This could include all files from intra-company file shares, possibly malicious files from anti-virus quarantine, web pages (including pictures and thumbnails) or company-wide e-mail attachments. De-

²Online at <http://www.microsoft.com/en-us/photodna>

pending on the local privacy laws there are hardly any limitations on which files to include.

The improvement on the storage backend which this chapter proposes is the creation of a **reduced working copy**. It is created as soon as all known, benign files are identified, as they can be safely excluded from the need to store them (except for their metadata). All other files are stored according to the file system metadata, and additionally all portions of the free space are extracted and stored as well. At worst, this can be a very large fraction of the original drive capacity. At best, a vast majority of files can be excluded in a fully automated process and without any interaction of the investigator. Since this process is strictly monotonous (the resulting working copy can only be at most the capacity of the drive), the resulting working copy will always be smaller than the full capacity of the storage drive. All further analysis steps can be done on this reduced working copy, and the original drive(s) can be locked securely away as the backup. If the drive(s) need(s) to go back into production use, a second copy is to be created using a bitwise copy. The second large improvement on the storage backend is the rigorous use of deduplication, at the very least across devices within each case. This step should also include the application of **fuzzy hashing** [130], since files which are similar but not the same until the very last bit cannot be identified using cryptographic hash functions. While the most commonly found fuzzy hash functions are *ssdeep* [80] and *sdhash* [129], there is still no common ground which is the best for specific use cases, and specialized similarity hash functions are still an active field of research [18], for example *mrsh-v2* [16] which can identify file fragments.

Hashing each file per device by default can be used to easily **identify the same files across devices** and reduces the need for storing them multiple times. This results in storage savings at the investigators backend due to deduplication. In particular with the use of cloud storage solutions like Dropbox or iCloud, many devices nowadays share local files which are kept synchronous across devices. However, the file system metadata of all copies needs to be preserved. Across cases and in the near future, efficient and privacy-preserving mechanisms will be needed to share hash value lists between multiple parties. Even though there are current mechanisms available to facilitate private set intersection [39], i.e. using zero-knowledge proofs [22], it is not yet known if they can be used for digital forensics and handle millions of hash values in practice. File system- as well as enhanced analysis of file **metadata** should be used in this step to compare file timestamps, EXIF metadata or other information sources in order to identify data sources and sinks and to reconstruct the flow of information across devices (and users). In very large environments with thousands of computers and users, this can be challenging.

Finally, the process should include the acquisition from various **online accounts** and the retrieval of the associated data and metadata using forensic methods. Online services like Facebook, Twitter, Apple or Google Services have hundreds of millions of users, and these online accounts are often tied to smartphones. While these companies have

mechanisms in place to aid law enforcement, this source of information is not available to foreign civil law suits or other third-party investigations. Even though approaches have been recently proposed to acquire the data without the explicit aid of service operators, e.g. using APIs [68] or based on observed network traffic [35] [109], they have not yet been incorporated in the standard processes. Cloud computing [11] can pose another, although related type of problem for digital forensics. Compared to online services and SaaS platforms, acquisition in IaaS cloud services is more related to the standard forensic process in direct comparison [92]. An obstacle is often how the investigator can access these services and whether or not the credentials needed for authentication can be obtained from the suspects, the hard drives or by other means. In most cases user consent is needed, and even though Great Britain is among the few countries that can convict a suspect if he/she is not releasing a password, this is not commonly found elsewhere.

2.2.2 Improved Forensics Process

Our improved steps for automated data analysis so far only enhance the current standards, in particular NIST SP-800 86. While RFC 3227 stops after the data acquisition, NIST SP-800 86 states specific steps to reduce the amount of files and data to analyze, i.e. using the NIST NSRL hash value collection. However, fuzzy hashing and cross-device checking are not mentioned, as well as the importance of online accounts for data storage and online services. It only exemplifies the use of multiple sources for data gathering, within a confined scope.

The core improvement in this chapter is the parallelized calculation and evaluation of hash values, and the reduced working copy. As before, the data should be acquired according to the order of volatility, and using a hardware write blocker to prevent manipulations (accompanied by rigorous documentation). Before the image is created, file system metadata is parsed and all files in the file system hashed numerous times, including cryptographic hash functions like SHA-1 and fuzzy hash functions like *ssdeep* or *sdhash*. These hash values are then stored in some form of database and automatically evaluated with the proposed improvements: known, benign files are excluded using e.g. the NIST NSRL dataset, and multiple copies of the same file are detected across devices. Similarity hash values are used to detect similar files and present a set of candidates that seem related. This information can be embedded and enriched within an automatic timeline creation from file system metadata in the acquisition steps. Deleted files where the data has not yet been overwritten should be extracted and hashed similar to the other files. Furthermore, known malicious files can be found using hash value black listing. In the future, additional hash value calculations can be added as well as additional hash value sources. This can include novel fuzzy hash functions, other cryptographic hash functions like the upcoming SHA-3 or new hashing methodologies like sector hashing as proposed in [161].

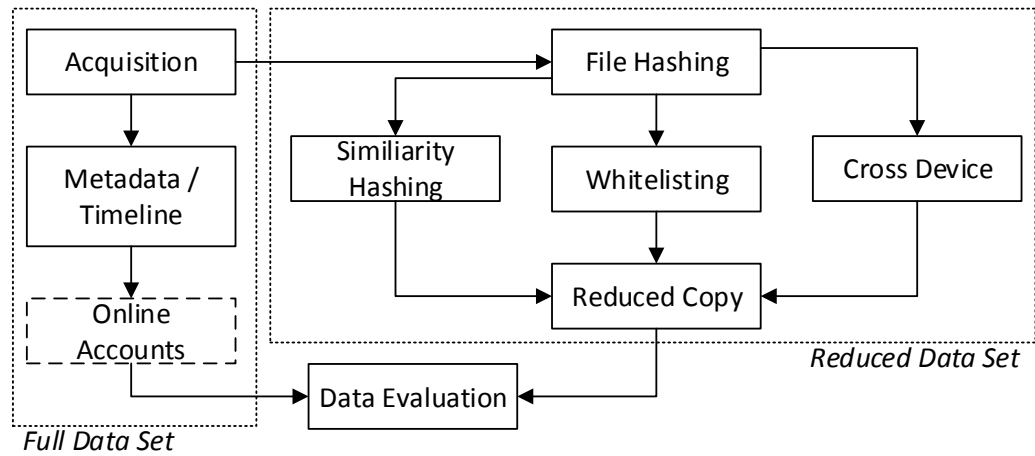


Figure 2.1: Improved Steps for the Forensic Process

After the automatic exclusion of files, the remaining files, folders and regions of free space are copied into the reduced working copy. Depending on the context of the analysis, this is expected to be sufficient for many cases. The use of cryptographic hash functions allows the argumentative exclusion of known files, since for each and every file there is a line of argumentation why this file was removed and ignored in further analysis steps. The final step is the optional extraction of online credentials from browsers, stored passwords or artefacts from online data services like e.g. Dropbox or iCloud. The entire process is visualized in Figure 2.1. Please note that the individual processing steps can be run concurrently: hashing the files may happen on the same byte stream as extracting the file system metadata, thus reducing the amount of read requests to the hard drive to the original bitwise copy as used today in digital forensics. Also, the extraction of online account information is considered optional, thus the different representation in the figure.

Most importantly, all of the steps discussed so far have the ability to run automatically and present their findings in an understandable format to the investigator as well as in machine-readable form for further analysis steps. The computational overhead is very likely to be negligible compared to the additional insights using automated analysis as well as the possible reduction in the number of files and file fragments needing manual inspection.

2.3 Evaluation

In this section we first describe the theoretical approach as described by Neuner et al. [111]. This first part describes an artificial scenario. However, the specifics we used were derived during our ongoing informal discussions with law enforcement officials as well as forensic investigators. Building upon this theory and conducting an evaluation to proof the statement on a real-world corpus, the second part of this section shows the practicability of the proposed deduplication approach as well as storage and performance improvements on 16 disks.

2.3.1 Theoretical Approach for Deduplication

On the theoretical basis we consider some form of malicious online activity as the initial reason for an investigation. The investigator is tasked with the acquisition of a relatively small number of devices from the following set, all devices which can be found in a modern household: computers or notebooks, smartphones respectively tablets, external storage devices like USB thumb drives or external hard drives, and lastly digital cameras. Furthermore numerous accounts at online services, e.g. Facebook, Google, Flickr or Twitter (just to name a few) which are out of scope for our evaluation.

For the evaluation of our theoretical approach we used the following setup. We consider the investigated person to have the following devices in use: Two computers, whereas one computer is a Desktop PC and one computer is a Laptop. The Windows PC is based on Windows 8 which uses roughly 160,000 files. We consider an additional total of 50,000 files to be from the user, including temporary working files and installed software. As described by Rowe et al. [133], commonly found hard drives include 18% Microsoft-related system files, 25% graphics (e.g. camera images), 4.7% documents (e.g. spreadsheets, presentations, etc.) and 4.3% executables to name the most important types. For mobility reasons the user has a Laptop computer with files daily mirrored with the Desktop computer and therefore these corpus' share 80% of the same files. He/she uses an Android smartphone with about 13,000 multimedia files such as images, photos, videos and music files as described by Lessard et al. [87], 2,000 files which are related to different installed Apps (assuming about 300 files per App) and 20,000 files which are either related to running Google services or related to the Android operating system itself.

In addition to the mentioned computation devices (computers, smartphone) the specified setup includes two digital cameras with 2000 photos in total, split across three SD cards and several external storage devices used for backup. Those external storage devices include two external hard drives with half a terabyte and one terabyte in capacity, and three USB thumb drives from various manufacturers and with different capacities. These external hard drives contain the backed-up files from the Desktop PC as well as the notebook, respectively. The Desktop PC was used for backing up the files from the cameras and the smartphone, meaning that these files are found in the backup on the

one terabyte hard drive as well. The USB thumb drives include an additional 20,000 files which are unique with respect to the other devices. Finally data is spread over the computers and the smartphone via a cloud service (e.g. Dropbox) and kept in sync with a remote copy. Therefore a large number of the user files are available on Desktop PC, Laptop as well as the smartphone. Please note that most of the specific numbers were chosen at random, they would differ in reality due to the different user's age distribution, usage patterns, personal preferences, professional background and other factors. The overall capacity and number files for each device can be seen in Table 2.1.

Table 2.1: Overview of devices and storage capacities

Device	Storage Capacity	# of files	used capacity
Windows 8 PC	1 TB	210k	250 GB
Windows 8 Notebook	500 GB	190k	180 GB
Android Smartphone	32 GB	35k	15 GB
SD Cards	{8 16} GB	2k	10 GB
external hard drive	{500GB 1TB}	400k	430 GB
USB thumb drive	{4 16} GB	20k	32 GB
Sum:	3.16 TB	857k	917 GB

2.3.2 Evaluation of the Theoretical Approach





The regular forensic process would need to acquire and copy each device at least once, resulting in the need to store roughly a little more than three terabytes of data only for the device images. If a backup copy is needed, this adds up to 6.2 terabytes of storage capacity needed. Overall, this would also mean extracting and analyzing 857,000 files. In the improved forensic process, however, an overall list for the entire case and thus all the devices is created which contains file names and hash values of all the unique files. This list also includes all metadata for the deduplicated files. To further reduce the number of files to be extracted for the working copy, the content is compared to available software reference lists like the NSRL. These steps allow to drastically reduce the numbers. The first reduction is caused by having redundant device contents on multiple devices. The Desktop computer and the Laptop both have their backup exclusively on their external USB backup drives. 80% of the Laptop user files (30,000) are duplicates from the Desktop PC, leaving 20% or 6,000 unique files as difference between the user files on the Desktop PC and the Laptop (due to cloud sync and working copies). As such, and starting with the acquisition on the Desktop PC, 210,000 files are to be extracted from the Desktop PC while the acquisition of the Laptop deduplicates the operating files and most of the user files. Therefore 184,000 files are duplicates and not added to the reduced working copy.

One cloud service is in use which synchronizes files over the Desktop PC, the Laptop computer as well as the smartphone, including the pictures of the user. A typical Desktop PC contains about 7.6% camera images as of [133], which would be in this particular case

roughly 16,000 pictures on the PC. This is a superset of the pictures from the smartphone, including the audio and video files outside of the synced folders, leaving 22,000 files on the smartphone to be included in the reduced working copy. The 2,000 pictures on the digital cameras (stored on three different SD cards) were already synced to the Desktop and are thus duplicates in this case.

The last step is the removal of commonly found files e.g., by using the NIST NSRL RDS. According to Rowe [132], 32% of a typical hard drive can be matched with files contained in the RDS set. This reduces the 210,000 files from the Desktop PC to roughly 143,000 files, and the files uniquely found on the Laptop to 4,080 files. Table 2.2 illustrates files to be extracted per source in the corpus. Grayscale areas mark the proportion of files that have to be extracted from that specific source, whereas white areas are duplicates that do not have to be taken into account for the created reduced working copy.

Table 2.2: File extraction distribution per source in corpus.

Desktop PC		68%
Laptop		2%
ext. USB devices		5%
SD card / camera		
Cloud		
Smartphone		63%

2.3.3 Real-World Evaluation Corpus

The evaluation was carried out on a real-world dataset from 16 participants captured in an IT consulting company and research institution respectively. This company has a managed network, which is used to install the latest, stable Windows operating system on all clients as well as the corresponding updates. To be more precise, Windows 8 or Windows 8.1 was installed on all captured disks respectively. To capture the data stored on the disks the tool *tsk_loaddb*³ was used which is included in the well-known *sleuthkit* (TSK) [24] forensic investigation software. This tool stores important information about every file and directory on-disk to a sqlite database, above all a cryptographic checksum (MD5) and the size. Two important modifications were made to the corpus: To reduce the size of the corpus and store only the data needed for evaluation (checksums and sizes) all tables except the table *tsk_files* were dropped. Furthermore, database rows containing temporary Windows files were deleted. This deletion process includes files such as *\$BadClus:\$Bad* which is a sparse file, including a named stream created by NTFS [135]. Additionally, the columns *name* and *parent_path* were filled with *NULL* to

³Online at: http://wiki.sleuthkit.org/index.php?title=Tsk_loaddb

preserve the privacy of the participants. Considering those 16 captured disks and only the remaining table, the corpus contains roughly ten million database rows in total.

2.3.4 Real-World Evaluation Design

Considering those millions of files the evaluation setup is as follows: The platform the evaluation was performed on is an Ubuntu Server with 16 physical cores and 72 gigabytes of memory. The developed evaluation application is written in Python. It reads all databases stored by *tsk_loaddb* and stores them in Python lists or dictionaries respectively. Keeping all of this information in memory, the comparing steps are faster and not I/O bound. Otherwise, the limiting factor would be reading the databases from disk.

The following describes the different evaluations carried out on the dataset to show the effectiveness of the proposed approach:

NSRL Reduction: The NSRL Reference Data Set (RDS)⁴ in its *minimal* version stores over 42 million unique hashes for identification of known files. Making use of this dataset, the first part of the evaluation covers the comparison of our corpus to the NSRL RDS.

Cross Comparison: This evaluation step covers the comparison of one corpus database to the remaining databases.

Incremental Reduction: To show the distribution of reductions over every database, this part of the evaluation shows the incremental reduction of one database to the remaining databases. In more detail, let D be the databases from one to n , n be the total number of databases and x the database to be compared, then:

$$D = \{1 \dots n\}$$

$$\Pi_i = \{x | x \in D \setminus D_i\}$$

$$f_{(i,j)} = D_i \Delta \Pi_{i,j}$$

Figure 2.5 shows this as an average distribution over all databases in the corpus.

2.4 Results

Considering the evaluation in Section 2.3 being split into two parts, this section is split in the same manner: The theoretical approach as well as the evaluation on the real-world corpus. The first part describes the results of the theoretical approach on the artificial dataset; in the second part we show that deduplication at the time of acquisition significantly reduces the number of to-be-saved as well as the required storage.

⁴Online at <http://www.nsrl.nist.gov/Downloads.htm>

2.4.1 Results of the Theoretical Approach

As described in the previous section, the working copy of the artificial dataset will finally contain 189,000 files. This means a total reduction of 78% compared to the full dataset. Furthermore considering the average size of 1.1 megabytes per file, the working copy is reduced by 709 gigabytes. Since a reduction of size also means a reduction in computing power (e.g. hashing of files) an investigator experiences an overall performance enhancement. The percentages of the overall reduction in files and storage space on the artificial dataset are represented in Figure 2.2.

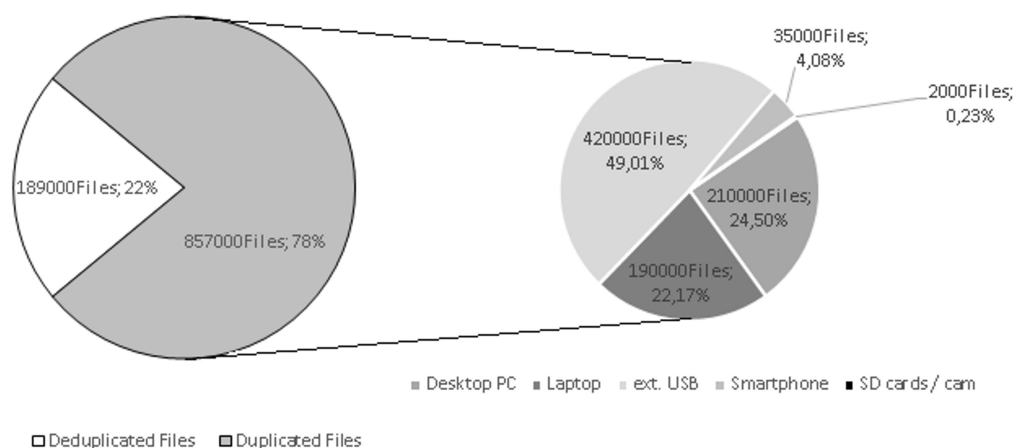


Figure 2.2: File reduction in the reduced working copy

2.4.2 Results of the Real-World Corpus

To state the improvement of the proposed approach, Table 2.3 shows the number of hashes (and therefore number of hashable files and directories) as well as the corresponding sizes for each database in the corpus.

In regards to the evaluation design the results of the evaluation are splitted into three parts.

NSRL Reduction: Figure 2.3 shows the deduplication rate when comparing each database of the corpus to the NSRL RDS dataset. The NSRL RDS dataset used in this evaluation contains 42,060,541 unique MD5 hashes (RDS minimal version) not including any size information.

As indicated by Figure 2.3 every database in the corpus can be reduced in terms of hashes as well as size when compared to the NSRL RDS dataset. The black bars show the duplication in percent for the hashes, whereas the gray bars show the

Table 2.3: Corpus Details

Database	Total # of Hashes	Total Size [GB]
1.sqlite	199255	243.63
2.sqlite	244499	238.82
3.sqlite	222561	304.11
4.sqlite	224166	238.03
5.sqlite	503764	242.16
6.sqlite	382571	122.49
7.sqlite	309853	90.90
8.sqlite	618218	171.16
9.sqlite	497088	249.42
10.sqlite	713162	147.07
11.sqlite	203187	244.03
12.sqlite	226521	244.01
13.sqlite	708268	302.76
14.sqlite	304996	470.73
15.sqlite	623331	244.63
16.sqlite	334121	83.13

duplication in percent for the size. Database 1 is the rare case of having the same relative amount of duplication in hashes and size at 4.59% (meaning a possible deduplication of 14,210 files/directories and 4.18 GB). In contrast 31.60% of the hashes within database 8 can also be found in the NSRL dataset, which corresponds to 2.42% of its (database 8) size (5.45 GB).

Cross Comparison: When e.g. acquiring large amounts of data a high number of semi-identical disks, duplicates exist. Figure 2.4 proves this assumption.

It shows reductions in hashes and size for every database compared to the sum of all remaining databases. Two strong outliers are database 5 and 16 with 66.81% (336,570 hashes) and 66.28% (221,465 hashes) possible hash deduplication respectively. Those numbers correspond to reductions of 23% and 35.16% in size, meaning absolute reductions of 55.59 GB and 29.23 GB respectively.

Incremental Reduction: To show the distribution of duplication throughout the corpus related to every database, Figure 2.5 illustrates an average over the incremental comparison as described in Section 2.3. The gray line corresponds to the average duplication in percent of the hashes, whereas the black line corresponds to the average duplication in percent in regards to the size of the databases. The x-axis describes the compared database to the initial database. To ensure the comparison of always the same sequence of databases, the Python built-in function *sort()* was used to pre-process the sequence (e.g. comparing the numbers [1,2,3,10], the sorted list according to Python *sort()* would be [1,10,2,3]).

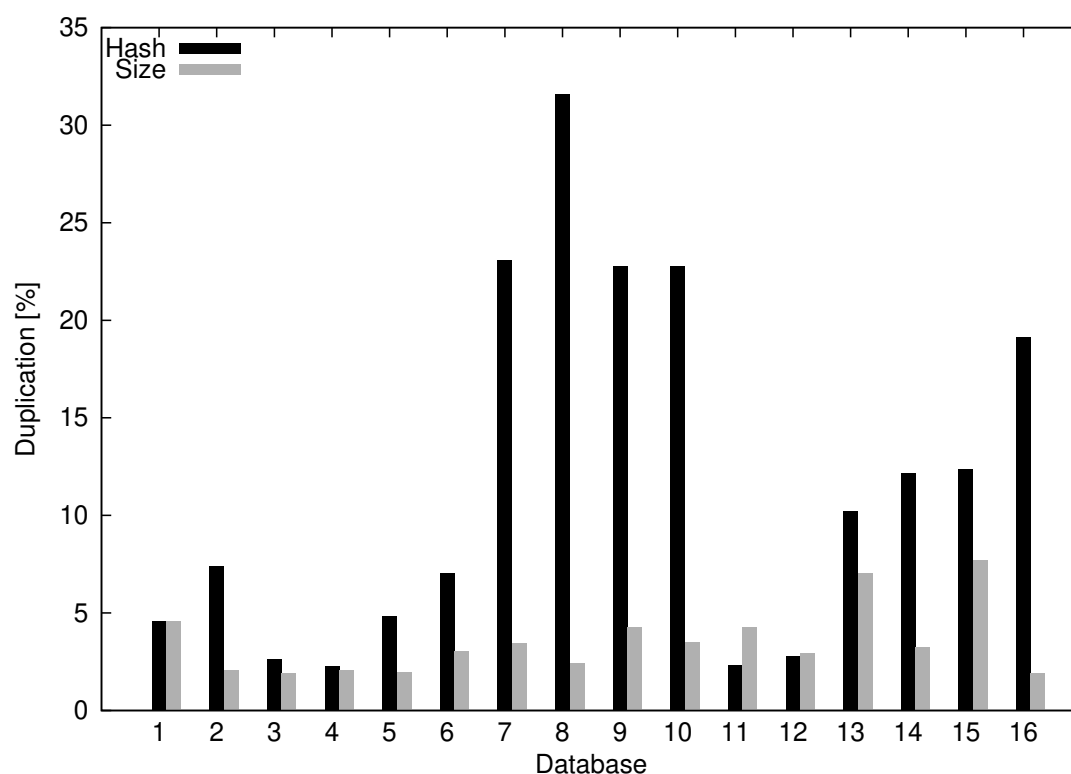


Figure 2.3: Deduplication rate of the corpus databases to NSRL RDS

To be more clear, describing this on the example of the first comparison: *1.sqlite* is the initial database, then $i + 1$ is database *10.sqlite* and so forth until *16.sqlite*, which is the predecessor of *2.sqlite*. In this example, $i + 15$ corresponds to *9.sqlite*, meaning the sum of all databases in the sequence from two to 16.

Figure 2.5 illustrates that the comparison of any database x_j to the remaining databases and their sum is an increasing trend. As expected, the highest number of the same hashes are found in the first compared database and slightly increasing with every incremented database. On average, the percentage of similarity is between three to five percent from the comparison of the initial database to the next and the comparison of the initial database to the sum of all remaining databases (step $i+15$ in Figure 2.5).

2.4.3 Discussion

Those improvements can be applied to the data acquisition process, since they are totally transparent to the user applying them, e.g. the forensic investigator. Since we proved our stated, theoretical approach on a real-world corpus, we argue that this workflow can be beneficial to the majority of forensic cases. However, when comparing the theoretical and tested results, we have shown that the amount of file and size reduction is highly

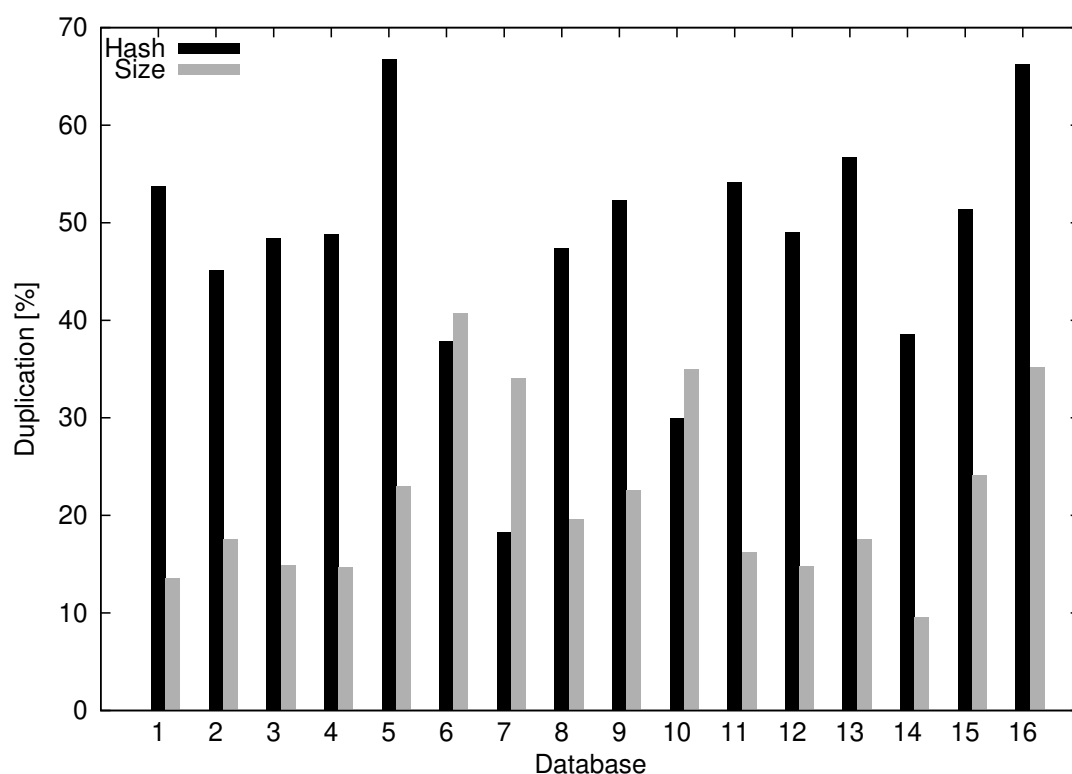


Figure 2.4: Deduplication rate of each corpus database to the remaining databases

dependent on the underlying data corpus. If the number of unique files is low, or the unique files are large in size, the resulting improvements will be lower. Therefore, the NSRL RDS is a good start, but the overall reduction depends on the quality of the whitelist(s) applied by the investigator.

2.4.4 Limitations

Our evaluation is based on largely homogenous set of computers which share the same operating system; our findings can, however, be transferred to cross-platform investigations that are common in real-world cases. In this chapter we clearly demonstrated the trend and showed the possibilities of file-based deduplication during image creation.

2.5 Conclusion

In this chapter we not only showed how an improved forensic process can be used to reduce the amount of storage requirement for forensic investigations by using file whitelisting and cross-device deduplication. While the metadata of duplicate files has to be preserved, our process is particularly useful in cases where the focus of the investigation lies on referenced

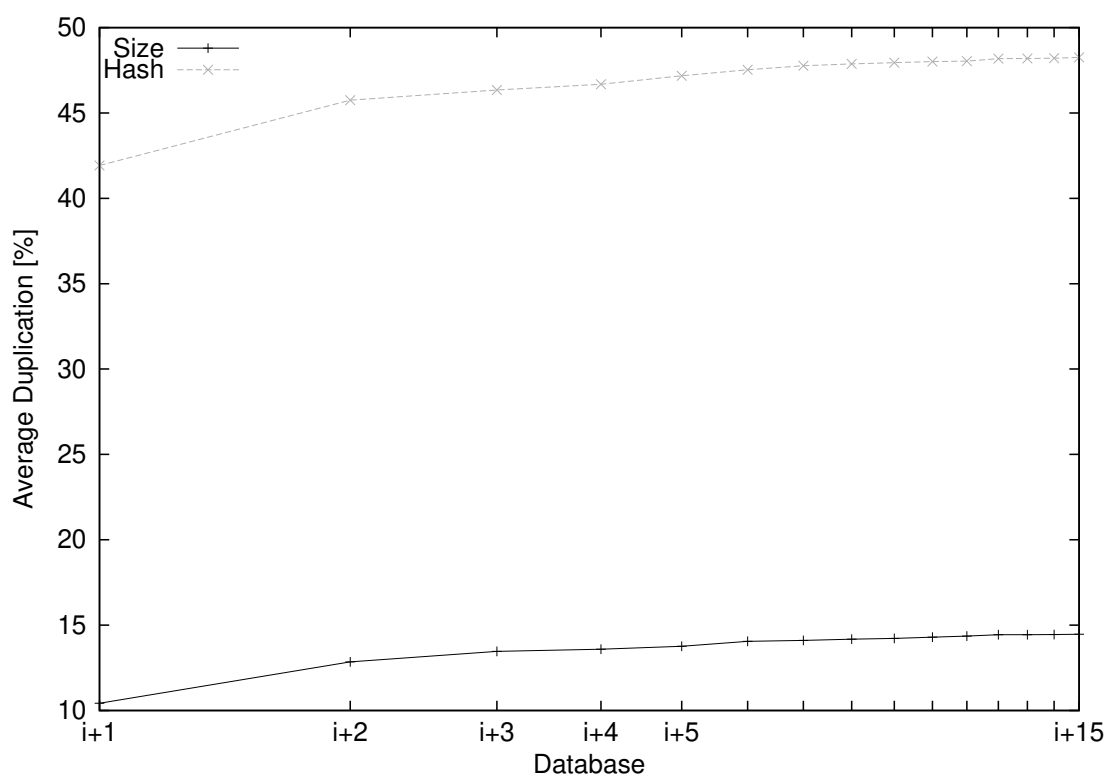


Figure 2.5: Average over the incremental deduplication rate

files in the file system. We described an exemplary use case where file deduplication and file whitelisting were used to save 78% respectively 700 gigabytes of storage capacity.

Additionally we showed on a real-world data corpus consisting of 16 disks that file exclusion during the acquisition process saves storage and therefore processing power. When excluding files found within the NSRL RDS the reduction is about 3.5%, when cross-comparing the databases to each other the reduction is over 22% in regards to their size. Overall we hope that our improved process will lead to interesting discussions in the community as well as an improved standard forensic process in the near future.

CHAPTER 3

Improving Incident Response with Sub-file Hashes from Open Sources

As shown in the previous chapter, file whitelisting harbours a great potential to enhance the process of analysing huge amounts of files. However, it is limited in numerous ways: for one, there is currently just one large corpus of hash values which is publicly shared – the NIST National Software Reference Library, containing 42 million file hashes. Secondly, these file hash values rely on hashing an entire file, and are thus unusable to identify files that are partially modified, or files which have been deleted respectively partially overwritten.

To cope with these problems, we present *peekaTorrent*, a methodology to identify files and file fragments based on data from publicly available file-sharing networks. It is based on the open-source forensic tools *bulk_extractor* and *hashdb* and can be readily integrated into the processes. It improves the current state-of-the-art on sub-file hashing [54] twofold: for one the hashed sub-file parts are larger than pure sector-based hashes, and thus less prone to false-positives for files that share common data segments. Secondly, we solve the problem that an a-priori sub-file hash database is required by creating one that can be shared openly. Lastly, no participation in file-sharing activity is needed as the torrent metadata or “metainfo”, which is stored in the torrent file, already contains all the necessary information including the sub-file hash values. This information can then be used for file and fragment identification and effective file whitelisting, as well as for other use cases. As such, the contributions of this chapter are as follows:

- We present a scalable methodology to identify files and file fragments based on sub-file hashing and P2P file sharing information.

- We collect and analyze more than 2.3 million torrent files, rendering up to 2.6 petabyte of data identifiable using that information.
- We identify several use cases for file (fragment) identification in the context of both file-whitelisting and blacklisting with that data.
- All obtained data and created source code is available online at <https://www.peakatorrent.org>.

3.1 Background

Digital forensics relies on a multitude of information sources to gain knowledge, ranging from hard drives and file system artifacts [23] to the dynamic content of RAM [89] to the user files and programs that store information in log files, SQLite databases, or digital images. This leaves the investigator with a broad spectrum of places where to look, where each investigation depends in its specific context and questions to be answered. The general process outline has been defined in both [19] and [74], whereas a great number of current challenges has been discussed in [52]. Another problem is the increasing spectrum of used devices, ranging from smartphones [66] to smart TVs to numerous other types of devices. Most pressing, however, is the general problem that the average case size is constantly increasing [131]. For one this is due to increasing storage capacities of hard drives, with modern hard drives being able to store many terabytes of data that need to be analyzed with respect to the traditional approach of digital forensics. Secondly, cloud storage services commonly push information automatically from device to device, like pictures taken or files edited, leading to duplicate files across devices. Lastly, the density of digital devices surrounding us is increasing, which is also true for the average number of devices per user.

In recent years, numerous forensic models and publications were specifically targeted to reduce the manual work in investigations with large amounts of data. Among them is the concept of *forensic triage*, which was initially presented in 2006 [128] and more recently quantified in [131] regarding the expected amount of computational power needed. The basic idea is that instead of analyzing all the data there is, only a specific subset of files which are known to be of interest are inspected. Only recently the concept of *sifting collectors* was proposed [125] in which the amount of data to be analyzed is reduced by ignoring known areas on hard drives that are of no particular interest, while still retaining the ability to create bit-identical images if needed. Our approach is different in that it extends the traditional process of forensic imaging by identifying large volumes of both files and file fragments to be either of particular interest (blacklisting), or not of any interest at all as the file is a known, good file (whitelisting).

Both *bulk_extractor* and *hashdb* are two very powerful open-source tools which were published by Simson Garfinkel. *Bulk_extractor* [53] recursively scans hard drive content,

and is able to retrieve information in compressed as well as embedded files like PDFs. It is extremely fast and can use all available cores on a machine to parallelize the task at hand. Hashdb [54] uses efficient algorithms to build a lookup database of hash values much faster than any relational or NoSQL-style database system. It can reliably identify the presence of a given list of target file hash values, and builds on previous work that showed that there is only a small percentage in shared file content on the sector level [161].

3.2 P2P Networks for Hash Values

The basic idea of our approach is to extend the existing knowledge and applicability on sub-file hashing and hash-based carving by leveraging vast amounts of publicly available hash values. While hashing was previously mainly used to uniquely identify entire files of arbitrary size, our concept presented here extends this to hashing variable-sized sub-file portions. Sub-file hashing [161] as well as hash-based carving [51] allow investigators to search for file fragments by hashing either each hard drive sector or aligned blocks of data. This can also be used if there is not enough time available to prove stochastically the presence or absence of specific files, e.g., in well below an hour and with only a relatively small error margin. We extend these concepts by mapping sub-file hashes with data from peer-to-peer file sharing networks with variable block sizes, both usable for black- and whitelisting of large volumes of files as well as sampling. We thus extend existing tools and concepts, such as bulk analysis of forensic media using *bulk_extractor* [53] and *hashdb* [54].

Peer-to-peer (P2P) file sharing applications and protocols rely heavily on hashing for integrity and as a foundation for parallelization, i.e., simultaneously downloading multiple parts of a file from different users for increased performance. While we used the popular BitTorrent file format for our evaluation, in many cases any application that uses sub-file hashing is directly usable: Dropbox for example, a popular cloud storage service, hashes blocks of 4 megabytes using SHA-256 and stores them in a local SQLite database [78, 106]. These sub-file hash databases can also be privately created and maintained, for example based on files and information within a company or an investigative bureau, but across cases. Our particular contribution is to propose that these pre-computed hash lists can be used to identify files and sub-files on hard drives. With millions and millions of torrent files publicly shared online, *peekaTorrent* uses the fact that each and every torrent file indexes all files and also contains their corresponding SHA-1 hash values. For efficiency, the files are split into equally sized pieces or chunks, solely depending on the overall size of information to be shared [33] in powers of 2 starting with 16kb. Thus, by splitting the hard drives into equally sized chunks and hashing them using SHA-1, it becomes a matter of comparing hash values to possibly identify hard drive content without relying on file system metadata. Also, this information is freely available without participating in any form of file sharing activities, but leveraging the initial seeders computing power in hashing any form of content.

Torrent files have a rather simple structure [33]: they contain generic information, e.g., when the torrent was created, which software was used and the specific information of the data to be shared. This includes the size of the blocks, their SHA-1 hash values, and how many there are. During the creation of the torrent file, all containing files are concatenated, and this stream of data is then split into equally sized blocks (except for the last one which does not need to be aligned with the block length). By default, the data is split into 256-kilobyte blocks, but the user can specify arbitrary block sizes during the creation of the torrent file. The size of the torrent file depends mostly on the number of blocks, because it contains an SHA-1 hash value of 20 Bytes for each block. To uniquely identify the torrent for clients and trackers, an SHA-1 hash value is calculated over a subset of the torrents' stored information: the so-called *info_hash*. Figure 3.1 shows a graphical representation of the file format as well as an example from a specific torrent file. The dashed line is the information which is hashed to obtain the *info_hash* value, while for each file the dictionary *files* contains the relative path and the length of the file. *Piece length* is the block size in which the data is split (in the order specified in the *files* field), and the field *pieces* contains the concatenated SHA-1 hash values.

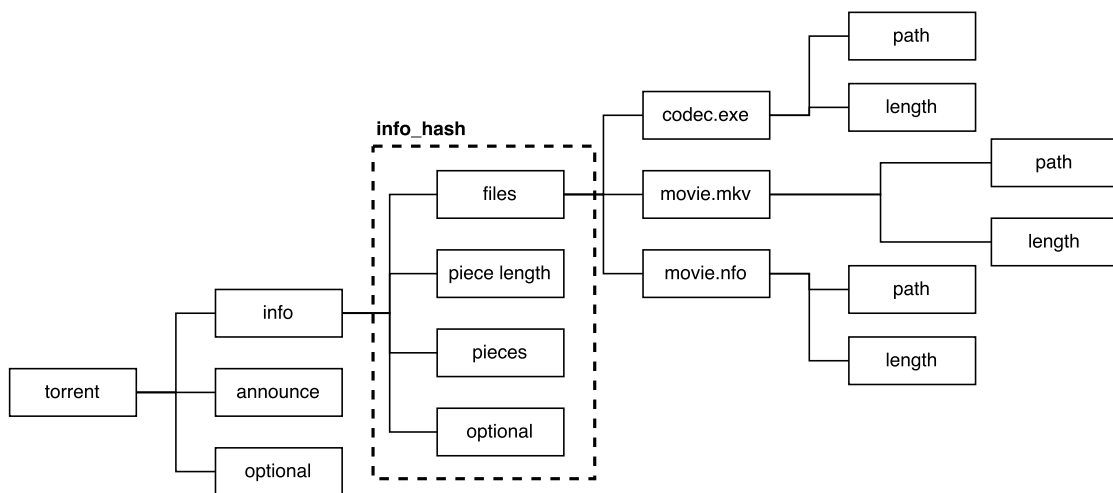


Figure 3.1: File content in a torrent file

3.2.1 Problem of Non-Aligned Files

One of the problems when using torrent files is the way these files are created: prior to hashing all chunks, the files are concatenated (in arbitrary order). If a chunk contains parts of two files, we cannot use the resulting hash value. This means that only files which are larger than the piece length can be identified, thus biasing the general applicability towards large files (which is obvious when looking at content from file sharing networks). Figure 3.2 shows a representation of block hashes in torrents, with the same content as Figure 3.1: the SHA-1 value of the first piece is usable, as *codec.exe* spans into the second

piece. As such, it can be used to uniquely identify that this file has been stored on the hard drive by hashing any hard drive with the same hashing window as the `piece_length` of the torrent. This can be readily integrated into `bulk_extractor`, which already facilitates the necessary requirements by default. If the first file is longer than in our example, and spans, e.g., n pieces in the torrent file, any of these areas on disc can identify the file as long as the data is consecutively stored somewhere. The second piece in Figure 3.2 is not usable for our proposed methodology, as it contains content from both the first and the second file. While it could theoretically happen that the operating system allocates the information in such a way that the hash value could be used, this is not necessarily the case as the files can be stored at different locations on the hard drive and in different orders. The third piece (i.e., the second piece that contains content from `movie.mkv` in our example) is usable if the missing length of the file in the beginning is used for *offset hashing* – it is no longer the `piece_length` which can be used for chunk hashing during acquisition, but rather aligned to the hard drive sectors, which tremendously increases the hash values to be calculated during analysis. Again, this is already integrated in `bulk_extractor` and the problem remains CPU-bound, which means it is solvable if enough computation power is at hand. The hash value for the last piece is unusable, as it must not be of the same length as the others [33], i.e., there is no padding for torrent files.

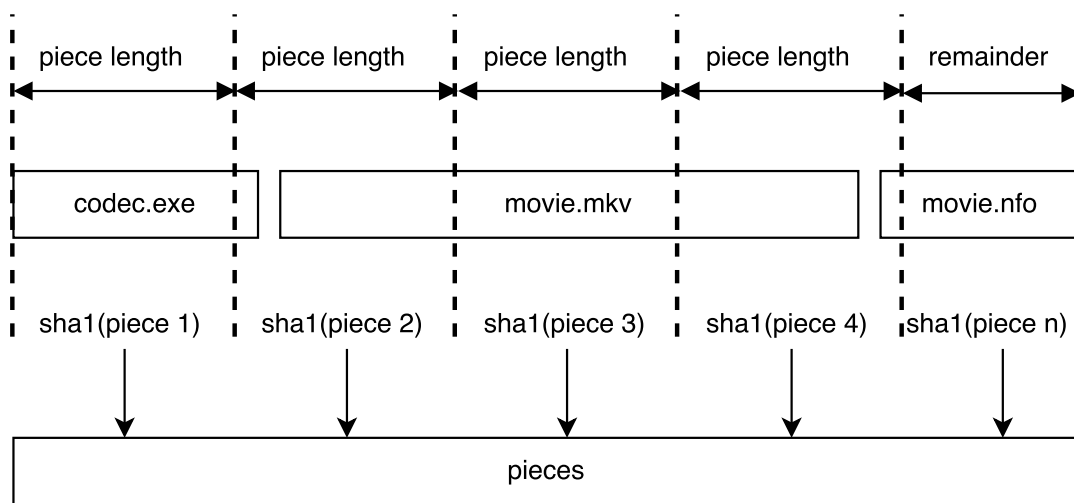


Figure 3.2: Chunk hashes

In the following, we discuss the different use cases where such a vast amount of file fragment information can be of use in the particular context of digital forensics. Other protocols are probably equally suitable, but have not been investigated in detail for this work, e.g., Kademlia [94] as well as distributed hash tables in general [139] often use SHA-1 hash values for searching.

3.2.2 Use Case 1: File Whitelisting

File whitelisting is a well-known technique to identify files that are common and of no particular interest during an early phase in digital investigations. One of the most commonly used databases of hash values is the NIST National Software Reference Library (NSRL) reference data set¹ which comprises at the time of writing of more than 43 million file hash values. Most of these hash values include binaries and program libraries for software on Windows, whereas our collected data contains information of relevance independent of the used operating system, and of much larger file size. While NIST also releases block hash values for the first 4k and 8k of about 13 million files, our dataset is able to identify popular files like movies, TV episodes or other commonly shared files on file sharing networks, even if they are deleted and some sectors were already overwritten by the file system.

3.2.3 Use Case 2: File Blacklisting

File blacklisting is used to find and identify files of particular interest for a specific investigation. While in our evaluation the usability of our data is mostly limited to cases of copyright infringement, it is still of use for investigations in general and might lead to new insights. Nonetheless, building a private sub-file hash database is always a possibility if a script can be used to hash blocks of arbitrary length of, e.g., all e-mail attachments in a company, all files on a Sharepoint server or source code within a company. This could also include illegal material like pictures and videos related to child pornography. Instead of using perceptual hashing [17] – as used by online services like Twitter and Facebook to detect such files [70] – sub-file hash values of variable block length can further identify files like these without access to such perceptually hashed data.

3.2.4 Use Case 3: File Fragment Identification

By default, file systems in modern operating systems do not overwrite files once they are deleted, but rather delete the index pointing to the data or mark the affected storage areas as free-to-use [23]. Depending on the operating system and the file system in use, as well as the actual user behavior, it is usually unpredictable when a specific area will be overwritten. Both methods in our approach described so far work for partially overwritten files, as they do not rely on file system metadata. This was already argued in [161] for sector hashing. As long as the data on a disc is not completely overwritten and leaves at minimum the piece length of the torrent files untouched, peekaTorrent will find it.

3.2.5 Shifting the Bottleneck

Considering these three use cases, the overall performance scales linearly with the number of available CPU-cores, similar to `bulk_extractor`. Sub-file hashing can leverage multi-

¹Online at <http://www.nsrl.nist.gov/>

core CPUs and scales with the number of available cores. As the file system metadata is not needed, there is also no need for disk seek operations. All the data from the hard drive can be split in constantly sized chunks, and processed recursively using the hashdb scanner within `bulk_extractor`.

3.3 Evaluation

To evaluate our methodology we implemented all the steps of the processing outline described above. This includes software we wrote to collect torrent files from the Internet and tools to process and use them within the context of a forensic investigation, see <https://www.peakatorrent.org>. This section shows and underlines the applicability of the proposed approach and the methods applied for gathering torrents on a large scale.

3.3.1 Data Collection

Collecting a large number of torrents from the Internet is non-trivial, as new torrents are added constantly and older torrents become unavailable once they are no longer shared.

Only a minority of websites hosts the torrent files containing all the sub-file hash values themselves, but rather rely on sharing magnet links that point to the information in the completely decentralized *distributed hash tables* (DHTs) [163].

To collect torrent files we focused on the following three main sources: (i) The Pirate Bay², (ii) kickassTorrents³, and (iii) various data dumps, e.g., from *openBay*⁴. For (i) and (ii) we implemented a crawling framework which recursively crawls and parses both websites for every magnet link listed there. After that we extracted the torrent `info_hashes` from the magnet links and constructed a download link for the torrent cache website <https://torcache.net/>. For (iii) and those torrent files which were not hosted at torcache.net we implemented a DHT lookup service, similar to the one Wolchok et al. used in their work [158]. The crawlers for (i) and (ii) were crawling the entire websites, including all subcategories to get the full archive for a specific point in time (January 2016 in our case).

From the various *openBay* dumps we were able to extract close to 30 million `info_hashes`. The dataset from *isohunt* contained 7.8 million `info_hashes`, while the complete archive for *openBay* included 23.5 million hashes. Both data sets were created after the police raid against Pirate Bay in December 2014 caused the website to be shut down. Previously generated data sets also include one notable xml dump of the Pirate Bay from February 2013 (about 2 million `info_hash` values). Not all of these files were retrievable using the

²<https://thepiratebay.org/> and its alternative TLDs

³<https://kickass2.biz/>

⁴<https://github.com/isohuntto/openbay-db-dump>

DHTs, in fact only a small fraction and in particular only newer files. The biggest fraction of torrent files we collected came from kickassTorrents and torcache.net, as torcache.net is used by default to distribute torrent files on behalf of kickassTorrents. So far we have collected 2.3 million torrent files. Our data collection is still going on, and as such the data we collected can only be considered a snapshot in time. Further processing was then done using Python as well as *hashdb*, which was used to efficiently store and query the sub-file hash values.

3.3.2 Theoretic Evaluation

Fragmentation of files can be a limiting factor using real cases, as for each time a file is fragmented one chunk (of arbitrary length) is no longer identifiable. Since there is no public instance of a SHA-1 pre-image attack, finding a small number of chunks using peekaTorrent has a very small likelihood to be coincidental and can be used for further analysis steps during the investigation. Compared to previous work [161, 54], the number of false positives is greatly reduced, as the block length used for hashing is larger than the previously used sector/cluster size of 512 or 4096 bytes. Hashing a larger file block, e.g., 256 kilobytes, drastically reduces the probability of resulting in the same hash value (for all files independent of each other). This also implies that shared file content across files, such as the ramping structure for Microsoft Office files as discussed in [161], is evaded as the block length increases.

3.4 Results

Overall, we collected and analyzed more than 2.3 million torrent files. These torrents comprise 3.3 billion block hash values. From these 3.3 billion block hash values, approximately 48% (or 1.62 billion hash block values) are usable to identify millions of files using various block length. Another 50% (or 1.66 billion hash block values) are usable even though the files do not align with the torrent chunking. 1.1% of the 3.3 billion hash values (or 39 million hash block values) are not usable for our approach, as the blocks and their corresponding hashes comprise content of two or more files. The exact numbers for the most popular torrent block lengths of 2^n (for various n) is shown in Table 3.1, with exotic chunk sizes omitted ($n=2,871$) for the sake of brevity.

From the 2.3 million torrent files we are able to identify 2.6 petabytes of data using Peek-a-Torrent, or 32 million files. Regarding only the most common chunk sizes with 100,000 or more torrent files found using our methodology, we are left with 2.1 million torrents. The pre-computed hashdb databases as well as the raw torrent files and the source code used for this chapter can be found on our website <https://www.peekatorrent.org>.

block length	torrents	chunks	usable chunks		offset chunks		unusable chunks
16k	75k	146m	123m	84%	22m	15%	305k
32k	95k	171m	112m	65%	58m	34%	662k
64k	335k	217m	124m	57%	90m	41%	2m
128k	201k	227m	115m	50%	109m	48%	2m
256k	669k	1.329b	631m	47%	690m	51%	8m
512k	297k	401m	201m	50%	194m	48%	5m
1024k	307k	357m	165m	46%	187m	52%	5m
2048k	170k	201m	75m	37%	121m	60%	4m
4096k	161k	229m	58m	25%	162m	70%	8m
8192k	18k	27m	8m	30%	17m	65%	975k
16384k	2k	3m	315k	9%	2m	84%	198k
Sum:	2.3m	3.314b	1.615b	48%	1.658b	50%	39m

Table 3.1: Results of data collection for 2.3 million torrent files

3.4.1 hashdb

We then imported the usable sub-file hash values for all torrents with a piece length of 256k into hashdb [54]. As it can be seen in Table 3.1, this sums up to 631 million hash values. From these 631 million only 474 million are unique, because of duplicate sub-file hash values. This is due to the fact that the same files can be contained in different torrents, e.g., duplicates for each kickassTorrents and Pirate Bay. Torrent files that became repackaged with different files or file ordering can be another reason to cause this rather large discrepancy. hashdb can then be used to deny that a given sub-file hash value is part of the database using Bloom filters. Otherwise the database is queried, and both filename and info_hash are returned if a corresponding hash value is found. All the features and APIs provided by hashdb are thus fully usable, and the entire project is well documented and active⁵.

While the majority of sub-file hash values are unique within the data we collected (474 million), the long tail of duplicates can be seen in Figure 3.3. The x-axis accounts for the number of duplicates found, starting from hash values with 10 duplicates or more. Note that the y-axis is log-scale. In the data there are also 17.8 million distinct sub-file hashes that occur twice, 2.5 million that occur three times, and about 440,000 that occur four times. We speculate that these hashes are again caused by some form of release group information or an embedded URL. The by-far largest number of duplicates observed was caused by one particular hash that occurs 8,462,788 times. We would speculate that this is caused by the “null” hash, for data areas that contain only zeros.

⁵<https://github.com/NPS-DEEP/hashdb>

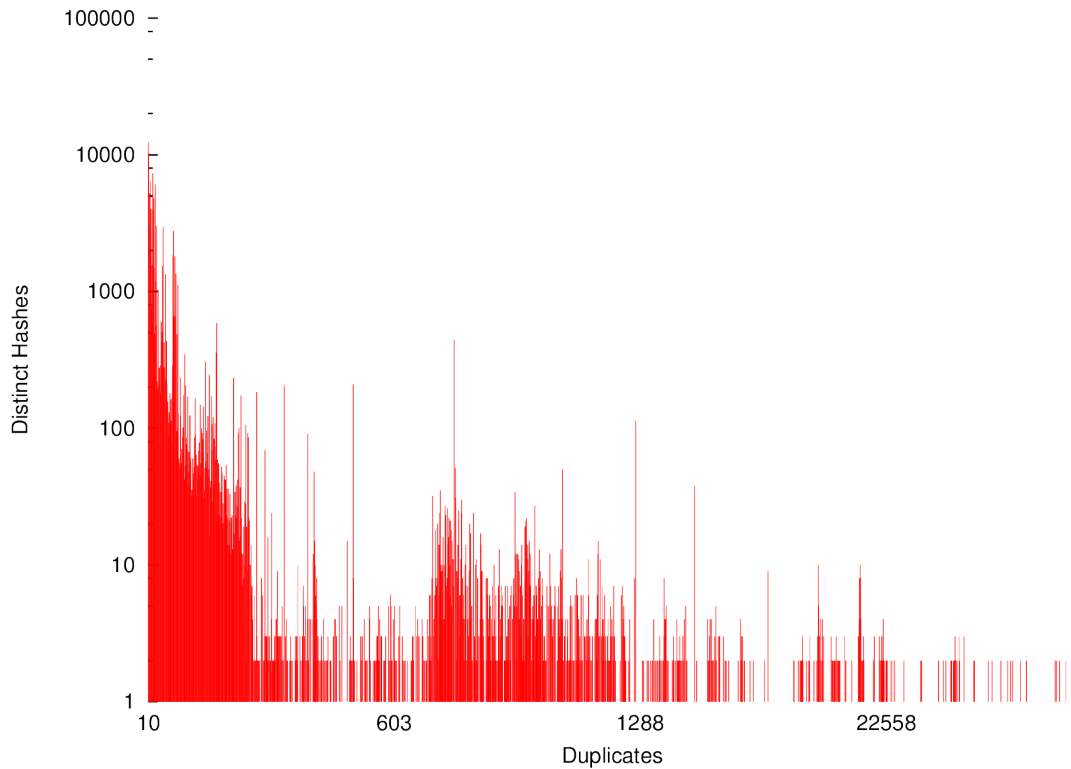


Figure 3.3: Distribution of sub-file hash duplicates

3.4.2 Real Runtime on Limited Hardware

To evaluate our approach further, we took a 5-year old notebook and created a one gigabyte image from a USB thumb drive. The notebook was a Lenovo X200s, with a Core 2 Duo processor (L9400), 4 GB of RAM and a regular hard drive. On the thumb drive we stored the ISO file for the current version of Ubuntu Desktop, which we downloaded over BitTorrent. We created a fresh hashdb database, and seeded it with the extracted SHA-1 hashes of the torrent file. Overall, we extracted 1158 hash values for the Ubuntu image, the chunk size was 512k. We then used a custom module for `bulk_extractor` to generate SHA-1 hashes of all blocks `bulk_extractor` processes, and disabled all other plugins.

Running `bulk_extractor` with solely the SHA-1 plugin activated on the notebook took 220 seconds to process the 1 GB image file. Since the CPU has two cores, two threads were spawned to process the image. From the 1158 chunks, 1154 were successfully identified using `peekaTorrent`. Three chunks could not be found since the file was stored fragmented in three fragments (verified manually using `fwalk`), and the last hash value is unusable as it has a different chunk length. Running the same analysis on a modern Xeon with 8 cores plus Hyper-Threading took less than 23 seconds. Running the same image against the hashdb database of all 474 million chunk hashes took 38 seconds. Since we do not

aim to evaluate the performance of either `bulk_extractor` or `hashdb`, we do not go into details of further performance numbers. Also, the average fragmentation on hard drives depends heavily on the type of usage, size and operating system. Measuring this for the average case is beyond the scope of this thesis.

3.5 Discussion

Our results show that a rather large number of block hash values is usable to identify files based on the data we collected from BitTorrent files, somewhere close to 98%. Due to the nature of file sharing networks and the content distributed there we assume that this is possibly biased, that these networks commonly share large files like movies in high quality. We did not investigate the distribution of filenames and file sizes to what extent one can expect that the largest file is the first in the torrent file. We assume that this is specific to the application that created the torrent, as this is not specified in the file format of BitTorrent [33].

Half of the usable chunk hashes come with an arbitrary offset due to the placement of the affected files. This is caused by the particularities of BitTorrent files. However, since `bulk_extractor` processes pages of memory without any file system information, these artifacts are also retrievable (as long as the file is larger than the chunk size). Other sources for sub-file hashing have to be investigated, like other P2P protocols or cloud storage solutions such as Dropbox. We expect similar functionality from other cloud storage solutions like Google Drive, OwnCloud or Microsoft OneDrive as well, where the local data structures could be used as a source for history hash values. Still, using the data we collected we can identify up to 2.6 petabytes of data for 3.3 billion chunks. We expect these values to increase, as we will keep collecting data and publishing it on our website.

Regarding the forensic application and typical use case, many scenarios come to mind. First, it depends on the data sources used for seeding the sub-file hashing – this can be for example all sent e-mail attachments in a company, a stack of sensitive corporate documents or encrypted data blobs in the corporate context. Secondly, this can be easily enlarged by investigators via adding data from private repositories of interesting files, file archives or any other data source at hand – like USB thumb drives – or portable hard drives, and hashing it in sub-file chunks. Another example could be the cross-linking of files between hard drives: if any of the hard drives during an investigation is hashed with a particular chunk size, all other related drives can use this information to identify non-fragmented overlaps. After all, this was obviously the original motivation behind the tight connection between `bulk_extractor` and `hashdb`. Foremost, `peekaTorrent` allows for hard drives without any meta information at all to find clues on the content – as long as the hard drive is not encrypted.

3.5.1 Limitations

While 2.6 petabytes of identifiable files sounds like a lot, its usefulness depends on the particular kind of investigation. If the goal is to whitelist as many files and file fragments as possible on a diverse set of machines, our approach looks promising. As always in digital forensics, it depends, however, on the specific context of the investigations and the questions of interest. For more specific investigations it depends on the type and volume of data – creating sub-file hash values of variable block length is easily scriptable, so if a large repository of files is available, our methodology is applicable. This can be, for example all attachments from an e-mail server, malicious files like malware from anti-virus companies, or even smaller sets of files with a direct connection to an investigation.

Another limitation is the behavior of storage devices, operating systems and file systems: SSDs regularly delete artifacts within the free space using the TRIM command [15], and depending on the operating system and file system, fragmentation can occur. There are no current numbers on the amount of fragmentation happening, with the latest study on file system metadata being already close to a decade old [2]. Also, the approach only works for files which have at least a file size bigger then the hashing window respectively the torrent piece length. Based on our findings with peekaTorrent, only files with a minimal size of 16 kilobytes are identifiable, while a vast amount of files needs to have at least 256 kilobytes due to the nature of the seeding data.

3.6 Conclusion

In this chapter we have demonstrated how vast amounts of sub-file hash values can be of use in digital forensics. We evaluated the idea of using torrent files from popular file sharing platforms and collected more than 2.3 million torrent files for our analysis. Based on these torrent files we extracted more then 3 billion SHA-1 sub-file hash values and were able to identify up to 32 million files or 2.6 petabytes of information using this data set. Both the collected data and the written software tools are available under open source licenses.

CHAPTER 4

Filesystem Timestamp Steganography Detection

The previous chapters lined out how to cope with the analysis of exponentially increasing amounts of data. But cybercrimes do not necessarily involve file contents only. Steganographic techniques are able to hide the existence of information passing through communication channels or resting in storage media for later access. These techniques are useful in a wide range of real-world scenarios, including but not limited to: circumventing censorship and restrictions imposed by governments and other adversaries [3, 6], assisting whistleblowers when disclosing documents [59], and supporting businesses to protect strategic corporate information during transmission [38].

Numerous steganographic techniques have been proposed and analyzed in the research literature [164]. The analysis focuses on criteria such as the achieved secrecy on specific application scenarios, the steganographic channel capacity, and the information channel utilization.

Storage or format-oriented steganographic techniques hide information in logical channels by utilizing redundant or unused fields in format specifications. This includes, among others, the master boot record (MBR) of non-bootable hard disks and the unused disk space caused by the misalignment of hard disk sector size and file size [77].

Modern filesystems support a wealth of operations that span beyond the primitive of mapping files into sequences of hard disk sectors. The filesystem specifications define additional data structures (i.e., “metadata”) to describe information like the owner, the access permissions, and the date and time when important file events took place.

We therefore propose and explore in this chapter, to the best of our knowledge for the first time in literature, the applicability of *filesystem timestamps* as a steganographic channel. Additionally, we evaluate the steganographic capabilities of such channels and propose techniques to aid digital forensic investigations.

More specifically, in this chapter, we make the following contributions:

1. We analyze the granularity of the timestamps that modern filesystems implement, and we evaluate their applicability for steganographic applications.
2. We propose the use of timestamps as a means to hide information in NTFS and other filesystems with sub-second timestamp granularity.
3. We describe a system design and a proof-of-concept implementation that support different levels of possible capacity to securely hide data on NTFS volumes.
4. We validate the proposed system using real-world and synthetic datasets, and we show that the embedded steganographic information cannot be distinguished from the information produced by normal filesystem operations.
5. We discuss the digital forensics implications of this new steganographic method.

Furthermore, in this chapter, we answer the following questions:

1. How do different storage media and connection interfaces affect the timestamp patterns?
2. How do different file creation approaches affect the timestamp generation patterns?
3. How does the regular use of the filesystem affect the capacity of the channel over the sub-second part of the timestamps?
4. How do practices followed in enterprise environment affect the channel capacity?
5. How can we utilize these information to design appropriate detection techniques and mechanisms?

4.1 Background

4.1.1 Data Hiding

Early works on digital steganography focused on hiding data in the clear, deriving and discussing different methods of embedding data, and arguing how steganography is and probably will be used in the present and in the near future [73, 164]. Such works did not anticipate the widespread use of the personal digital devices and the role of the Internet in our daily lives [48, 8].

A considerable amount of research was devoted to embedding unobservable communication within normal network traffic, ranging from the utilization of TCP/IP timestamps [56] to the more general usage of TCP/IP fields [107]. Many implementations of steganography hide encrypted data in innocent-looking network traffic (e.g., `ptunnel` [140]), header

fields [136], or use timing intervals and artificial transmission delays for information transmission [85, 14, 95]. While it has been shown that secure steganographic protocols are feasible, we are still lacking functional implementations and widespread use of such tools [67].

A second line of research focused on embedding unobservable information within the contents of stored files, introducing undetectable degradation of multimedia quality (e.g., manipulating the low-significance bits of pixel representation in images [13]), the color palettes in GIF images [49], or (possibly) encoding information in YouTube videos that look like static snow [157].

4.1.2 Filesystems

A plethora of different filesystems is available, including FAT and NTFS for Microsoft-Windows-based devices, ext4 and btrfs for GNU/Linux systems, and HFS+ for Apple OS X and iOS devices¹. Most of them store different artefacts at various levels of granularity and detail, collectively known as “filesystem metadata”.

Filesystem metadata can be classified in five categories: *file system*, *application*, *file name*, *content*, and *generic* metadata [23]. *File system* metadata are information on how the filesystem is to be read and where the important data structures reside. *Application* metadata are information useful for the application utilizing the filesystem, such as the file owner and the file access permissions. *File name* metadata are information for the human-readable names mapping to logical data locations. *Content* metadata are information about the logical addressing of the files, the file allocation status, and the actual data of the files. *Generic* metadata are information mostly used internally by the filesystem for its operations. This includes information such as the timestamps of various events in the lifecycle of a file.

4.1.3 Steganography using Filesystem Metadata

The topic of hiding data in filesystem metadata was heavily discussed in the late 1990s [7]. Back then, export restrictions on the use of strong cryptographic algorithms outside the USA were in place, and there was an increased concern by the public regarding key escrow. StegFS, a steganographic filesystem compatible with the Linux ext2 filesystem, was developed [97, 120]. This filesystem achieved plausible deniability of the hidden content thanks to its indistinguishability from unused content. This behavior was achieved by applying encryption on the content under the assumption that a good encryption algorithm ensures that encrypted data appear as random data. However, the use of StegFS is not undetectable as the needed filesystem driver is not hidden. Additionally, there is no integrity check of the data. Thus, StegFS cannot recover from any kind of intrusive data modifications.

¹An exhaustive list is provided in the Wikipedia entry available at https://en.wikipedia.org/wiki/Comparison_of_file_systems.

Encoding (hiding) information in the order that a filesystem indexes the file names is explored in [12]. The approach is applicable only in the case of a FAT filesystem and cannot be generalized. The file fragmentation is explored in [77]. This approach introduces significant performance penalties, more evident in magnetic storage media, in the form of delays when accessing the file contents. This delay is due to the heavy file fragmentation that is enforced in order to create the steganographic channel. The delay and the heavy fragmentation can serve as indicators for the presence of steganographic information, thus *defeating* the steganography. Furthermore, (automatic) defragmentation of the storage medium can *destroy* the steganographic information.

Application metadata (e.g., the file owner or the file access permissions) can encode only a few bytes of information and the encoding is easily detected. For example, it is technically feasible to attach an arbitrarily large list of user–permission pairs in an NTFS file, even by referencing non-existent users [121]. However, the mismatch of the users mentioned in the system user list and the user–permission pairs, on top of having such long lists in first place, would raise suspicions in a forensics investigator.

The file name cannot be considered as a good candidate for steganographic operations. Indeed, an odd pattern of filenames will look instantly suspicious.

Mixing steganographic information with the actual contents of a file is studied extensively [73, 27, 88, 4, 69]. Format containers for multimedia content (e.g., audio or video) are transparent to and independent of the underlying filesystem that hosts the multimedia file. Thus, a filesystem-level analysis will not be able to disclose the presence of steganographic information in a format container. Also, we note that multimedia transcoding can effectively *destroy* the steganographic information without significantly affecting the original information channel.

Generic metadata, such as temporal information describing the lifecycle of a file, are very sensitive to both the actions of the user and the operating system itself. For example, certain timestamps of file events can be overwritten at any moment while using the filesystem in a normal way. This includes a timestamp of the (last) file modification and (last) access of the file. The fragility of the temporal information might be the reason why, to the best of our knowledge, timestamps have not yet been explored as a steganographic means.

4.2 Timestamps in Modern Filesystems

We analyze in the following paragraphs how modern filesystems use timestamps. The assumption we seek to validate is that there is unused (redundant) capacity in timestamps that is sufficient enough to create a logical channel with steganographic strength.

NTFS is the standard filesystem for Microsoft Windows operating systems. NTFS uses the number of 100 nanoseconds passed since January 1, 1601 for its timestamps [102]. The timestamps are saved as 64-bit values in the `$Standard_Information` field of the Master File Table (MFT). Additionally, they are saved in the NTFS attribute

`$FILE_NAME`. Each file has four 64-bit timestamps: (i) creation of the file, (ii) last access of the file, (iii) last modification of the file, and (iv) last modification of the corresponding MFT entry.

ext4 is the successor of the Linux standard filesystem `ext3`. Ext4 uses 64-bit values to represent timestamps with nanosecond granularity [93, 46, 76]. Ext4 uses the following four timestamps per file: (i) creation of the file, (ii) last modification of the file, (iii) last access of the file, and (iv) the last attribute modification of the file (e.g., permissions or ownership).

btrfs is the upcoming major filesystem for the Linux operating systems [126]. It is a “copy-on-write” filesystem based on B-trees. All file timestamps in `btrfs` support nanosecond granularity and are saved as 64-bit values [123]. The first 32 bits of the timestamps are the seconds since the epoch (January 1, 1970) and the remaining 32 bits are the nanoseconds since the beginning of the second. The provided per-file timestamps include: (i) creation, (ii) last modification, (iii) last modification of the file’s attributes (e.g., permissions or ownership), and (iv) last access.

ZFS is intended to be a highly performing, decentralized filesystem [127]. The following per-file timestamps of ZFS have a nanosecond granularity, saved in 64 bits each: (i) creation, (ii) last modification, (iii) last access, and (iv) the last attribute modification [142].

FAT32 is the predecessor filesystem of NTFS on the Microsoft Windows operating system. FAT32 uses three different timestamps per file: (i) creation, (ii) last modification, and (iii) last access. The first two timestamps are saved as 32-bit values and the last one is saved as a 16-bit value. The difference comes from the fact that the first two timestamps are provided with a granularity of two seconds, whereas the date of last access is provided with a granularity of one day [50].

HFS+ is the standard filesystem for the Apple Macintosh and iOS devices. HFS+ uses the following per-file timestamps: (i) creation, (ii) content modification, (iii) last attribute modification, (iv) last access, and (v) the last backup [21]. All of these timestamps have a granularity of one second and are saved as 32-bit values.

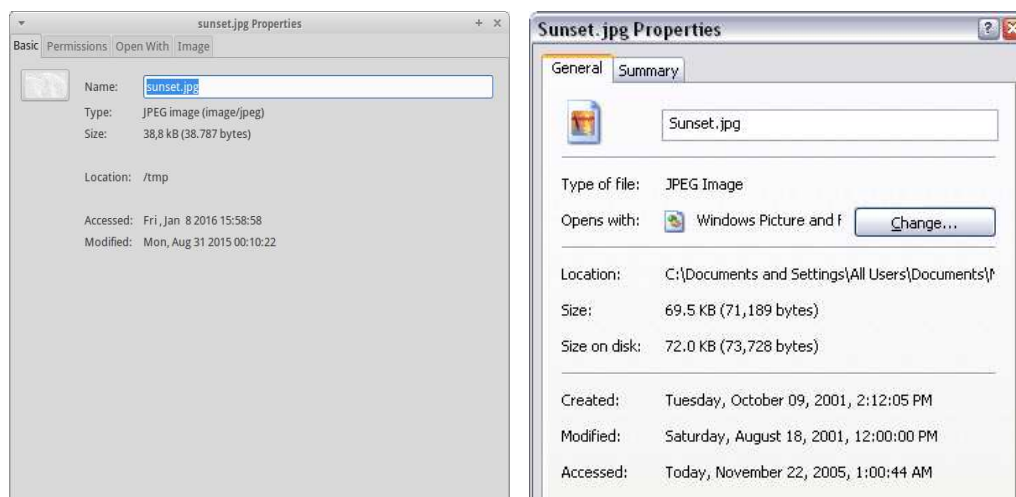
ext3 is the successor of the `ext2` filesystem and enhances it by providing journaling capabilities. Ext3 uses three timestamps per file: (i) last access, (ii) last modification, and (iii) last attribute modification. The timestamps have a granularity of one second and are saved as 32-bit values. The use of the undocumented large-size `inode` feature can increase the granularity of the timestamps to one nanosecond [10].

Table 4.1 summarizes our analysis. We confirm that many modern filesystems use 64-bit values as timestamps and offer sub-second granularity [10]. This statement covers all filesystems that mainstream consumer operating systems use or access nowadays (e.g., Apple OS X, Google Android, GNU/Linux, and Microsoft Windows) with the exception of the HFS+.

4. FILESYSTEM TIMESTAMP STEGANOGRAPHY DETECTION

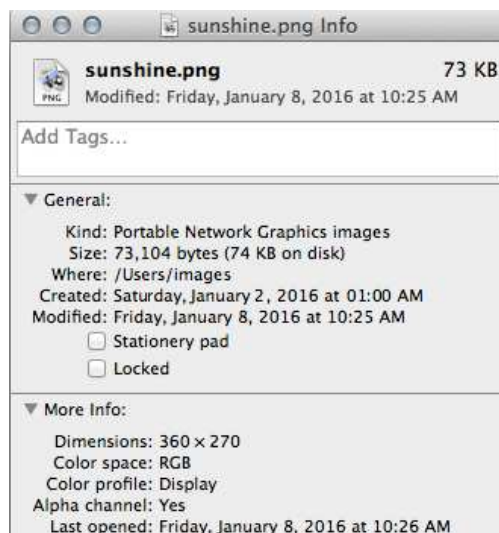
Three file timestamps, namely *creation*, *access*, and *modification* are supported by almost all the analyzed filesystems. All three timestamps store date and time information with sub-second granularity (one or 100 ns).

The nanosecond precision is not communicated, explicitly or implicitly, to the end users who access the filesystem. They are confronted with file timestamp information that resolves to a second granularity, as depicted in Figure 4.1. Thus, there is an information gap between how timestamps are stored and how they are used.



(a)

(b)



(c)

Figure 4.1: How file timestamps are displayed to the users: (a) Ubuntu Linux, (b) Microsoft Windows, (c) Apple OS X.

The *creation* timestamp is by and large a static piece of information, as it refers to a unique event, the creation of the file itself. The *access* and *modification* timestamps are updated each time a file is accessed or modified.

Modern operating systems are exploiting latest advances in storage technologies to deliver increased performance and reliability while reducing costs. USB flash drives and SSD storage media are commonplace. In this setting it is advisable, if the application scenario allows so, to reduce filesystem overheads for timestamp housekeeping. This includes disabling the update of per-file access and/or modification timestamps. Such an approach can increase both the performance and the lifetime of a storage medium. Yet, in most consumer-grade usage scenarios, we can expect that only the *access* timestamp remains intact.

The analysis above validates the first part of our initial assumption: there is unused (redundant) capacity in filesystem timestamps. Depending on the filesystem and usage scenario, this capacity ranges from one to nine bytes per file. With modern filesystems hosting hundreds of thousands or even millions of files, this provides enough capacity for storing up to a few megabytes of extra information. In the next sections, we explore how the available capacity can be utilized to create a logical channel with steganographic strength.

4.3 Steganography Based on File Timestamps

We assume a threat model where the attackers can inspect the file contents and can manipulate the filesystem metadata. Also, the attackers can freely remove, rename, or insert new files in the filesystem, and they accept the associated risk of thereby disclosing their presence.

We aim for a steganographic storage system based on file timestamps, namely TOMS (Time-On-My-Side) that is stealthy, robust, and applicable in a wide range of scenarios. “Stealthy” means that the attacker cannot deduce the presence or absence of steganographic information by examining the timestamps. Thus, the attackers are left only with the option of a denial-of-service attack i.e., to overwrite all timestamps and destroy the steganographic channel, thereby disclosing their presence. “Robust” means that the system can sustain and recover from file manipulation attacks. “Widely applicable” means that the system can be configured to match different operation scenarios, balancing performance and secrecy.

4.3.1 System Design

The aim of the TOMS system is to hide an input (the *message*) inside the metadata of the hosting filesystem (the *carrier*). For the sake of clarity, we assume that the system can identify the necessary space (i.e., the file timestamps to use) and that all the necessary space is already available. We will return on this issue at the end of the design description (cf. Section 4.3.2).

Table 4.1: Characteristics of filesystem timestamps.

Filesystem	File timestamp	Size	Granularity
NTFS	creation	64 bits	100 ns
	access	64 bits	100 ns
	modification	64 bits	100 ns
	modif. of MFT entry	64 bits	100 ns
ext4	creation	64 bits	1 ns
	access	64 bits	1 ns
	modification	64 bits	1 ns
	attribute modif.	64 bits	1 ns
btrfs	creation	64 bits	1 ns
	access	64 bits	1 ns
	modification	64 bits	1 ns
	attribute modif.	64 bits	1 ns
ZFS	creation	64 bits	1 ns
	access	64 bits	1 ns
	modification	64 bits	1 ns
	attribute modif.	64 bits	1 ns
FAT32	creation	32 bits	2 sec
	access	16 bits	1 day
	modification	32 bits	2 sec
HFS+	creation	32 bits	1 sec
	access	32 bits	1 sec
	modification	32 bits	1 sec
	attribute modif.	32 bits	1 sec
	backup	32 bits	1 sec
ext3	access	32 bits	1 sec
	modification	32 bits	1 sec
	attribute modif.	32 bits	1 sec

The design of TOMS follows a layered approach. From top to bottom, the system comprises: (i) a storage container layer for the message, (ii) an error correction layer for redundancy, and (iii) an encryption layer.

4.3.1.1 Storage Container Layer

The storage container layer maps the message into the underlying file timestamp metadata elementary storage units. The naïve approach for keeping track which files and directories have been used to embed the information is to keep an *encrypted metadata file* with the absolute paths of the files and directories used. The metadata file approach has the benefit that the correct ordering of the files to extract the information is trivial. Also,

this file does not necessarily need to be stored in the same filesystem with the hidden data. Rather, it can be stored in another storage media altogether. This is beneficial, since the very presence of a metadata file inside the examined filesystem is neither elegant nor stealthy, even if its contents are encrypted. On the contrary, such an encrypted file can raise further suspicions.

A second option is to reliably embed and extract information *only* based on the files and their timestamps. This can be realized using *oblivious replacements* on whole filesystems (e.g., an NTFS volume) or on the subfolder level (e.g., an NTFS non-root directory). In this case, all files and directories are sorted by their creation timestamp, either globally (filesystem level) or locally (subfolder level). This ordering defines a (logically) continuous storage space that can be used to write and later read the hidden data.

4.3.1.2 Error Correction Layer

The normal use of the storage medium hosting the filesystem as well as the actions of the attackers may remove some of the files stored on the filesystem. Also, the attackers might intentionally change the creation timestamps of some of the files. Such actions, deliberate or not, cause a new ordering of the creation timestamps, which results in the inability to either access the input file segments in the correct order or altogether.

The error correction layer augments the initial representation of the input file with additional information that can cope with the aforementioned issues. As a first step, an error correcting code (ECC) is appended to the representation. The ECC can both detect and reconstruct missing information. As a second step, this layer enforces a *start offset* for the used files. This allows the program to start from a random point in the ordering and use both older and newer files. Thus, not only old files are used to hide information.

The selection of an appropriate ECC is left to the implementation. By and large, an ECC should not introduce significant storage overhead.

4.3.1.3 Encryption Layer

The error correction layer introduces data redundancy. This redundancy comes on top of the structured information needed to represent the links from timestamp to timestamp in order to form a logically continuous storage space. These can be sources that result in timestamps with patterns. If patterns are detected, the whole steganographic system will collapse, since they reveal the existence of hidden information.

The role of the encryption layer is twofold: On the one hand it protects the hidden information from disclosure. Only somebody in possession of the related cryptographic key(s) can access the encrypted and hidden information. On the other hand the confusion and diffusion properties of a (good) secure cipher ensure that hardly any pattern will exist in the output allowing it to appear totally random.

The encryption layer uses symmetric-key cryptographic algorithms to encrypt the information of the two previous layers. Stream ciphers, as for example AES-OFB or RC4, can be used in this layer. The advantage of stream ciphers over block ciphers is that the former do not need to expand the output of the operation and that they can recover to a certain point from errors (e.g., missing timestamps) at a bit rather than a block (i.e., dozens of bytes) level.

4.3.2 Information Representation: The Case of NTFS

We can now describe how the TOMS components work together to hide a message in the file timestamps. In the following, we will use the NTFS filesystem as an example. However, the description is valid for any other filesystem with similar characteristics. Figure 4.2 depicts the NTFS inode data structure used to represent various filesystem objects, including a file and a directory. In the following, we will use the term “file” to refer to NTFS inodes.

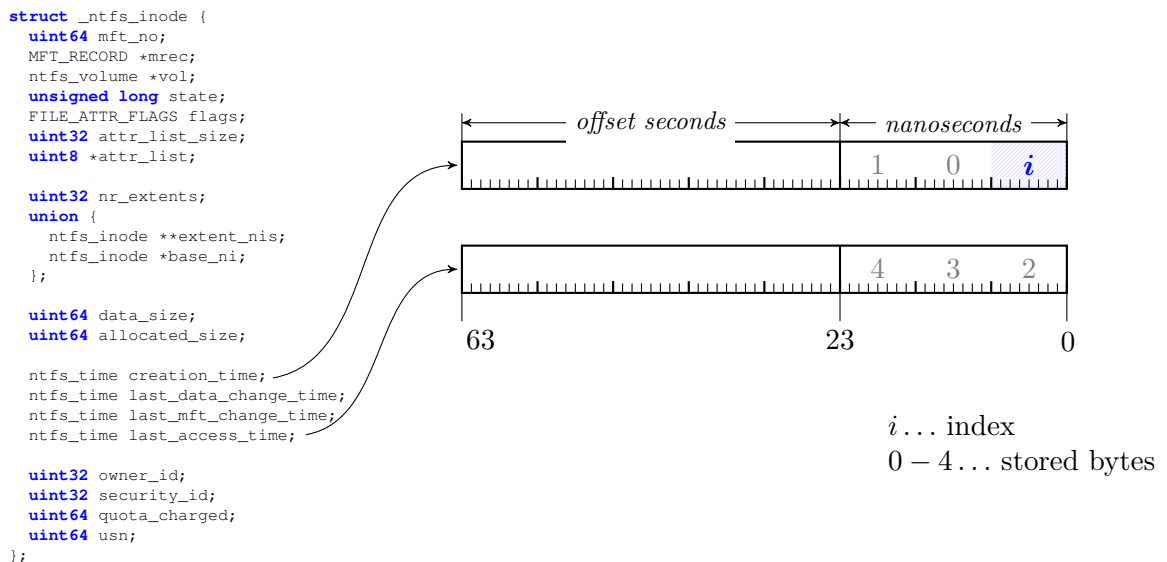


Figure 4.2: Overview of storing data in the nanoseconds part of the timestamp fields.

Two file timestamps can be used by TOMS in the case of NTFS: the *creation* and the *last access*. Each timestamp uses 24 bits to represent its nanoseconds part. Thus, a total of six bytes per file can be used to hide information. This constitutes the elementary storage unit (ESU) for the TOMS system. We assume that the size of the input (steganographic) message, M , is much larger than the size of an ESU. First, an error correcting code function is applied to the input message, $E = ECC(M)$. Then n , the number of ESUs needed to store E , is prepended ($n||E$).

4.3.2.1 Information Hiding

The information hiding process works as follows. The encoded message (E) is fragmented into n blocks of five bytes each (B_1, B_2, \dots, B_n). Then, every block is prepended with one byte that is used as a block index ($i \in \{1, \dots, n\}$). The special value of 0×00 for the index byte is used to prepend a block of five bytes that contains the number of needed ESUs, n . The resulting structure is a linked list of six-byte blocks: $(0, n), (1, B_1), \dots, (n, B_n)$. This structure is then encrypted with a stream cipher and a secret key, producing an output list of six-byte blocks: C_0, C_1, \dots, C_n .

TOMS constructs the list of candidate files that they can be used as carriers. The list, F , is ordered based on the creation timestamp of each file, and a start offset, s , is chosen randomly. The ordered list of files, $F_s = \{f_s, f_{s+1}, \dots, f_{s+n}\} \subseteq F$, will be used as the carrier. TOMS then proceeds and replaces the nanoseconds part of the creation and access timestamps of each file in F_s with the six-byte encrypted chunk C_i .

4.3.2.2 Large Message Handling

Using just one byte as index limits the length of the hidden message (E) to only 255 bytes. We overcome this limitation by allowing multiple index bytes to share the same value (overflow upon reaching the value $0 \times FF$ and restart numbering from 0×01). Whenever an overflow occurs, an ESU is consumed in order to store the length of the whole message, using again an index byte of 0×00 . Thus, every ESU with an index byte holding the value 0×00 contains the *total* length of the message.

4.3.2.3 Recall of Hidden Information

The information recall process works as follows. The timestamps for *all* the files in the filesystem are extracted, sorted by their creation time, and then saved as a list G . For every list entry, the nanoseconds part of the creation and the access timestamps are decrypted by applying the same stream cipher and key material used during the information hiding process. If the decrypted first byte of the creation timestamp equals the index byte value 0×00 , the respective timestamps are added in an offset list L and the number of ESUs, n , is recovered. Then, the next n list entries are processed, recovering the respective index (i.e., $0 \times 01, 0 \times 02, \dots, n$) and the structure H . Next, the error correction code function is applied on H , recovering the original (hidden) message M .

4.4 Evaluation of the TOMS System

In this section, we evaluate the design principles of the TOMS system. Our evaluation is based on theoretical, experimental, and evidence-based analysis of the steganographic strength of TOMS under the assumed threat model (cf. Section 4.3).

4.4.1 Stealthiness

“Stealthiness” describes the degree to which the very existence of hidden information is disguised, irrespectively of the ability to recover the hidden message(s). We analyze the two factors defining the stealthiness of the TOMS system in the following.

4.4.1.1 Timestamp Handling by Operating Systems

The use of the encryption layer ensures that the steganographic information are not recoverable without having access to the key material of the stream cipher used. Furthermore, a good stream cipher ensures that each output bit will be a one or a zero with equal probability. But how do modern operating systems handle the timestamp information in first place? Do they fill these data structures with sub-second-precise information or do they opt for a different strategy? If the former, what is the precision of the provided time information? We sought the answer to these questions using three approaches.

4.4.1.2 Code Audit

As a first approach, we performed a code audit of the NTFS-3g implementation of the NTFS filesystem [150]. This is the default driver for accessing NTFS volumes from within the Linux and Apple OS X operating systems, and it is an open source code. A similar code audit for the NTFS implementation of the Microsoft Windows operating system was beyond our reach, since the source code is not publicly available. The code audit revealed that the NTFS-3g fills the related timestamp structures with nanosecond-granular information provided by the Linux system clock which also has a nanosecond granularity.

4.4.1.3 Synthetic Data

The second approach was to create synthetic data for experimentation, which is online at the authors website together with the most recent version of the code used for this purpose². We created files in batches using a Python script on a Linux system accessing an NTFS volume via NTFS-3g. Each batch created 100,000, one million, or ten million different files. Half of the files were created with a random delay of one to two seconds between each creation. The other half of the files were created with zero delay, i.e., as fast as the computer system could handle the requests.

Our “synthetic” dataset contains 117 million files. We collected the three timestamps (create; access; and modify, all equal to each other) for this dataset as well. We conducted an exploratory data analysis to determine if the timestamp distribution was uniformly distributed. Results for the Kolmogorov-Smirnov goodness-of-fit test for uniformness indicated that the timestamp distribution did not deviate significantly from a uniform distribution ($D = .99, p = 2.2 \times 10^{-16}$).

²<https://www.sba-research.org/dfrws2016/>

The source code audit and the experiments validate that the Linux operating system uniformly uses the full spectrum of the 24-bit sub-second granularity to store the timestamp information.

4.4.1.4 Real Data

The third approach was to collect evidence from Microsoft-Windows-based systems that are actively used to perform day-to-day tasks (“real-world systems”). We therefore collected the file timestamp information from a sample of 70 filesystems (NTFS volumes) from multiple Microsoft Windows computers available at our research lab. On average, each filesystem of our sample contained about 290,000 files and 40,000 directories; the largest one contained over 2.2 million files and directories. The majority of the sampled filesystems ($n=63$) were actually “system volumes”, i.e., they contained the files of the Microsoft Windows operating system (e.g., those files commonly found in the `C:\Windows` directory) and (most likely) of the majority of installed software (e.g., those files commonly found in the `C:\Program Files` directory). Only seven systems contained more than one NTFS volume (i.e., “non-system volumes”). Such volumes are often used as storage for work or personal data (e.g., documents, spreadsheets, and pictures). In total, our “real-world” dataset contains the timestamps of 22,261,386 files and directories.

We analyzed the timestamps contained in this dataset and we noticed some irregularities in their distribution. Creation timestamps that are filled with zeroes in their nanoseconds parts were disproportinally more frequent than expected. This is the case when files are migrated into NTFS volumes from FAT32 filesystems. The latter use a granularity of two seconds at best, hence the zeroes. This assumption was empirically tested and further confirmed by Microsoft’s documentation regarding timestamp changes [101].

4.4.1.5 Time of Filesystem Inspection

In the previous paragraphs, we saw that the timestamps can be used as stealthy information carriers, since the sub-second information follows a uniform distribution, as does the output of a stream cipher encryption. Before replacing any timestamps, one should consider if and how often the filesystem is inspected by an attacker. As an example, we consider the case of (operating) system files. These files are installed once and are seldom, if ever, touched again (e.g., only when operating system updates are installed). Thus, if their timestamps are proactively collected, any future modification by the TOMS system will be easily detected.

In a forensics analysis scenario, we can assume that the investigator will inspect the metadata *after* the message was hidden in the timestamps. We can also assume that the investigator does not have access to earlier versions of the filesystem metadata information. Thus, existing timestamps can be utilized to hide steganographic information. In a scenario where the filesystem can be proactively inspected, already existing files might

not be good candidates for carriers. Thus, only new files (i.e., generated after the last inspection) can be utilized to hide steganographic information.

We assume for our subsequent analysis a more conservative scenario, in which the filesystem is *proactively* inspected. In this case, one should exclude all system files and all files with timestamps containing zeroes in the sub-second part. Applying this to our initial real-world dataset, it resulted in an almost 50% drop of available files, down to 11 million files spanning 70 NTFS volumes.

4.4.2 Robustness

“Robustness” refers to the ability of the TOMS system to cope with changes in the filesystem contents. The information hiding and recall procedure of TOMS is straightforward when the initial ordered list of files F_s remains intact between information hiding and information recall(s). In the following, we analyze how the TOMS system defends against actions that result in modifications of F_s .

4.4.2.1 Encrypted Metadata

This is the simplest of the the proposed storage container layers. The ordered list of files F_s is not affected by operations on the filesystem level (assuming that these operations do not touch the timestamps). Should some files have been removed from the filesystem, or some timestamps are updated, the encryption and the error correction layers may be able to recover the lost information thanks to the use of the stream cipher and the ECC. If and how the recovered information is stored back to the timestamps (e.g., insert new files, re-encode information, or even fix the “corrupted” timestamps) is a decision to be made taking into account the severity of the errors and the assumed time and frequency of inspection.

4.4.2.2 Oblivious Replacements

In this approach the ordered file list F_s results from sorting the timestamps that are provided by the filesystem. Thus, it might be the case that the TOMS system unknowingly uses a different list F'_s for information recall instead of the one originally used for information hiding. If some files were removed between information hiding and recall(s), the same arguments as in Section 4.4.2.1 apply.

We assume that some additional files, F_a , are included in the $F'_s = F_s \cup F_a$ list, and that the computing system has a proper clock. If oblivious replacement is applied globally (filesystem level), the TOMS system will always recover the correct F_s . This is feasible because the file creation timestamp is immutable on an NTFS volume, i.e., it does not change when the file is copied, renamed, or moved *within* the same NTFS volume [23]. Thus, even if files are moved across different NTFS folders, their creation timestamp will not change. Also, these additional files F_a will have more recent creation timestamps than those already contained in the original F_s and allow therefore a clear separation of the two sets.

If oblivious replacement is applied locally (subfolder level), then it is possible that the ordering of F_a is intermixed with the ordering of F_s . This is the case where files are moved to the specific subfolder containing F_s ; as mentioned earlier, the file creation timestamps are immutable. This situation is accommodated by the use of the encryption and the error correction layers. Assume that a file $f_m \in F_a$ is inserted in the ordered list F_s . First, the ESUs of f_m must decrypt correctly and not be rejected by the encryption layer. Then, the value of the index byte contained in the (erroneously) decrypted ESUs must match the currently expected sequence number in order to not be rejected by the error correction layer. Finally, the payload information contained in the ESUs must pass through the ECC. Only then, these information are accepted as valid. If the processing of f_m fails, the error correction layer provides the necessary protection to recover from the error. Thus, the two layers provide an adequate defense (up to a certain point) against such (deliberate or not) insertion attacks. The amount of redundant information handled by the ECC defines this protection level. An oblivious replacement at the subfolder level requires a stronger ECC compared to the filesystem level.

4.4.3 Applicability

“Applicability” refers to the degree to which the TOMS system can be utilized in various application scenarios. The layered design of TOMS provides an initial indication of its wide applicability. The TOMS system supports three different storage layers and is agnostic to the ECC used as well as to the stream cipher. Furthermore, TOMS can be easily applied to any modern filesystem that supports sub-second timestamp granularity; while the basic description supports two timestamps often found in modern filesystems (namely, *creation* and *last access*), there is no design constraint regarding the number of timestamps or the use of filesystem-specific timestamps (e.g., *last attribute modification* for `ext4`). The design of the TOMS system allows to explore various performance tradeoffs in order to match the secrecy requirements of the selected application. We discuss these tradeoffs in the following paragraphs.

The application scenario defines the use of existing files or opts to create new ones to embed a steganographic message. In the latter case, it is advisable to generate small files that act as carriers (e.g., files in the range of few thousands bytes).

The use of an ECC introduces an overhead of 10-20%. If the risk of information loss can be sustained, the use of an ECC can be omitted altogether.

The selection of the storage container type is important. If an encrypted metadata file is used, one must decide if the contents of the file should be embedded in the filesystem or stored elsewhere. The resulting size of this metadata file can be a decisive factor. When embedding about 1.5 MB of data into 175,000 timestamps, the corresponding metadata file takes about 215 KB of disk space. A benefit of this approach is that there is no need to store index bytes to rebuild the ordered list of carrier files and recover the hidden information. Also, file reordering is not a threat in this case (unless someone is tampering

with the metadata file) and thus the performance requirements for an ECC are more relaxed (or can be omitted altogether).

The oblivious replacement approach mandates the use of index bytes. Each ESU uses one index byte per five payload bytes (ratio of 1:5). If only one timestamp is available for each file, the ratio becomes 1:2, which may cause a lot of overhead. On the other hand, if three timestamps are available, this ratio becomes 1:8, which is quite efficient. Compared to an encrypted metadata approach, oblivious replacement needs between 12.5% (two timestamps) and 20% (three timestamps) more files in order to store the same amount of hidden information.

4.5 Experimental System Validation

We developed a proof-of-concept implementation of the TOMS system for the experimental validation of our steganographic proposal. The implementation targets the NTFS filesystem and is based on the Python language version 2.7 for flexibility and increased portability. Our implementation can be delivered as a stand-alone executable and does not require the installation of special software or any modifications of the Linux kernel. It realizes the layered design described in Section 4.3 and can be easily ported to work with any filesystem that uses a nanosecond timestamp granularity.

The development and experimentation platforms are based on Xubuntu Linux 15.04 64-bit, running kernel version 3.19.0-25, the latest stable one at the time of writing. The underlying disk on which the operating system is installed is a solid state disk (SSD) for faster I/O access. As NTFS is Microsoft-proprietary, we opt for NTFS-3g in its current version. The steganographic executable application takes care of all information management tasks. The application is assumed to have full access to the NTFS volume (filesystem).

The application supports the use of two- and three-file timestamps. The file *creation* and *last access* timestamps are not modified by the operating system: starting with Microsoft Windows Vista, the default value of `NtfsDisableLastAccessUpdate` is set to one [65]. The corresponding mount option in Linux is `noatime`; in most of the popular Linux distributions this option is not activated by default. The file *last modified* may be modified under normal use, so it is up to the users to decide if they enable it (and pay attention not to destroy the related information during the normal use of the filesystem).

4.5.1 Information Hiding and Recall

The typical work flow for information hiding is as follows: the user starts the Python application and provides (i) the message to be hidden, (ii) a key to encrypt the message, (iii) the method for hiding (metadata file, oblivious replacements on volumes, oblivious replacements on subfolders), and (iv) the number of different timestamps to use (either two: *creation* and *access* or three: *creation*, *access*, and *modification*). Once the necessary

Listing 4.1: Embedding data in timestamps.

```

1 def hide(path, msg, key):
2     rs= calcReedSolomon(msg)
3     m= msg · rs
4     C= chunk(m, 5)
5     index= 0
6     temp= ∅
7     for c ∈ C:
8         s= c
9         if index = 0 or index % 255 = 0:
10            s= length(m)
11            temp= temp · index · s
12            index++
13     files=sort(recEnumFiles(path),by=creation_time)
14     offset= calcRandomOffset()
15     while offset:
16         files.pop()
17     em= encrypt(temp, key, mode=RC4)
18     C= chunk(em, 6)
19     for c ∈ C:
20         f= files.pop()
21         f.creation_time.nsec= c[0:3]
22         f.access_time.nsec= c[3:6]

```

information are collected, the application performs the following steps: (i) it concatenates the message with the error correction code, (ii) adds the index bytes to the resulting data (if the chosen hiding method is not the metadata file), (iii) encrypts the data with the stream cipher, and (iv) embeds the encrypted data into the timestamps. On the information recall path, the user enters the encryption key and the application displays the decrypted message.

Listing 4.1 and Listing 4.2 outline in pseudocode the two work flows for information hiding and recall respectively.

4.5.2 Metadata File Information Protection

All information processed by the application are held in memory (RAM) and are encrypted with AES-256-CBC using a user-provided key. This is a precautionary measure in order to protect against extraction of the plain metadata file during a forensics analysis of the storage medium, e.g., in the slackspace of the hard disk [105].

After the information has been embedded, the metadata file is built from the information kept in RAM so far. Before writing this information to the disk, it is compressed using gzip and encrypted with the AES algorithm using a user-provided password. Our implementation supports the use of different passwords for the metadata file and the actual data.

Listing 4.2: Extracting data from timestamps.

```

1 def extract(path, key):
2     F= sort(recEnumFiles(path), by=creation_time)
3     em= {}
4     for f in F:
5         c= f.creation_time.nsec * f.access_time.nsec
6         em= em * c
7     m= decrypt(em, key, mode=RC4)
8     C= chunk(m, 6)
9     i, l= 0,0
10    for (i, c) in enumerate(C):
11        if c[0] != 0x00:
12            continue
13        l= int(c[1:6])
14        break
15    S= sort(C[i:l], by=first_byte)
16    temp= {}
17    for (i, c) in enumerate(S):
18        t= c[1:6]
19        if c[0] != i:
20            t= 0x00 * 5
21        temp= temp * t
22    return decodeReedSolomon(temp)

```

We take care not to accidentally write the unencrypted metadata file to the disk, as this could leave persistent traces which particular files were modified. During the embedding process the information resides unencrypted in the RAM, and we did not implement countermeasures to prevent the operating system to store the corresponding memory pages on the disk, e.g., due to paging or hibernation. However, our application supports the encryption of information on a per-path basis, right after embedding the information in order to minimize the time the unencrypted information resides in the RAM, at the cost of creating a much larger metadata file due to the lack of compression.

4.5.3 Performance

Two of the main considerations of steganographic systems are the undetectability and the confidentiality of the hidden data [103]. The performance of the system is also an important factor with respect to applicability.

We performed a series of experiments to gain insights regarding the performance of TOMS when embedding and extracting information using volume-wide oblivious replacement. Table 4.2 summarizes our findings. The reported figures are the averages of ten consecutive executions of hiding (embedding) and recall (extracting). The amount of space used to embed data is reported as a percentage of the overall available storage space provided by the ESUs (i.e., 6 bytes per file). The time needed to hide (embed) the information is almost constant, irrespective of the data volume. On the other hand the time to recall

Table 4.2: Time to embed and extract information on filesystem.

Space used	15%	30%	50%
Timestamps needed	78,687	157,325	264,193
Time to embed [sec]	74.78	76.17	76.33
Time to extract [sec]	20.19	36.92	60.29

(extract) the information is almost linear to the percentage of embedded data. In both cases the time ranges in dozens of seconds, which might be considered too high. Upon closer inspection, it appears that the calculation of the Reed-Solomon ECC dominates the processing time for both. However, since the file metadata are extracted from the MFT, which resides in the RAM, the average time to extract is lower than the average time to embed. This lower time is caused by performing the vast majority of filesystem operations within the RAM instead of directly accessing the hard disk.

4.5.4 Effect on Actual Filesystem Operation

As a final consideration, we examined if the filesystem remained operational for normal use after manipulating the stored file timestamps. We mounted and unmounted the NTFS volumes that were modified by our proof-of-concept implementation using the drivers provided by Linux, Microsoft Windows, and Apple OS X operating systems. We did not notice any problems in using the volumes, and no error messages were logged by the operating systems. We also performed regular file operations in the volumes and did not notice any issues. Recall of the steganographic information after the regular use succeeded without any problems as well.

The analysis validates our initial assumption: typical usage scenarios of modern filesystems allow to persistently store additional information in file timestamps without affecting their normal use.

4.6 Steganographic Capabilities of TOMS

4.6.1 Methodology and Datasets

We designed three experiments to study the characteristics of the TOMS steganographic channel. We derived one dataset per experiment: a synthetic (artificially-generated) dataset, a consumer-grade dataset contributed by individual volunteers, and an enterprise-grade contributed by a collaborating company. We describe in the following the experiments and the collected information in greater detail.

4.6.1.1 Synthetic Dataset

We used the same laptop computer running Microsoft Windows 10 64-bit build 1607 for all the steps described below. The laptop computer has an Intel i3 with 2.1 GHz and 4 GB RAM.

Storage Media: We used five different storage media, namely a mechanical, spindle disk (HDD), a solid-state disk (SSD), two external hard disks (E1 and E2), and a USB flash drive (U1). The HDD is manufactured by Hitachi (model number: HTS541010G9SA00). It has a storage capacity of 100 GB and spins with 5,400 rpm (revolutions per minute). The SSD is manufactured by Micron (model number: MTFDDAK256MAY). It has a storage capacity of 256 GB. Both E1 and E2 disks drives were mechanical, spindle disks hosted in a separate case. E1 is manufactured by Western Digital (model number: WD2500I032-001). It has a storage capacity of 240 GB and spins with 5,400 rpm. An IDE connector is used to mount the disk inside the case. E2 is manufactured by Seagate (model number: STDR2000200). It has a storage capacity of 2 TB and spins with 5,400 rpm. A SATA connector is used to mount inside the case. Finally, U1 is manufactured by Kingston (model number: DTR30G2, Datatraveler). It has a storage capacity of 16 GB.

Connectors and Interfaces: The two internal disks (HDD and SSD) were connected over a SATA bus. The two external disks were connected over a USB 2.0 (E1) and a USB 3.0 (E2) interface. A USB 3.0 port was used to connect the USB flash drive. We also used the HDD in a different setup (eS), connected to the system through an external SATA (eSATA) connector.

Manual File Copy from External Sources: After the initial setup of the operating system, several files were delivered using a USB flash drive. Some of those files were then installed to provide a basis for scripted file generation described in Section 4.6.1.1. The exact sequence was as follows:

1. Copy the Python 3.5 .msi installer to the disk.
2. Copy the Python 2.7 .exe installer to the disk.
3. Copy the Python and the Powershell file creation script to the disk.
4. Copy the timestamp derivation application to the disk.
5. Install Python 2.7.
6. Install Python 3.5.
7. Execute the file creation script.
8. Execute the file timestamp extraction application.
9. Collect the log files for further analysis.

Scripted File Generation: We used two scripts to generate files³, one based on Python and one based on Microsoft Powershell. The Python script utilized the default packages, such as “os”. The Microsoft Powershell script relied on no external packages.

Both scripts accept as an input parameters the path to create the files (e.g., C:\tmp); the number of files to create; and, optionally, a delay between each file creation. For the latter script, we suppressed the default behavior of printing the created files in the standard output (stdout), for performance (speed) reasons and for compliance with the behavior of the former script.

Automated File Generation Process: We proceeded with the file generation as follows. First, we formatted the storage media and created an NTFS volume (filesystem) on them. Then, we did a fresh install of the operating system from a readily-available Microsoft Windows 10 image available on a spare external USB drive. We repeated the same procedure at the start of each experiment described below. This approach ensured a common starting point for all the experiments and a ground truth for the files of the operating system.

The format step in the beginning is a necessary step to ensure that the MFT is reset and all experiments start from the same point. Indeed, the MFT is a special file on NTFS-based filesystems. It contains a record for every file and directory ever contained in the filesystem. As such, a growing-in-size MFT impacts the read and write time of the files, due to the longer access time to the MFT. Hence, the more files are created by its iteration of script execution, the longer the access time becomes.

The following list summarizes the experiments we ran:

- One set of 100,000 files created with the Python script on the HDD and SSD with a random (uniform) delay of 0.1 to 1.0 second between each file creation. The same using the Microsoft Powershell script.
- Two sets of 100,000 files each created with the Python script on the HDD, SSD, E1, E2, U1, and eS with no delay between each file creation, i.e., as fast as possible. The same using the Microsoft Powershell script.

File Timestamp Extraction: We developed a Python script based on the stat subpackage of os to extract the file timestamps from each file on the filesystem. We extracted the creation (C), last accessed (A), and last modified (M) timestamps of each file and directory with a granularity of 100 nanoseconds. The outputs of the script were saved on an external disk for later processing. Once logs were collected, the storage media under test was wiped out, formatted, and the process concluded.

³The scripts are available at <https://www.sba-research.org/ares2017hiccups/>

4.6.1.2 Consumer Dataset

The second dataset was derived by four individuals that volunteered to participate in our study through personal invitations. We explained to the volunteers the aim of our study and shared with them the source code of the Python script used to extract the file timestamps for review before getting their consent. To protect the innocent, the script parsed through the SHA3 hashing algorithm both the filename and its path; the information collected are the hash output and the three timestamps (C, A, and M) for each file.

In the first round, the volunteers executed the timestamp extraction Python script on their personal computers and laptops running a version of the Microsoft Windows operating system and hosting at least one NTFS filesystem. We label these partitions as *consumer-grade*. The partitions span diverse uses, such as playing computer games, IT freelancing, leisure activities (e.g., Internet surfing and movie watching), backup storage for valuable personal information (e.g., photographs and long documents), and mobile computing.

The script outputs were written in text files and the volunteers shared back these files. The volunteers contributed timestamp information for ten NTFS partitions and 2.5 million files in total.

In the second round, we contacted the volunteers again after ten weeks and asked them to perform the same steps again. All the participants responded within one week. The combined logs provided information for 2.6 million files in total, i.e., a 100,000 increase in the number of files.

4.6.1.3 Enterprise Dataset

The third dataset was contributed by a collaborating company that agreed to participate in our study. The company policy dictates common rules for each computer (e.g., automated installation of licensed software packages) and centralized administration by authorized personnel.

Using the same script as in the case of the individual volunteers, the company personnel responsible for its IT infrastructure extracted and shared with us the timestamps for more than 22 million files of 70 different computers and NTFS partitions.

In this enterprise environment, we assume that the file timestamps for a large number of files across different computers will be the same, as the files come from the same installation media.

4.6.2 Analysis

Our analysis is based on the three types of datasets we collected (synthetic, consumer, and enterprise). We focus on the following parameters: effect of the underlying storage

Table 4.3: Average and standard deviation of the occurrences of unique timestamps in the synthetic dataset (delayed creation)

Storage	Python	Powershell
HDD	1.00 (0.02)	1.51 (0.62)
SSD	1.00 (0.02)	1.55 (0.68)

technology and creation techniques, effect of the regular use of the filesystem, and effect on large-scale installations.

For the sake of readability, hereafter we use the term “timestamp” to refer to the 24-bit sub-second part of each file’s 64-bit “full timestamp”. Recall that NTFS stores this parts of information with a granularity of 100 nanoseconds. Hence, there are in principle 10 million unique *timestamps*.

4.6.2.1 Storage Technology and Scripted Creation

We analyzed the extracted creation (C) timestamps for the six storage media used to produce our synthetic dataset. Since the files are not accessed or modified afterwards, we do not discuss in the following the access (A) and last-modified (M) timestamps. They are the same as the creation timestamp.

The first step relates to the delayed creation of 100,000 files in the two internal disks (HDD and SSD). It took one minute and 35 seconds for the Python script to create all these files, and 31 seconds less for the Powershell script. Table 4.3 summarizes the average and standard deviation number of occurrences of each unique timestamp included in this part of the synthetic dataset.

In the case of the Python script on the HDD, the distribution is quite uniform (average 1.0) and there are 100,000 unique full timestamps. In the case of the Powershell script on the HDD, the situation is slightly different. There are multiple repeating timestamps and only 66,397 unique full timestamps. This number is significantly low. There are even more than 4,300 cases where three files share the exact same full timestamp.

The situation is similar in the case of the SSD. The distribution is quite uniform for Python and all the 100,000 files have a unique full timestamp. Again, the Powershell script results in repeating timestamps and there are only 64,668 unique full timestamps. There are even more than 6,800 cases where three files share the exact same *full timestamp*.

We conclude from the above that the storage media and connectors do not affect the timestamp distribution. However, it appears that the scripting language used to generate the files *does* affect the produced timestamps.

The second step relates to the creation of two sets of 200,000 files in each of the six available setups described in Section 4.6.1.1. In general, the creation of the files was faster using Powershell rather than Python, as summarized in Table 4.4. It took a Powershell

Table 4.4: Time in minutes to create a set 200,000 files for the synthetic dataset (no delay)

Storage	Python	Powershell
HDD	1:35	1:04
SSD	1:34	1:01
E1	0:59	1:01
E2	1:01	1:00
U1	0:59	0:24
eS	0:58	0:57

close to one minute to generate the 200,000 files of each set. This is quite an interesting observation: it is almost the same time needed in the first step to generate the 100,000 files. Python exhibited the same performance as before for both HDD and SSD. It exhibits similar performance to Powershell in the case of the externally-connected disks E1, E2, and eS. The latter is also worth-mentioning, as the storage media is the same as HDD; changing just the connector from internal SATA to external eSATA, it takes 33% less time to create the files. The case of U1 is also interesting in that for Python it takes the same time (about one minute) like the other externally-connected. However, it takes less than half a minute for Powershell to create the files on it.

We proceed with an analysis of the 400,000 creation timestamps for each of the six storage media used to produce our synthetic dataset. Table 4.5 summarizes the average and standard deviation number of occurrences of each unique timestamp included in this part of the synthetic dataset. The situation now is quite different compared to that summarized in Table 4.3. The Powershell script results in an average number of occurrences that is close to 2.0 and a standard deviation less than 0.6. On the other hand, both the average and the standard deviation of occurrences is significantly high in the case of the Python script, ranging from about 3.5 for the USB flash drive (U1) to 31 for the hard drive connected over an eSATA connector (eS). We also observe that the same device has an average of almost 13 when connected through the internal SATA interface (row “HDD”). Figures 4.3-4.6 visualize these striking differences in the distribution of timestamps for the four cases of HDD and eS using Python and Powershell.

The aforementioned information suggests that an analysis of the distribution of the (sub-second) creation timestamp part may reveal both the scripting language and the storage media type and connector used when the files were created. This can be useful to detect if a set of files were originally created on the disk under investigation or were transferred to it through other means (e.g., copy from another media, which might disclose a data leakage).

Table 4.5: Average and standard deviation of the occurrences of unique timestamps in the synthetic dataset (creation without delay)

Storage	Python	Powershell
HDD	12.95 (2.09)	1.99 (0.37)
SSD	13.52 (1.62)	1.97 (0.39)
E1	3.93 (2.04)	1.80 (0.50)
E2	17.31 (2.80)	1.68 (0.56)
U1	3.46 (0.94)	1.50 (0.52)
eS	31.00 (6.13)	2.15 (0.52)

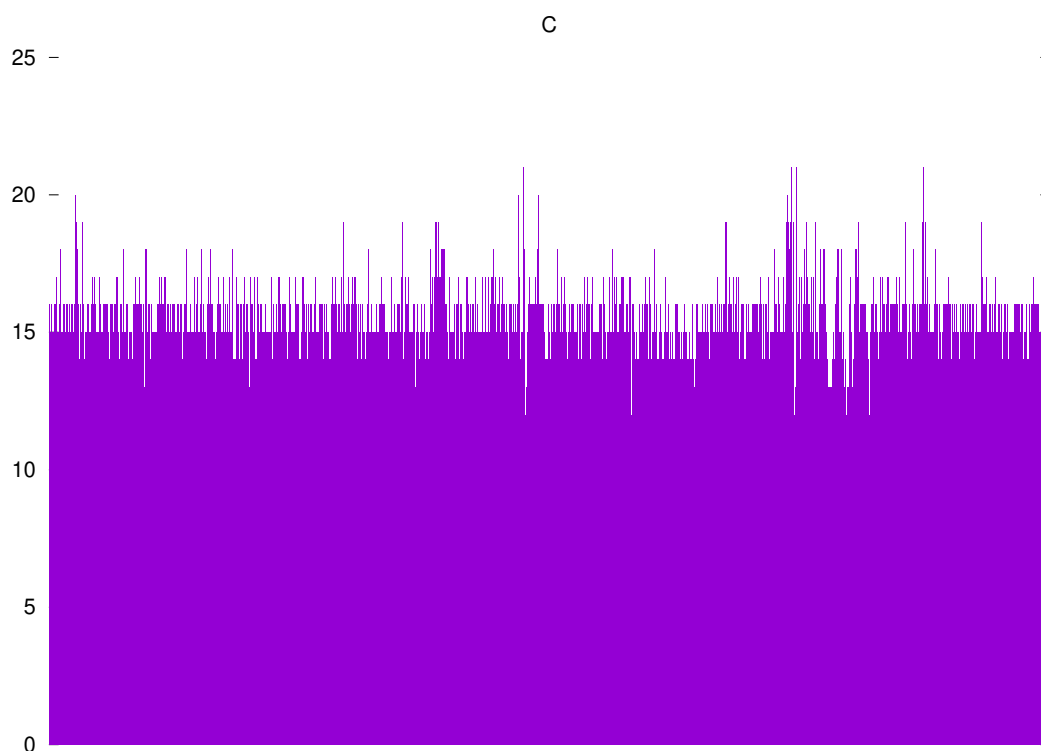


Figure 4.3: Creation time (C), Synthetic, HDD, no delay, Python

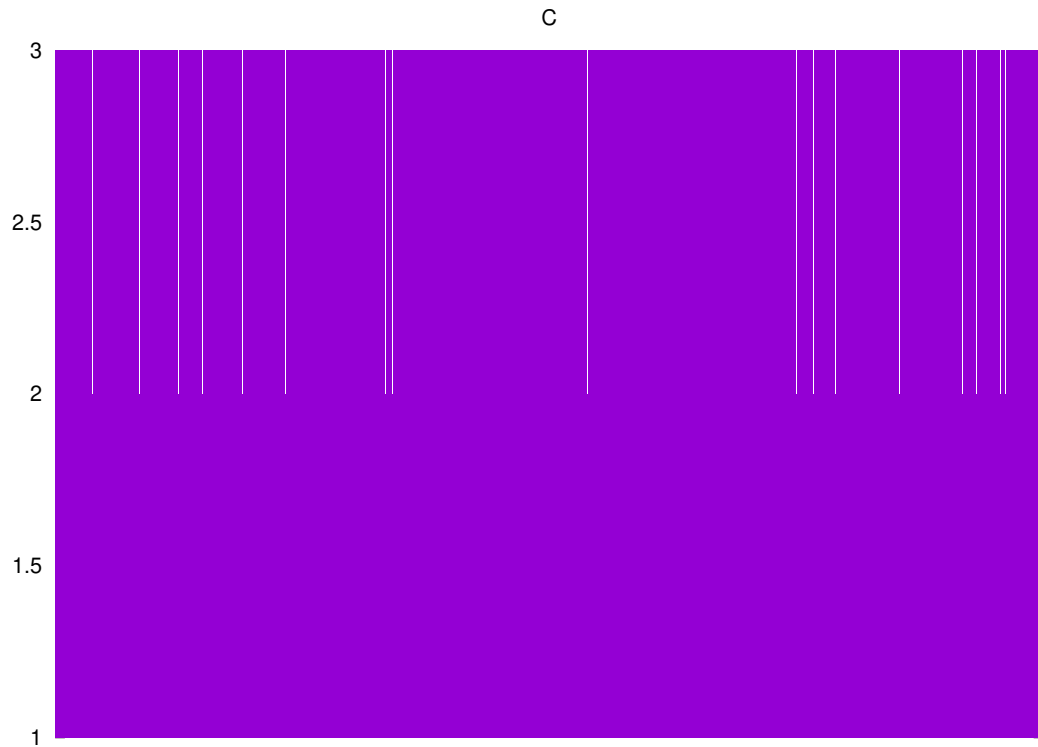


Figure 4.4: Creation time (C), Synthetic, HDD, no delay, Powershell

4.6.2.2 Regular Use of the Filesystem

The first part of our analysis, described in Section 4.6.2.1, justified that it is feasible to generate a large number of files in a short time. This does allow the fast creation of a steganographic channel to hide information.

In the second part of the analysis, we study the effect of the day-to-day use of a filesystem on the timestamps. We use the consumer-grade dataset for this analysis. There is information for ten NTFS volumes (filesystems).

Table 4.6 summarizes the number of unique create (C) and last-access (A) timestamps observed (C-bins and A-bins respectively) in each volume of the consumer dataset at the beginning of the experiment (Round 1) and after ten weeks (Round 2). There are some interesting observations to further discuss in the next paragraphs.

A first observation is that there is a *thirtyfold* difference in size among the ten volumes in the number of C-bins, as demonstrated in the case of A2 and T2 in Round 2. This indicates different usage patterns for the volumes (e.g., storing only the operating system files and using it as a working space).

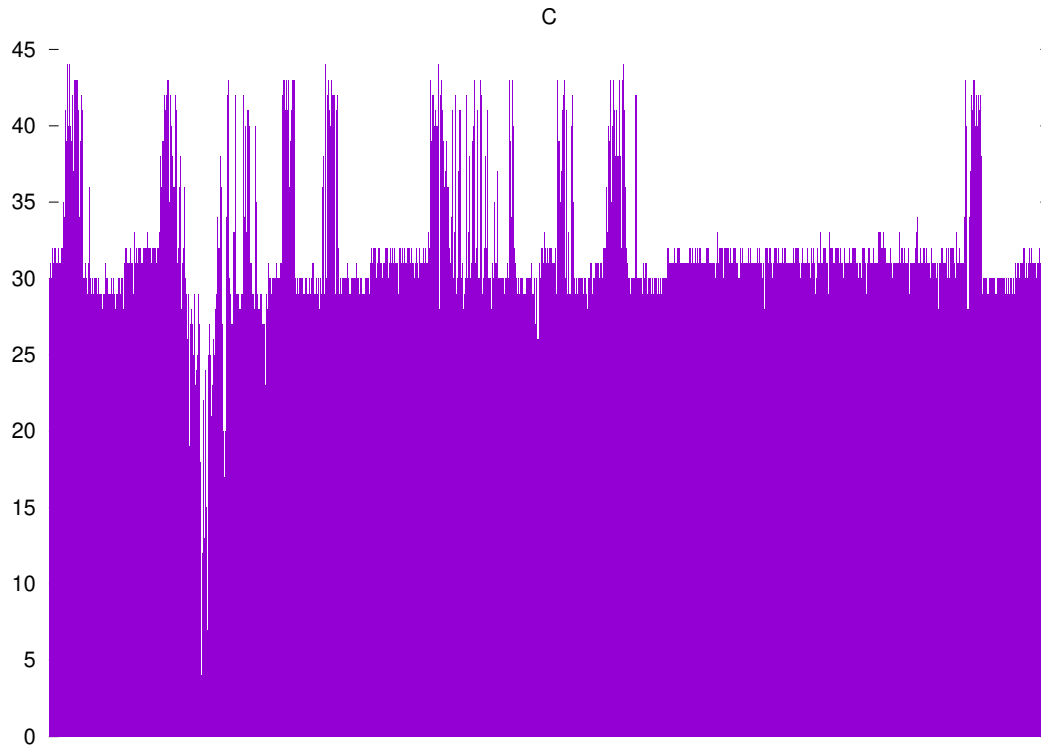


Figure 4.5: Creation time (C), Synthetic, eSATA, no delay, Python

Table 4.6: Number of unique timestamps (bins) of files in the consumer-grade dataset filesystems

id	Round 1		Round 2	
	C-bins	A-bins	C-bins	A-bins
S1	66,146	83,907	74,897	92,662
S2	57,920	92,405	104,538	68,224
T1	67,787	103,290	76,182	113,017
T2	132,149	138,917	625,753	766,984
T3	127,086	135,902	127,086	135,902
T4	270,206	317,427	270,273	317,503
M1	385,088	384,961	292,000	317,906
M2	206,461	287,538	194,620	297,097
A1	184,367	207,896	88,782	113,356
A2	-	-	20,745	21,811

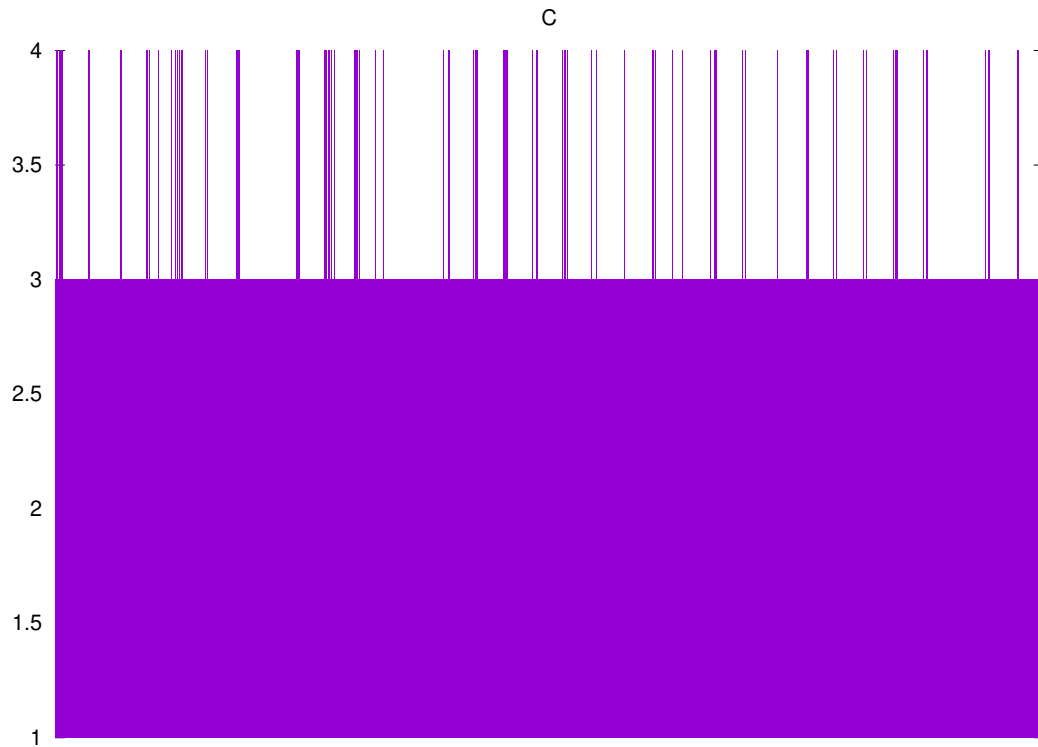


Figure 4.6: Creation time (C), Synthetic, eSATA, no delay, Powershell

A second observation is that number of unique timestamps heavily fluctuated between the two rounds of the experiment. There are NTFS volumes that remain almost intact (e.g., T3 and T4), volumes that rapidly expand (e.g., S2 and T2), or shrink (e.g., M1 and A1).

A third observation is that in all but two cases (M1 in Round 1 and S2 in Round 2), there are more A-bins than C-bins. It appears that many files share the same creation timestamp but their access times are modified later on. This is further supported by the evidence provided in Table 4.7. The average number of occurrences for the C-bins is greater than the one of A-bins for all cases. As the total number of creation and last-access timestamps are equal, this is an expected finding. However, the standard deviation is quite bigger than the average value. This is an indication that the distributions deviate from a uniform one (even for the reduced number of available bins) and there might be a significant number of outliers.

Our assumption is valid, as depicted in Figure 4.7 and Figure 4.8. The figures depict the histogram of C- and A-bins for two NTFS volumes, namely S2 and A1. There are a lot of outlier values, reaching even 900 occurrences. Similar trends are observed in all volumes of our dataset. They are not reported here due to space limitations. One of the

Table 4.7: Average and standard deviation of the occurrences of unique timestamps in the consumer-grade dataset (aggregated)

id	Round 1		Round 2	
	C-bins	A-bins	C-bins	A-bins
S	3.80 (17.44)	2.73 (9.53)	3.60 (8.96)	2.56 (8.96)
T	2.87 (37.81)	2.40 (7.53)	2.21 (56.91)	1.80 (10.33)
M	2.36 (59.62)	2.08 (37.66)	2.47 (64.04)	1.96 (33.09)
A	2.02 (5.29)	1.79 (3.58)	3.07 (34.43)	2.45 (4.75)

volunteers kindly agreed to share the file names and paths of the whole NTFS volume (identified as S1) for the purpose of this thesis.

Our analysis indicates that a large number of files are part of software installations on the volume (e.g., an office productivity suite and a computer game). Such files are installed from compressed archives and/or DVD/CD-ROM media. As such, the original timestamps are preserved when copied to the NTFS volume. The original media do not support 100-nanosecond granularity⁴, the “hiccups” in the histogram hence. These hiccups do not invalidate the TOMS steganographic channel. Rather, an attacker must carefully select a subset of files that exhibit a smooth, uniform distribution and avoid specific paths that come from installation media (e.g., the `C:\Program Files\` folder). Another approach would be to create a new set of files altogether for hiding information in their timestamps.

4.6.2.3 Enterprise Environment

In the third and last part of the analysis, we study the effect of a homogeneous environment on the distribution of the file timestamps. We use the enterprise-grade dataset for this analysis. There are information for 70 NTFS volumes.

Our analysis revealed that a large number of files are, as in the case of the consumer-grade dataset, part of software installations from compressed archives and UDF volumes. There are stronger patterns now, as these installations are instrumented from a central location and from the same installation media.

Figures 4.9-4.18 depict the distribution of the create (C) and last-access (A) timestamps for ten randomly-selected volumes; all the 70 volumes exhibit the same patterns and are not reported here due to space constraints. The “hiccups” are evident once more but a TOMS-based attack is again possible.

We note that in the case of an enterprise environment, the attackers have a harder task to solve, as the channel capacity is further reduced. Indeed, the IT administrators

⁴The UDF filesystem supports microsecond granularity (Source: <http://www.osta.org/specs/pdf/udf260.pdf>).

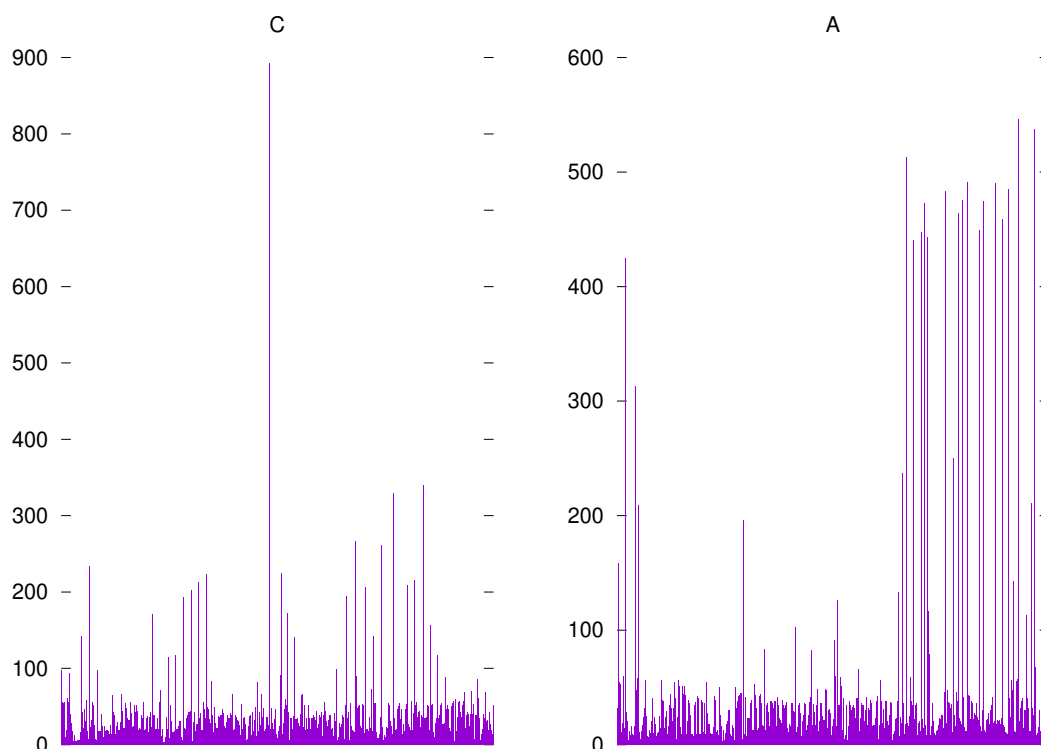


Figure 4.7: Histogram of C-bins and A-bins for S2

can compare the distribution of timestamps of an investigated volume with many more available in the enterprise - this is not possible in the case of single consumer-grade volume as there is no “reference” volume available to compare against. Furthermore, company policies can reduce the number of volumes and paths where an attacker can place the files for the TOMS channel, further minimizing (but not eliminating) the risk of an attack.

4.7 Implications for Forensics Analysis

Responsible research in steganography involves both developing new techniques for information hiding and detection (steganalysis). The issue of detection is increasingly important for digital forensics examiners, as criminal activities through digital means are becoming prevalent [44].

Embedding information in file timestamps is feasible. As discussed in Section 4.4, these information are indistinguishable from that of normal operations, provided that a stream cipher is used in the encryption layer. Thus, a statistical analysis of file timestamps should be incorporated in the forensics examination procedures as a first line of defence.

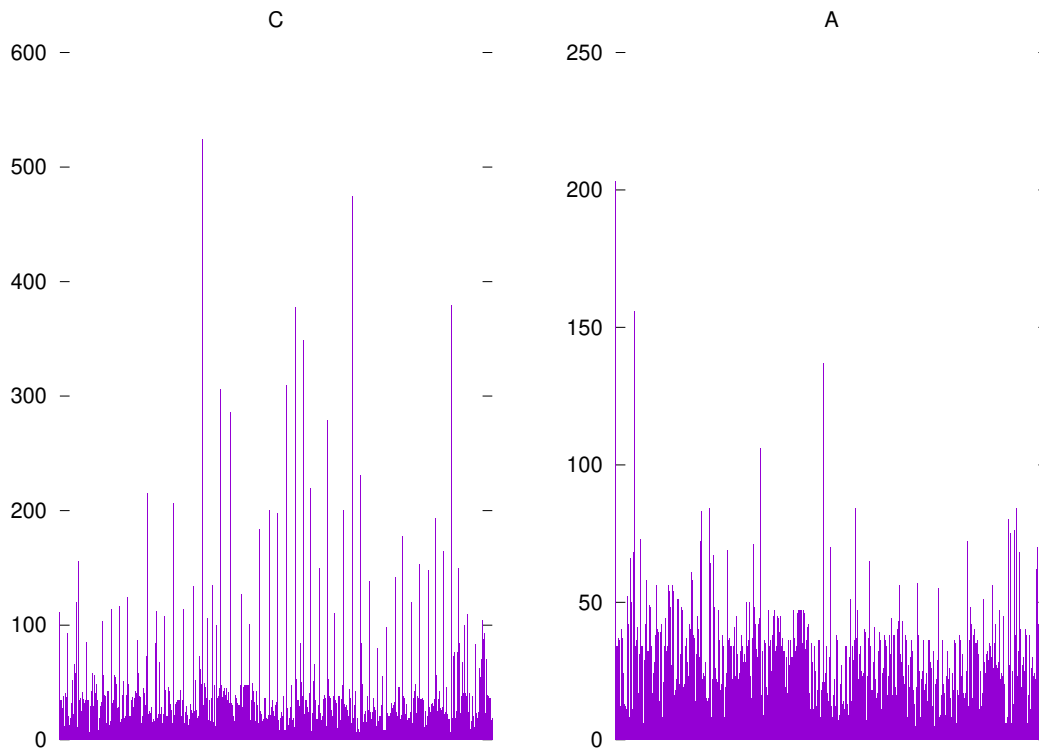


Figure 4.8: Histogram of C-bins and A-bins for A1

This analysis can provide hints for the presence of information hidden in the timestamps, if one opts to disable the encryption layer of the TOMS system.

There may be additional artefacts and implementation details which can assist a forensics investigator in disclosing the presence of hidden information. A fully-functional TOMS demands careful implementation and operation decisions. The developer must ensure that the application leaves no installation or execution traces that could reveal its presence. If a backup image of the filesystem contents at an earlier time is available to the examiners, they can compare the timestamps regarding unjustifiable modifications, especially for the case of the creation timestamps. Furthermore, the generation of new files to use them as carriers must be justifiable from a *modus operandi* point of view. For example, if modification timestamps are utilized, it must be justifiable why they differ from the creation timestamps – this would be suspicious if it occurs in the same second for a large batch of files. It is advantageous to check for *modus operandi* violations during the forensics examination process.

Another approach for the investigation procedure is the correlation of an installation timeline for an operating system and its well-known application files (e.g., the Microsoft Office suite). If such files are used to hide information and if they share the same

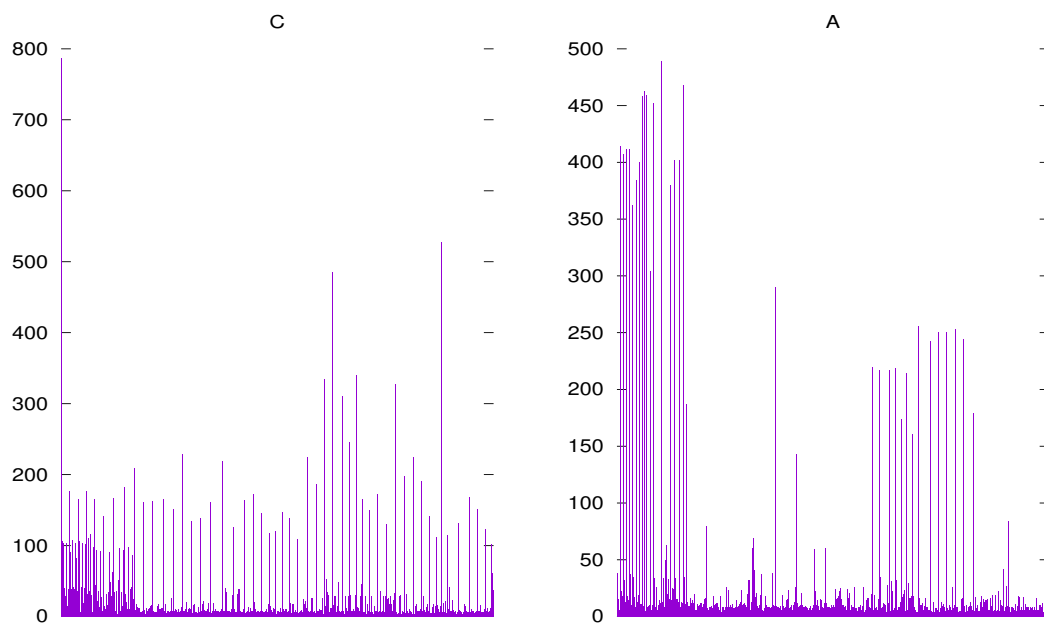


Figure 4.9: Histogram of C- and A-bins in enterprise volume 9

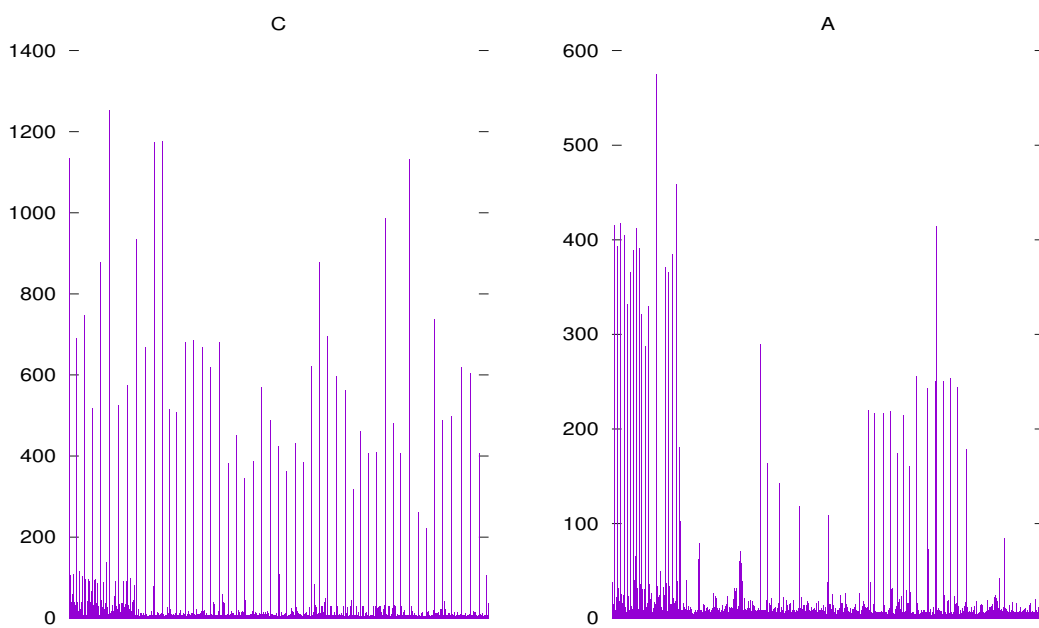


Figure 4.10: Histogram of C- and A-bins in enterprise volume 15

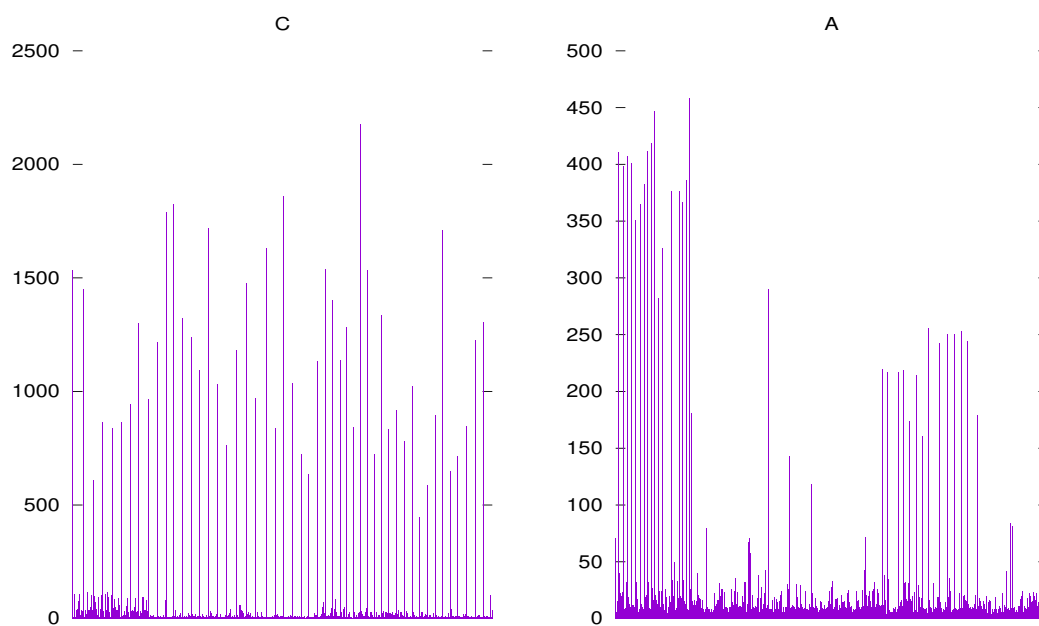


Figure 4.11: Histogram of C- and A-bins in enterprise volume 18

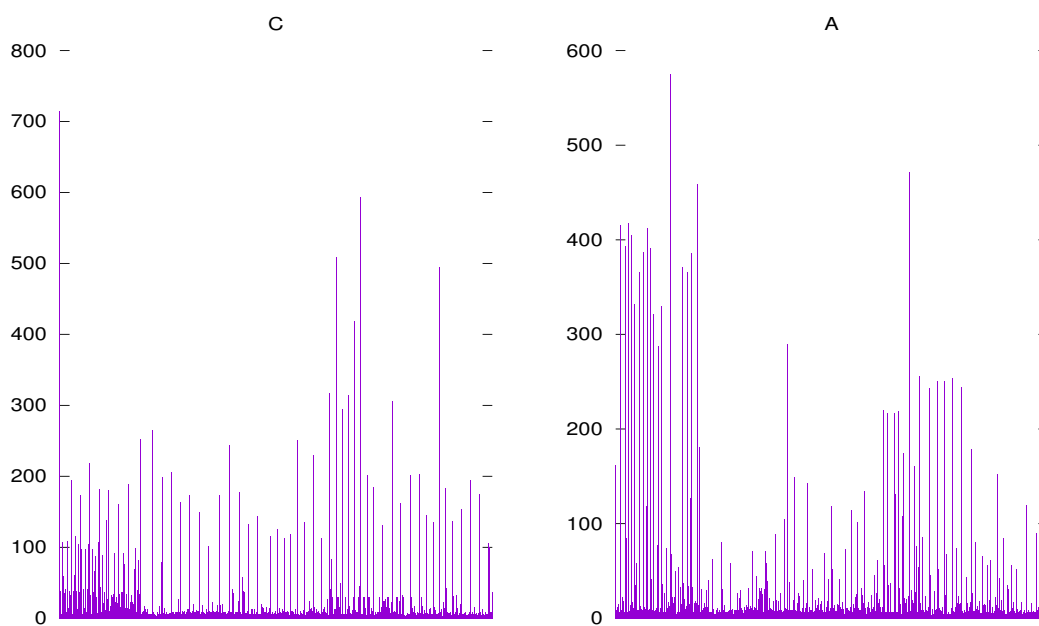


Figure 4.12: Histogram of C- and A-bins in enterprise volume 40

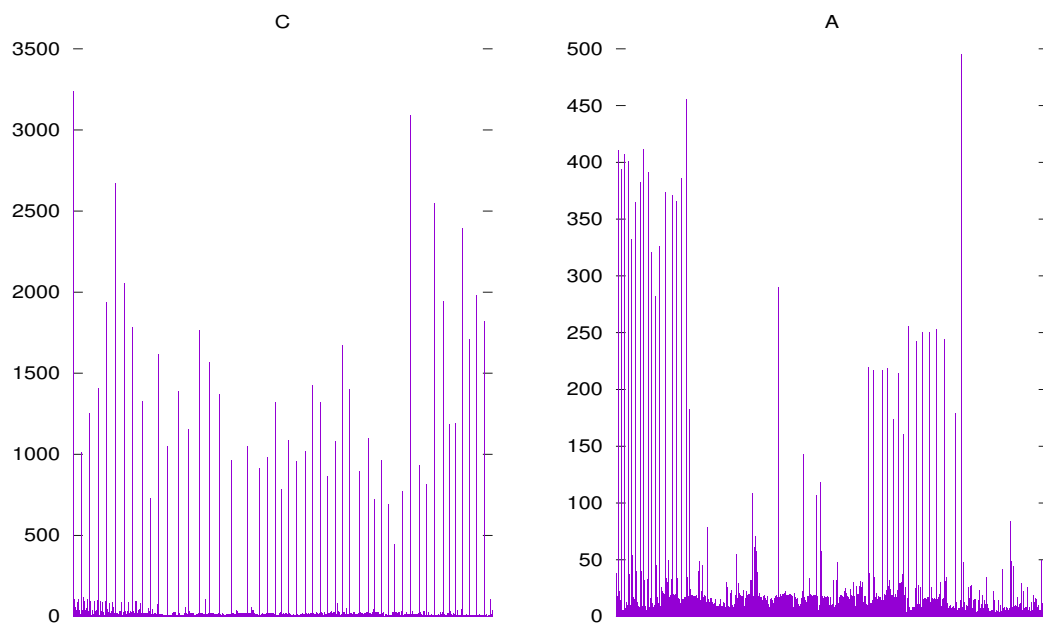


Figure 4.13: Histogram of C- and A-bins in enterprise volume 42

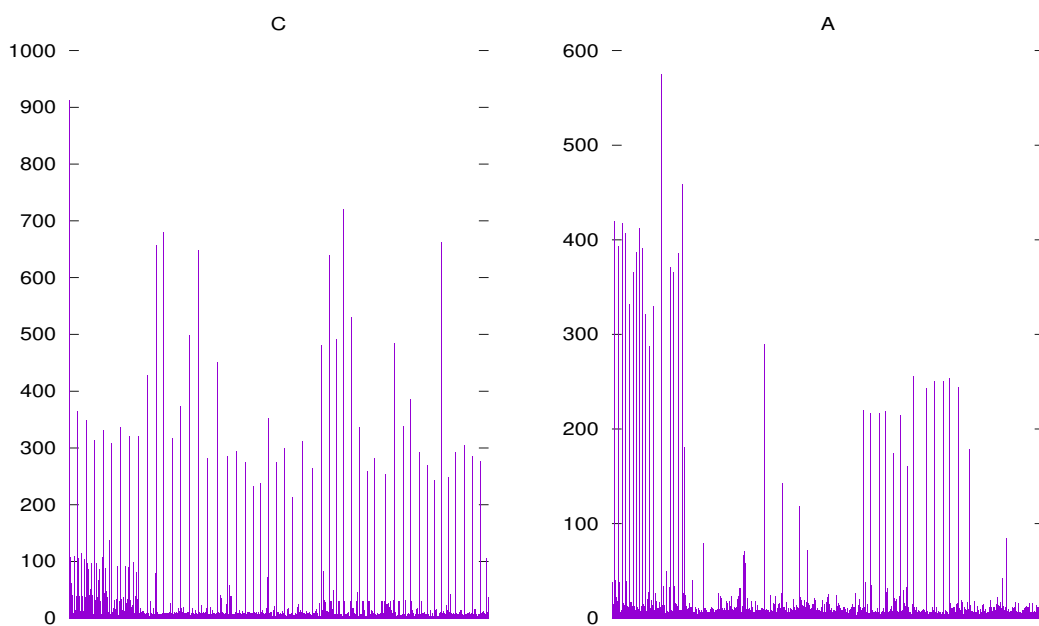


Figure 4.14: Histogram of C- and A-bins in enterprise volume 45

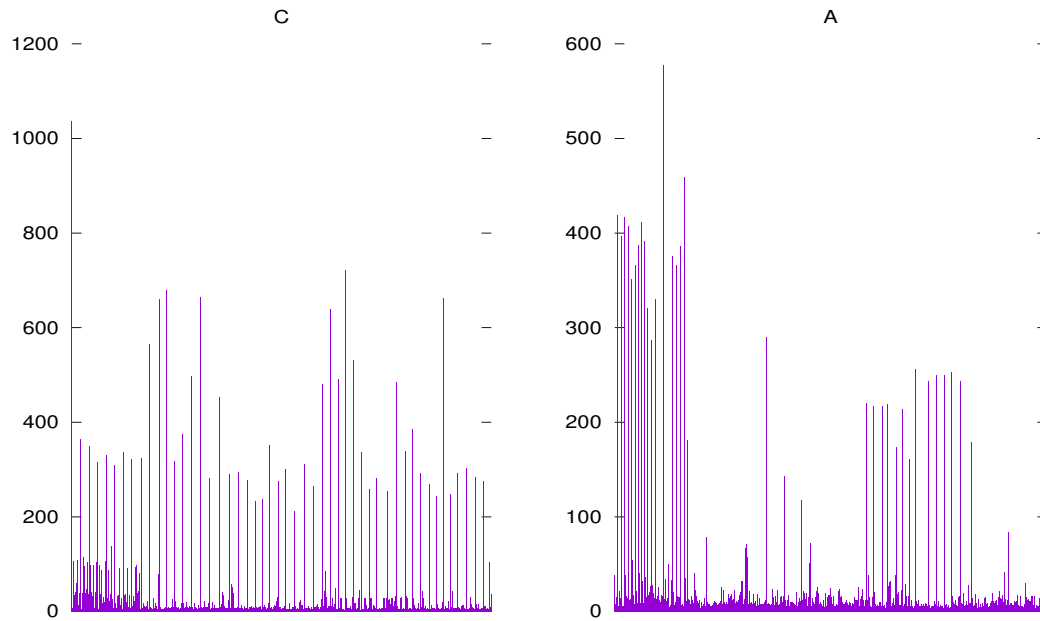


Figure 4.15: Histogram of C- and A-bins in enterprise volume 48

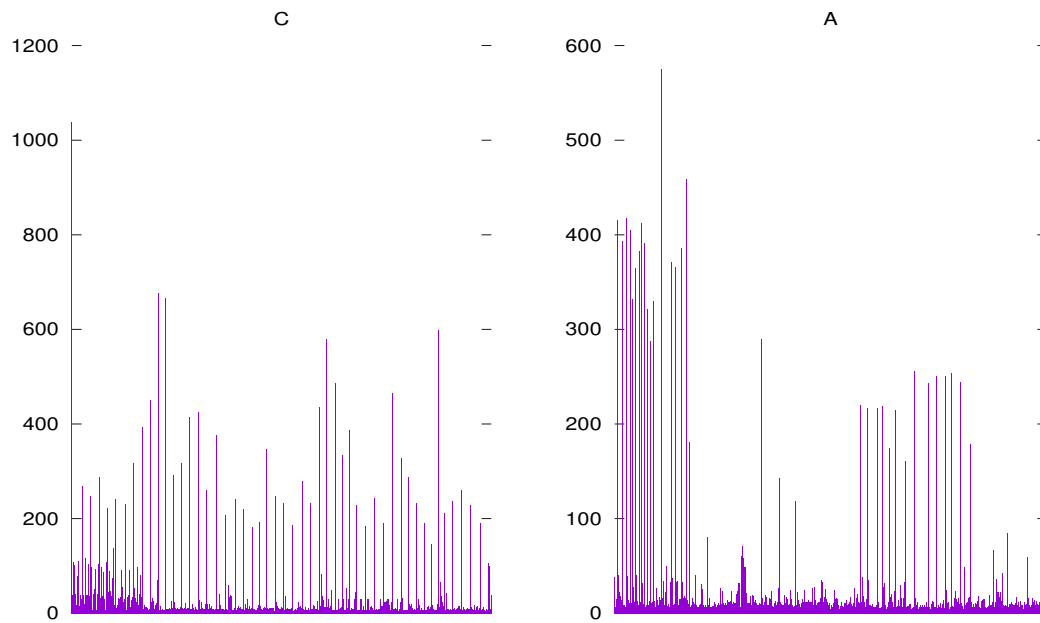


Figure 4.16: Histogram of C- and A-bins in enterprise volume 60

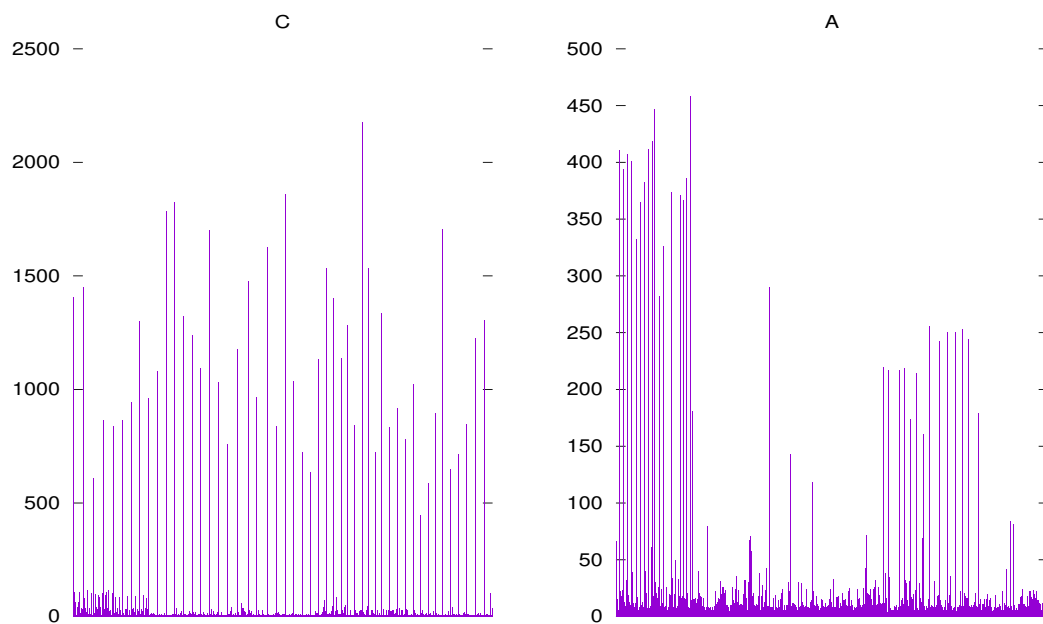


Figure 4.17: Histogram of C- and A-bins in enterprise volume 65

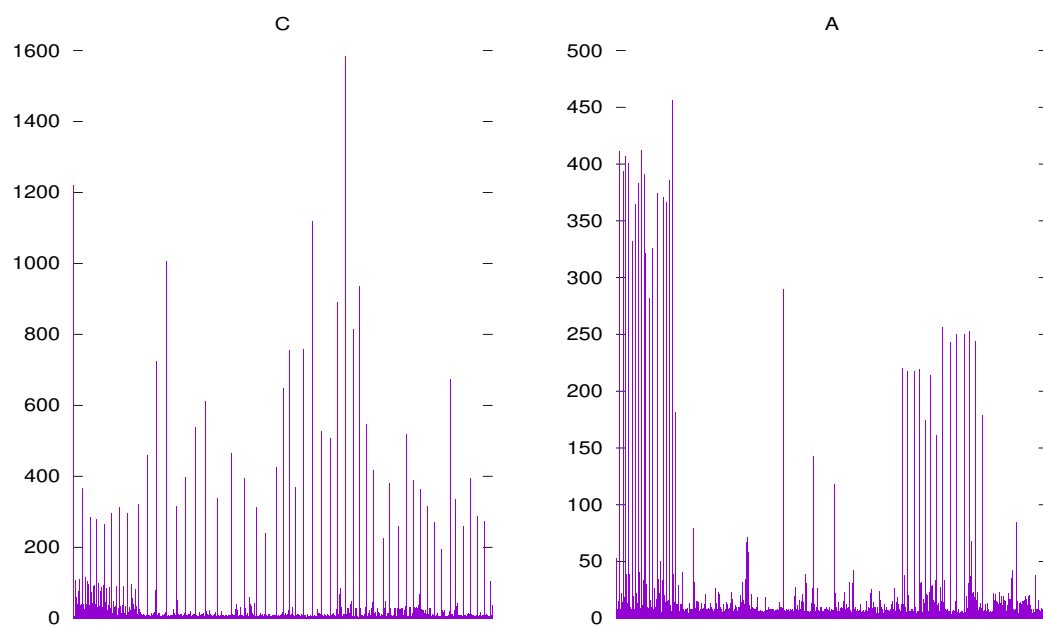


Figure 4.18: Histogram of C- and A-bins in enterprise volume 67

timestamp up to the second part, an installation timeline can reveal that the creation order of some files does not match the expected one.

The filesystem data structures are not the only place where timestamps are stored. If an operating system records file-related events in its system logs with nanosecond precision, a digital forensics investigator can perform a correlation analysis between these two information sources in order to detect unjustifiable mismatches [32, 40, 41]. Operating systems also use transaction logs (journals) for recovery processes (e.g., the NTFS Transaction Log `$LogFile` [31]). These can also be used for correlation analysis. Wiping out such log files or carelessly modifying them can raise further suspicions and assist aiding the investigation along.

In the case of NTFS, an informed decision in our proof-of-concept (PoC) implementation was to modify *only* the filename attribute of the MFT. However, the same information are maintained in the standard information attribute as well. Thus, an investigator can compare the two attributes and detect the use of the PoC implementation.

4.8 Conclusions

In this chapter, we proposed and explored the applicability of file timestamps as a steganographic channel. Based on our analysis of how modern operating systems store timestamps for file events in filesystem data structures and how they are displayed to the users, we reveal a redundant space to hide information. We described how this space can be utilized as a steganographic channel using a layered design that offers stealthiness, robustness, and wide applicability. We evaluated our design through theoretical, evidence-based, and experimental analysis in the case of the NTFS filesystem with datasets containing millions of files: the hidden information are statistically indistinguishable from timestamps produced during normal use. We also validated the applicability of our proposal through a proof-of-concept implementation targeting the NTFS filesystem. We furthermore discussed the implications of this new steganographic technique for digital forensics analysis.

Additionally, we assessed the feasibility of building such channels using different storage media and connectors and evaluated the attainable channel capacity in artificial (synthetic), consumer, and enterprise environments.

We confirm that TOMS is a feasible threat. However, the attacker has to spend significant effort to hide the manipulations from a proper digital forensic investigation, either by disabling the TOMS encryption layer or by choosing those files that exhibit a uniform distribution of timestamps already. The capacity of the steganographic channel is further reduced in an enterprise environment with centralized administration, where different NTFS volumes can be compared with each other for irregularities in timestamp patterns.

CHAPTER 5

System-level Detection of Keystroke Injection Attacks

As attackers are proceeding with their attacks to lower levels of the computer system architecture, also this thesis is. Therefore, in this chapter we provide insights on how kernel-level USB-based attacks work and explore how their attack patterns can be detected at the system level, without involving the user into the trust decision.

The Universal Serial Bus (USB) is by far the most widely-used connector for modern computer systems. It is used to connect a plethora of peripheral devices to computers, including keyboards, mice, cameras, printers, and storage media. Many different attack vectors abuse the pervasiveness of USB, as for example dropping USB thumb drives on parking lots for users to pick up and attach on their computers [149]. As network-based defenses steadily improve and can block efficiently the malicious network traffic reaching an organization, USB becomes an attractive entry point for penetrating an organization.

Under the hood, USB is more than a simple connector. It is a complex *communication protocol*, often implemented and offered as a firmware. Lately, there are devices on the market with the ability to update their USB firmware. This capability has been exploited as a subtle attack vector, hiding malicious functionality on an abstraction layer that modern computer antivirus cannot cope with. BadUSB¹ and Rubber Ducky² classes of attacks are successful demonstrations of the attack feasibility. The associated threat is rather high: at this level the device interfaces directly with device drivers that run in the most privileged level of modern consumer-grade operating systems (e.g., as ring-0 modules of the Linux kernel).

¹<https://srlabs.de/badusb/>

²<https://github.com/hak5darren/USB-Rubber-Ducky>

We make the following contributions:

- We study the behavioral characteristics of Rubber Ducky and BadUSB classes of attacks.
- We devise criteria for automating their detection.
- We design, implement, and evaluate a simple yet *very effective and extensible* system-level countermeasure based on USB packet traffic analysis to detect and defend against such attacks *without* requiring user intervention.
- We explore how network-wide analysis tools for monitoring the spread of USB devices across an enterprise network can further enhance the detection of attacks and the incident response efforts.

5.1 Background

5.1.1 The USB protocol

USB is the most widely-used computer peripheral connector today. USB 3.2 is its latest revision. The USB device communication is based on a tiered-star topology with one dedicated master controller. Besides the controller, a hub manages the connected USB devices. If the master controller acts as a hub too, then the hub is called the “root hub”. Every USB hub uses seven bits to address connected USB devices. This leads to a limit of 127 attachable USB devices per hub.

The connection of a USB device to a hub works as follows³: The USB hub waits for new devices to be plugged in. Upon connection, channels for communication are created: the so-called “endpoints”, acting as sources and sinks of data. The endpoints are logically grouped together to “interfaces” and are announced to the host via “interface descriptors”.

The USB communication is realized by exchanging “USB packets” over the shared serial bus. The USB protocol defines four transfer types (Control, Isochronous, Interrupt, and Bulk) and three packet types (Token, Data, and Status).

Modern operating systems, including Microsoft Windows, Linux, and Apple Mac OS, utilize the information collected by the hub from the connected USB devices to (dynamically) load the appropriate device drivers. For each announced interface descriptor of a device, the operating system combines the device-provided device class, interface class, and vendor and product identifiers (VID and PID respectively) to decide which capabilities are provided by the device and to bind the appropriate device driver(s).

As an example, a modern USB mouse may offer the capabilities of a human interface device (HID) and those of a display (e.g., to display its sensitivity level). Or a USB headset may offer the capabilities of an audio output device and that of a constrained

³<http://www.beyondlogic.org/usbnutshell/usb3.shtml>

HID for its volume up/down and mute buttons. In such cases, the devices will have two different interfaces and each of them will get a different driver bound to it.

5.1.2 USB Protocol Security

The USB protocol does not dictate a form of device authentication. Rather, every USB hub *blindly trusts* any information announced by the connected device about their capabilities. We note that modern USB devices incorporate, for legitimate reasons, multiple functionalities (e.g., a mouse announcing itself also as a display device). Such functionalities are hard for a user to link together and reason for any associated risk. These combined with the wide prevalence of USB devices, render USB an attractive attack vector [5, 155].

In the past, the entry barrier for realizing attacks based on the inherent weaknesses of the USB protocol was very high. It dropped significantly by the time USB firmware chips with reflashing capabilities became available on the open market⁴. On the one hand, firmware updates for consumer products are often a necessity due to shortened time-to-market and insufficient testing. The alternative would be a product recall which would cause a logistics nightmare. On the other hand, firmware updates significantly lower the resources and expertise for launching USB-based attacks.

Figure 5.1 depicts the principle of a USB device and endpoint setups, which occur during the Control transfer phase using Setup packets. In this example, the device announces support for two functionalities, namely a mass storage and a keyboard (HID device). The former seems like a normal behavior, assuming that the user plugged in a USB thumb drive or external disk. In this case, the user expects that the operating system (host) will load the mass storage device driver and be able to further interact with the storage device. However, we note that without having knowledge of the device specifics, this announcement could also have been an attack vector.

The latter announcement forces the host to bind a keyboard driver as well. If the announcement comes from a modified, malicious firmware, then the first step of the attack is already successful. The firmware then launches the second step of its attack. This involves sending keypresses from the (non-existent) keyboard. This “USB-based keypress injection attack” assumes that the system interaction caused by these keypresses will go unnoticed by the user and, thus, will succeed to deliver the malicious actions (e.g., download from the Internet or access from the USB storage and then execute a zero-day malware).

So far, various attacks exploiting the inherent weaknesses of the USB protocol have been proposed [118]. We review the most characteristic of them in the following paragraphs.

⁴<https://adamcaudill.com/2014/10/02/making-badusb-work-for-you-derbycon/>

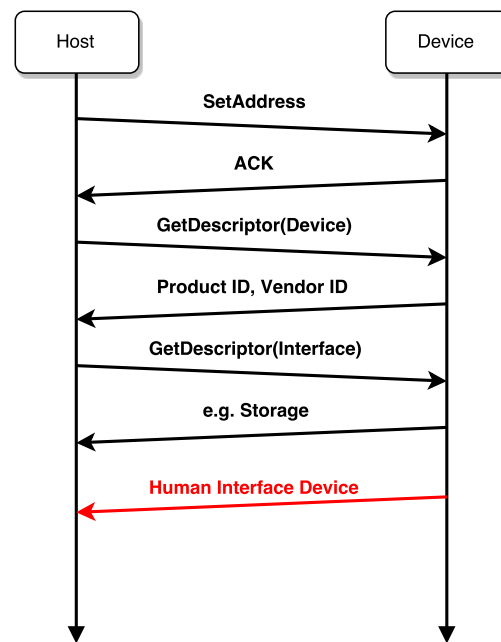


Figure 5.1: USB packet sequence diagram (malicious behavior)

5.1.3 BadUSB and IRON-HID

The BadUSB attack enables a USB storage device to act not only as a SCSI device (mass storage), but also as an HID one. By acting as a keyboard, the data coming from the connected USB device is interpreted as keypresses. An attack can install a backdoor to the host system or “call home” over the network for example. From that point on, the attacker has total control over the infected host system.

Researchers and practitioners work both on improving BadUSB-like attacks and reducing their attack surfaces. In the so-called “IRON-HID” attack, additional programmable hardware (e.g., a Teensy board⁵) is hidden in places like keyboards and portable USB batteries [63]. By connecting a smartphone to the crafted USB battery via a crafted USB On-The-Go cable, the smartphone is switched into USB host mode. From that moment on, the smartphone is able to eavesdrop on all USB communications. IRON-HID can be used also to inject fake keypresses with the aim to brute-force the Android screen unlock PIN.

5.1.4 Rubber Ducky

The *Rubber Ducky* is a physical device designed by the Hak5 group⁶. The Rubber Ducky works as a normal keyboard when it comes to driver binding: it simply needs the

⁵<https://www.pjrc.com/teensy/>

⁶<https://hak5.org>

operating system's HID driver to work. However, the Rubber Ducky delivers USB-based payloads (keypresses pre-defined by an attacker) upon being connected to the victim system.

The pre-defined keypresses are written in Ducky Script, a simple-to-use scripting language⁷. Once the payload is developed, it is compiled into a binary and placed on the microSD card of the Rubber Ducky device. Upon connecting the Rubber Ducky to a host computer, the built-in Atmel AT32UC3B1256 chip of Rubber Ducky emulates the pre-defined keypresses in the *fastest* rate the USB port can deliver and the device driver of the attacked system can handle.

5.1.5 BadAndroid

BadAndroid⁸, like BadUSB, adds malicious functionality to an otherwise benign Android device. In contrast with BadUSB, the firmware of the Android device needs not be flashed.

A possible attack scenario using BadAndroid looks like the following: using social engineering, an attacker pretends that she needs to charge the battery of her Android smartphone and asks to plug it in to the target's laptop. While the smartphone is connected for charging, BadAndroid actually alters the routing table of the host (laptop) system without the user noticing, i.e., it changes the default network gateway of the laptop to be the IP address of the Android smartphone. From that moment on, all the network traffic of the laptop is routed via the smartphone, enabling the attacker to inspect and alter the whole bi-directional network traffic.

In a second attack scenario, BadAndroid could change the entries for the laptop's DNS servers and, therefore, redirect the laptop's traffic to servers controlled by the attacker.

5.1.6 BadBIOS

A maliciously-crafted BIOS hidden on the USB device could be installed on the computer by emulating keypresses at boot time⁹. This BadBIOS overrides the original BIOS and becomes the default BIOS to boot from. This allows an attacker to execute commands even before the actual operating system is loaded.

The applicability of BadBIOS is demonstrated¹⁰ for modern Smart TVs. In this case, a Smart TV is forced to produce high-frequency audio signals. These signals contain information which is then transmitted to other devices.

⁷<https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Duckyscript>

⁸<https://opensource.srlabs.de/projects/badusb/wiki/BadAndroid>

⁹<https://srlabs.de/wp-content/uploads/2014/11/SRLabs-BadUSB-Pacsec-v2.pdf>

¹⁰<https://nakedsecurity.sophos.com/2015/11/16/badbios-is-back-this-time-on-your-tv/>

5.1.7 Other Attacks

The published literature includes USB-based attacks that exploit weaknesses beyond the ones spawned by BadUSB. The *teensy* USB development board¹¹ is designed to instrument and audit USB drivers, ports, and related software. This board is reprogrammable and can be used to launch attacks that are based on emulating keypresses and mouse movements.

The current version of the *USBdriveby* attack tool¹² targets Apple iOS devices. Once successful, USBdriveby alters the routing entries of the attacked system so as to redirect traffic to spoofed websites. The tool also installs a backdoor for the case the user detects the route modifications and changes them back to the legitimate ones.

5.1.8 Defenses Against USB-based Attacks

There have been proposals in the literature for defending against USB-based attacks, especially after the Stuxnet malware, which spread via infected USB drives and penetrated air-gapped systems [83, 62]. A first attempt towards more organizational security of USB device security is described in [160]. There, a trust management scheme, namely TMSUI, is proposed. TMSUI protects an ICS by allowing the connection of USB storage devices *only* on certain protected terminals and *only* for a specific amount of time.

ProvUSB is an architecture for fine-grained provenance collection and tracking on smart USB devices [147]. ProvUSB aims at environments where the use of pre-approved-only USB drives can be controlled and enforced.

UScramBle is a proposal for protecting against eavesdropping attacks that are feasible due to the broadcast nature of (pre-USB 3.0) hub-to-device communication [110]. It can be used to defend against reverse engineering of legitimate devices.

A line of defense against BadUSB-like attacks incorporates the user into the trust decision. One example is USBWall [72]. USBWall uses a Beagle Board¹³ in order to enumerate USB devices on behalf of the operating system. As soon as the enumeration is carried out, the user is asked to decide whether the USB device must be removed from the system or is safe for further use. USBCheckIn is a hardware-based approach, where the user is forced to actively interact with the HID using guided patterns so as to authorize its use [60].

The “G DATA USB Keyboard Guard” software¹⁴ is another system that relies on the users’ decision whether a USB device is malicious or benign. There, when a new HID device is connected to the protected computer, the software asks the user to decide if the device interface(s) are to be trusted or not. Once a decision is made, the device (in fact, the combination of product and vendor identifiers) is either whitelisted or blacklisted so as to avoid asking again in the future. If an attacker flashes a malicious device to

¹¹<https://www.pjrc.com/teensy/>

¹²<http://samy.pl/usbdiveby/>

¹³<https://beagleboard.org>

¹⁴<https://www.gdatasoftware.com/en-usb-keyboard-guard>

present with a previously whitelisted combination of product and vendor identifiers (e.g., a legitimate keyboard), then their attack will go unnoticed.

GoodUSB is a similar approach described in [146]. GoodUSB includes a Linux kernel module that maps USB devices to specific whitelisted drivers. Upon connection of a new USB device, GoodUSB involves the user to decide if it should allow or deny the new device. If the user marks the device as malicious, the control is transferred to a virtualized USB honeypot running on QEMU-KVM. This allows to monitor and profile the activity of the USB device for further analysis.

USBFILTER is a packet-level filter (firewall) for USB communications developed for the Linux kernel space [148]. The user defines access rules in `USBTables`, the userland component of USBFILTER and the kernel-space component checks each USB packet received for match with one of the rules and decides to either forward or drop it. By design, USBFILTER supports only per-packet processing. Given the simplicity of the supported rules, attackers can evade the rules by adjusting their behavior accordingly; overall USBFILTER is a *deterministic* solution that detects already known attacks and does not have any anomaly detection capabilities [118].

Cinch is an approach similar in principles to USBFILTER [9]. However, Cinch isolates all USB devices from the host and passes communication through a virtual machine acting as a gateway that enforces the access policies.

SandUSB offers a GUI for the users to mark a newly plugged-in device as malicious or benign [91]. Should the users consider the device being malicious, they can either blacklist it or redirect it to a USB sandbox for further analysis.

5.2 A Novel Approach for Detection of USB-based Keypress Injection Attacks

We consider a keypress injection attack as the most severe USB-based threat to system security. This is because a carefully-crafted attack, launched through innocent-looking keypresses, leverages the powerful resources and flexibility of the host system so as to take over the full control of the system itself.

The proposed defenses against keypress injection attacks have in common that they rely at some point on user decisions. The user insights in such low-level system trust decisions is not an optimal solution. This is especially true when such interactions break their mental model for the primary task at hand, so as to cope with a secondary one [42, 47]. For example, when users try to help and connect a visitor's USB thumb drive to print a file, one should not expect them to pay any attention to notifications and questions about an extra device; this is just an obstacle that blocks or delays them from their primary task.

In the following, we explore how we can detect and defend against USB-based keypress injection attacks by performing USB packet traffic analysis at the system level, *without*

involving the user in the decision loop. Our aim is to simplify attack detection and offer neutralization upon connection of a malicious USB device that acts as a keyboard. This includes fast detection and no user involvement in the security decision.

5.2.1 Threat Model

We assume an enterprise environment where computers are equipped with USB ports and the users are free to plug in and unplug USB devices for their day-to-day work duties (e.g., mass storage devices, headsets, and web cameras for teleconferences).

We further assume that the attackers succeed in connecting a crafted device with a malicious USB firmware to one or more of these computers. This can be achieved by the attackers themselves, if they have physical access to the targeted computer, inside or outside the premises of the enterprise. Or, by handing in a malicious device to legitimate users and exploit their curiosity (e.g., drop a USB thumb drive in their postal mailbox) or apply a social engineering attack vector (e.g., “can you please print the file from my USB thumb drive?”).

We do not consider attack vectors such as USB storage media loaded with malware (e.g., exploit the “autorun” feature). Mitigations for such attacks are already offered by commercial antivirus products [122]. We also do not consider attacks that exploit (unknown) vulnerabilities of the USB device drivers of the host operating system triggered by malformed USB packets [71]. Instead, we assume that all USB packets are well-formed and valid according to the USB protocol.

5.2.2 Patterns of Keypress Injection Attacks

The first step for developing appropriate defenses is to get a better understanding of the keypress injection attack patterns. Towards this direction, we experimented with two USB device types that their firmware can be updated and for which appropriate reflashing tools are available. The first one was a Rubber Ducky device (cf. Figure 5.2a). The second one was a Toshiba USB 3.0 USB drive (cf. Figure 5.2b). The latter includes the Phison PS2251-03 micro-controller chip built-in, which is known to support firmware reflashing¹⁵.

In the case of Rubber Ducky, it sufficed to compile a new firmware, which contained an attack payload, by using the provided compiler and then copy the firmware on its SD card.

The case of the benign Toshiba USB drive required some additional steps. First, the original firmware of the device was dumped, then the attack payload was integrated in the firmware, and finally the firmware was written back to the chip. This was not an error-free process. Our first attempts ended up with unusable (bricked) devices and unreliable functionality of the controller chip resulting in unstable behavior.

¹⁵<https://github.com/brandonlw/Psychson/wiki/Known-Supported-Devices>



Figure 5.2: A Rubber Ducky USB drive featuring an additional 128 MB SD card (left) and a Toshiba USB drive featuring a Phison PS2251-03 chip (right)

We also prepared a desktop computer that acted as the host for the attacks. We used the Wireshark network protocol analyzer¹⁶ to monitor the USB connections and collect the related USB packet traces for further analysis.

As an attack demonstration scenario, we opted to use the automatic launch of a text editor in the Linux operating system, including some text filling. Once connected, the malicious devices registered themselves as keyboards and sent the necessary keypresses. The sequence of events was as follows:

1. An artificial delay of 500 milliseconds.
2. Send the ALT key followed by the F2 key in order to prepare an application launch.
3. An artificial delay of 500 milliseconds.
4. Send a string of characters for launching a text editor (e.g., gEdit or mousepad), followed by the ENTER key.
5. An artificial delay of 500 milliseconds.
6. Send one paragraph of text comprising 515 characters from the Bacon Ipsum (<http://baconipsum.com/>) text.

¹⁶<https://wireshark.org/>

Listing 5.1 depicts the attack script in the Ducky Script language¹⁷.

Listing 5.1: Attack demonstration in Ducky Script.

```

1 DELAY 500
2 ALT F2
3 DELAY 500
4 STRING mousepad
5 ENTER
6 DELAY 500
7 STRING Bacon ipsum dolor amet [...]
8 ENTER

```

Albeit not malicious in nature, the attack scenario above serves as a baseline for building weaponized attacks, such as opening a terminal, disabling any running antivirus service, and running a `wget` command to download malicious code to the attacked system. The artificial delays are necessary to provide the operating system with enough time to successfully respond to issued commands, e.g., opening the text editor. The attack script sends 526 keypresses in total.

We ran each attack (BadUSB and Rubber Ducky) ten times and collected in total 20 Wireshark traces. We analyzed these traces and focused on the timing patterns of the `KEY_DOWN` events that are sent when a key on a keyboard is pressed.

Figure 5.3 depicts the distribution of the distance between each of the 5,260 recorded events (i.e., the interarrival time of the keypresses) for each of the ten repetitions of each attack in box-and-whisker diagrams. There are only a couple of outliers for each trace (Capture ID), which are at about 1.35 seconds (in the case of BadUSB) and 1.00 seconds (in the case of Rubber Ducky) as depicted in Figure 5.3a and Figure 5.3b. The median value in both cases is about 6 ms and almost all values are concentrated in a narrow band around this value, as depicted in Figure 5.3c and Figure 5.3d.

This was a stable behavior in all traces and for both attacks; Rubber Ducky exhibits a greater variability, but still within the narrow band. We ran repeated experiments with all the payloads made available by the Rubber Ducky authors¹⁸. There were in total more than 70 different payloads at the time of writing. The analysis of the traces revealed that *all* of the payloads produced the keypresses with no delay, i.e., the USB-based keypress injection attacks try to conclude their malicious actions as fast as possible.

From an attack point of view, it is a rational choice to inject keypresses as fast as possible: for an attack to be successful, all the events of the script must execute without being interrupted by any user-initiated typing activity. Since the attacked system has two keyboards now, it is possible that the user continues their typing activity. In this case, typing will intermix with the (fake) keypresses of the attack script and, thus, neutralize the

¹⁷The Bacon Ipsum text is truncated for the sake of readability.

¹⁸<https://github.com/hak5darren/USB-Rubber-Ducky/wiki/Payloads>

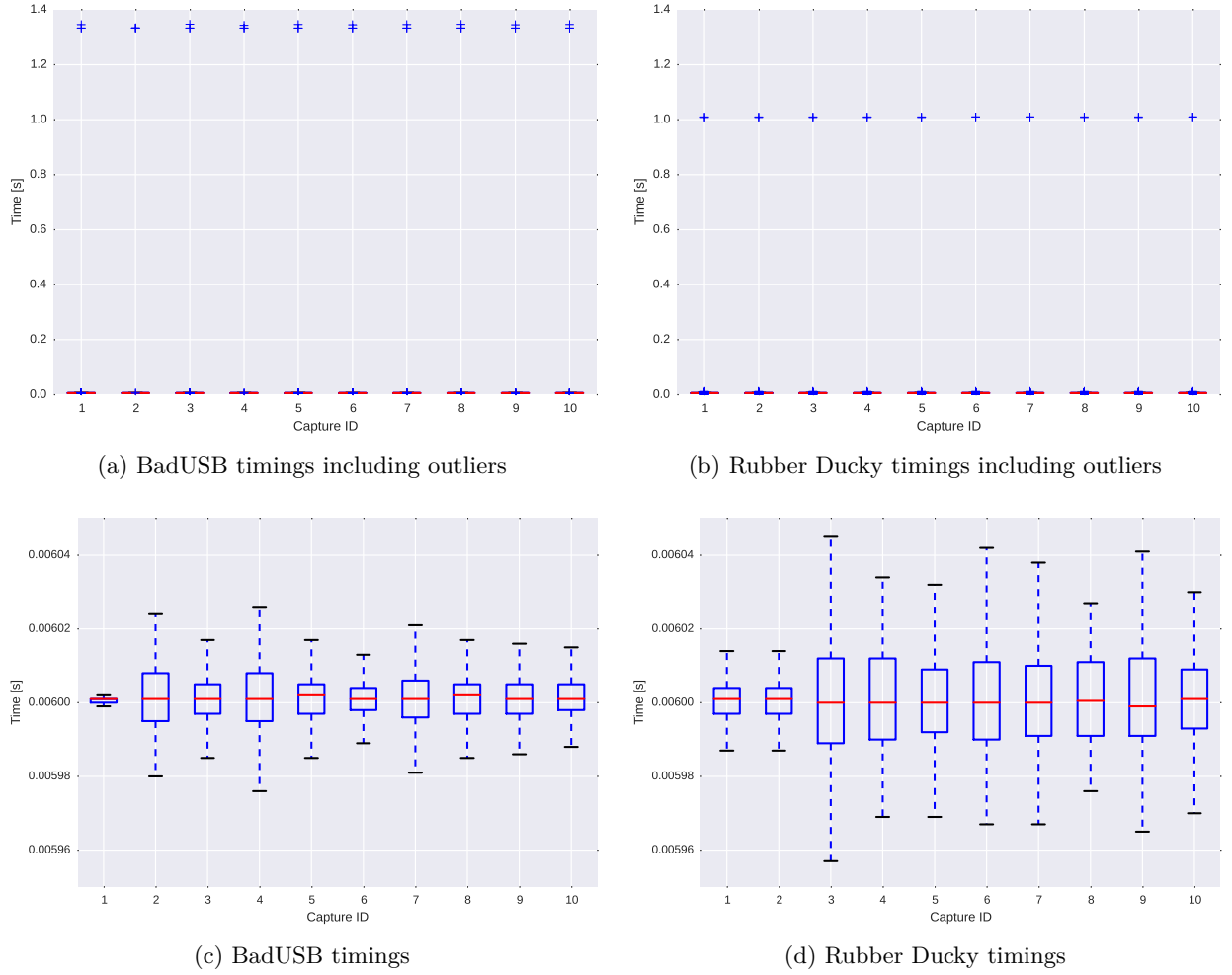


Figure 5.3: Interarrival times of KEY_DOWN events in collected traces (outliers excluded for the sake of clarity)

latter by accident (and possibly frustrate the user with the extra characters). The less the time in between the (fake) keypresses, the more the probability of a successfully-launched attack. This is a key observation for the design of our defense.

5.2.3 Keystroke Dynamics

Research in *keystroke dynamics* (or *keystroke-based biometrics*) suggests that human typing patterns exhibit variations and these “typing dynamics” can be strong enough to be used for authentication or identification purposes [144]. Also, human beings cannot perform better than 80 ms between their keypresses [151]. In contrast, our analysis reveals that BadUSB-like attacks inject keypresses at an almost stable rate of 6 ms.

These two rates differ by an order of magnitude.

5.2.4 USBlock: Blocking Malicious USB Packet Traffic

USBlock is a defense we devised that exploits the temporal gap between human and BadUSB-like attack typing dynamics. The design assumption is that a USB host monitor (i.e., USBBlock) has access to precise timing information of received USB packets and is able to distinguish *fast* between the normal typing behavior of human beings and the abnormal keypress sequence (AKS) timings of BadUSB-like attacks.

A simple case of an AKS is a “*rapid (keypress) event sequence*” (RES). We define a RES using two components, a time threshold value t and a sequence threshold value s . We say that a RES occurred whenever we observe a sequence of s consecutive keypresses with an interarrival time less than t seconds between each of them. If a RES occurs, a defense action must be taken.

The selection of the exact values for t and s is a design choice. The t -threshold can be any value $0.006 < t < 0.08$, i.e., anything between the 6 ms median value of the BadUSB-like attacks and the lower limit of 0.08 seconds for humans. A t close to the lower bound of 0.006 makes USBBlock *more prone* to false negatives, thus, risking a successful attack going unnoticed. Yet, it reduces the probability of false positives, as humans cannot type that fast. In contrast, a t close to the upper bound 0.08 makes USBBlock *more prone* to false positives, thus, risking user complaints on legitimate usage scenarios. Our analysis of the collected attack and human-typing traces (cf. Section 5.3) suggests that a $t = 0.02$ is a sufficient threshold for the current generation of BadUSB-like attacks.

An $s = 1$ makes USBBlock *blindly aggressive* (many false positives), penalizing even in a single occurrence of a keypress under the timing threshold. In contrast, a large s (e.g., over 100) allows USBBlock to make *confident decisions* on the presence of an AKS. However, one must account two additional points. First, if the decision algorithm does not react in real-time, it risks to allow an attack to have occurred already by the time a (correct) decision is made. This is unacceptable from a security point of view. Second, the length of the attack vector, i.e., the number of keypresses injected to realize the attack, might be lower than s (e.g., only 20 or 30 keypresses compared to an $s = 100$). In this case, USBBlock will fail to detect the attack altogether, as there are not enough “malicious” keypresses present to react on.

Our analysis of the available traces suggests that a short $s = 3$ offers the clear advantage that a keypress injection attack is detected and prevented just after the very first few fast keypresses are sent. The attack traces *contained not a single case* where two or more outlier values came in pairs or bursts. Furthermore, to the best of our knowledge, no attack vector can be realized with only three keypresses. Hence, $s = 3$ is an excellent choice.

The capabilities of USBBlock are better demonstrated with an example of a defense realization against current generation of Ducky Script malicious payloads. A USB

monitor runs as a piece of software on the host system and measures the interarrival times of keypresses sent by each connected USB device for a RES, using the t and s parameters. When a RES is detected, the USB monitor instructs the USB hub to switch off the power of the USB port for some (e.g., ten) seconds (configurable). It also instructs the operating system to unload (unbound) the respective device driver. This in effect blocks access to the suspicious device. We highlight that *no additional* piece of hardware is needed for performing these actions. The disconnection approach ensures that no user involvement is required in restoring the connectivity of other USB devices that are possibly attached to the specific USB port where the attack occurred.

One may argue that the design opts for a rather aggressive reaction to suspicious events. We believe that our two-step disablement approach accommodates this. It is better for users to notice an occasional interruption of their normal flow (in case the events are indeed produced that fast by human beings) rather than risking an infection by a malicious device. If the interruptions become too frequent, it is an indication of an attack in progress (e.g., a USB drive trying to relaunch the attack). In this case, it would be better for the IT support personnel to inspect the offending system.

5.2.5 USBlock Defenses Against Advanced Attacks

USBlock with RES defends currently against *all known* malicious Ducky Script payloads. The approach of USBlock is generic enough and allows to realize and integrate new defenses against new attacks. As the cat-and-mouse game between attackers and defenders might evolve for BadUSB-like attacks, RES can be replaced by a more complex AKS detection logic. Such logic will push attackers to adjust keypress injection rates to mimic human behavior. This task will be more and more difficult to both realize in the constrained environment of USB firmware and to match specific user typing patterns. At the moment, this is not deemed necessary and would incur additional and unnecessary processing overhead.

We note that USBlock with RES cannot be defeated by malicious firmware that delays the launch of the payload (i.e., the start of the attack), once the fake device is plugged in. USBlock is continuously monitoring for keypress events. Hence, when a RES occurs, USBlock will detect it, no matter how long ago the fake device was plugged.

One may argue that a malicious Ducky Script can introduce delays between keypresses to avoid detection. However, the constant interarrival time of the keypresses is an easily-detectable AKS pattern. Should a Ducky Script generate random delays between keypresses, these must occur as a human typing pattern, sparse in time. At this rate, they will become intermixed with the normal typing activity of the user (and thus, neutralized) or they will become noticeable by the user, as the attacks do not happen “in the blink of an eye” anymore (e.g., the cursor is moved or the focus of the working window is lost). Should the user be incapable or unwilling to notice the additional typing activity on their monitor, we must rely on enterprise network security defenses that might be able to detect the malicious activity of the attacked system, once infected.

Clearly, analyzing the typed *command strings* (from humans or scripts) and reasoning about their (possibly) malicious *intentions* is beyond the scope of USBlock. Such fine-grained keypress analysis will probably not be acceptable by the users, as it severely violates their privacy, effectively creating an Orwellian feeling of constant monitoring.

One may argue that a malicious USB firmware can monitor the status of the operating system and launch the payload during periods of user inactivity (e.g., when there is no typing activity for a long time). We are not aware of such capabilities for Ducky Scripts and, to the best of our knowledge, such information cannot be requested by the USB firmware from the operating system over USB packets. Even if such capabilities exist in first place, USBlock is on the side of the operating system. Thus, it can also access such information and integrate them into its decision logic to block malicious USB traffic. We further note that such an attack cannot be launched at all if the screen is locked (e.g., due to inactivity or a precautionary measure by the user when plugging an untrusted device).

5.2.6 Limitations of USBlock

While USBlock defends successfully against keypress injection attacks, it is not free of limitations. USBlock lies and relies on information at the level of the operating system. As such, it cannot defend against attacks launched at the BIOS level, as is the case of BadBIOS discussed in Section 5.1.

Hardware keys, like Yubico YubiKey¹⁹, are a popular means of two-factor authentication. Such devices identify themselves to the operating system as keyboards and “type” one-time passwords and other sensitive information on behalf of their owner. Hence, USBlock will interpret the YubiKey rapid keypresses as an attack. To overcome this, USBlock implements internally the following check for YubiKey devices: if a connected USB device reports a (VID, PID) of Yubico, the RES logic is disabled for this device. Rather, USBlock monitors the USB device traffic to ensure that the USB packet payload comprise exclusively “MODHEX” characters²⁰. This is an improved approach over (VID, PID) whitelisting [146], as anyone can fake the reported (VID, PID). As discussed earlier in Section 5.1, these are inherent limitations of the USB protocol itself rather than of USBlock. The host system must blindly trust the information provided by peripheral devices without any authentication support.

5.3 Proof-of-concept Implementation and Evaluation

We now describe a proof-of-concept implementation of the designed system and its evaluation in realistic environments. The evaluation comprises two parts: the first part relates to the long-term stability of the prototype and to evaluating the effect of the

¹⁹<https://www.yubico.com/products/yubikey-hardware/>

²⁰The modified hexadecimal characters are {b, c, d, e, f, g, h, i, j, k, l, n, r, t, u, v}, cf. <https://developers.yubico.com/OTP/>.

temporal variations in typing habits; the second part explores the typing dynamics of different users.

5.3.1 Proof-of-concept Implementation

We realized a proof-of-concept implementation for the Linux operating system comprising two parts. The first part is a Linux loadable kernel module (for kernel version 4.2) that monitors for the keypresses. Being in the kernel, this part is as close as possible to the raw information about keypresses received from the USB device and enriches them with very precise timings. The kernel module then forwards the enriched information to the second part residing in userland. The communication between the two parts occurs over a `netlink` socket that is registered in both the kernel space and userland. This is an approach similar to the one of [148]. However, the latter does not support timestamps and multi-packet processing, which are necessary for our aims.

The second part is a Python script. This part implements the rapid event sequence (RES) detection logic and is responsible for unbinding the offending USB driver from the kernel. This effectively disconnects the device interface from the system. If the driver of the device is automatically re-bound (as part of an ongoing attack), then, as an additional protective measure, the driver for the corresponding USB hub on which the device is connected is also unbound for ten seconds. This effectively removes all the devices connected on this hub. The unbind/re-bind procedure is repeated until a user action is initiated or a system administrator takes over. We note that the USB packet processing, from kernel capture to RES detection (via the Python script) and reaction, requires on average about 0.3 ms per packet. Hence, it takes 1 ms to detect an $s = 3$ RES. This is more than enough processing time, given that the median interarrival time for BadUSB-like keypress injections is about 6 ms.

Listing 5.2: Help output of the USBBlock userland tool.

```

1 usage: USBBlock.py [-h] [--start] [--stop][--daemonize] [--boot]
2                   [--threshold] [--keycounter][--debug] [--logfile] [-v]
3
4 optional arguments:
5   -h, --help            show this help message and exit
6   --start               Start the userland part of
7                       USBBlock
8   --stop                Stop the userland part of
9                       USBBlock
10  --daemonize            Start the userland part as a
11                       daemon
12  --boot                 Start the daemon on boot time
13  --threshold            Set the detection threshold for
14                       USBBlock. Default is 0.03 seconds
15  --keycounter           Set the number of successive
16                       keystrokes to be within the
17                       threshold
18  --debug                Enable debug output. Default
19                       output into /var/log/kern.log
20  --logfile              Change the log file destination.
21  -v, --version          show program's version number and
22                       exit

```




Figure 5.4: Pictures of the research prototype system used for the user study

Listing 5.2 displays the USBBlock usage message output, when launched with the help option. The default actions, if the application is launched with no arguments, are to log the collected keypress timings to the `/var/log/kern.log` file and to start it as a “root daemon” with the default threshold of $t = 0.03$ seconds. The latter is a precautionary measure to prevent non-privileged users from stopping the application.

5.3.2 Evaluation of Temporal Variations

We evaluated the effect of temporal variations in typing as follows. We installed the prototype in one of the authors’ computers, a notebook running Ubuntu 15.10. The notebook was used on a day-to-day basis with a USB-connected keyboard for a period of three months. The aim was twofold: on the one hand, to evaluate the stability of the prototype and to discover any problems that it might cause; on the other hand, to study the user’s typing behavior in the course of a long period that involved a multitude of typing activities including code development, debugging, system administration tasks, scientific paper writing, preparation of talks and presentations, shell scripting, email typing, and web browsing.

The architecture of the monitoring infrastructure is depicted in Figure 5.5a. We instrumented the kernel module `usbmon` [162] to collect and log the USB events and act as a middleware between the kernel and our offline analysis tools. In userland, the TShark part of the Wireshark bundle was used to “listen” to USB events sent by `usbmon`. A Python script was used to generate pcap-formatted files containing all the USB-related events. A second Python script was then used to process the files and store the collected information into a database for later analysis.

Overall, we did not experience any kind of stability problems while operating the prototype and the additional monitoring infrastructure. Neither we experienced any kind of measurable performance degradation. It did not cause any side effects (at least that we

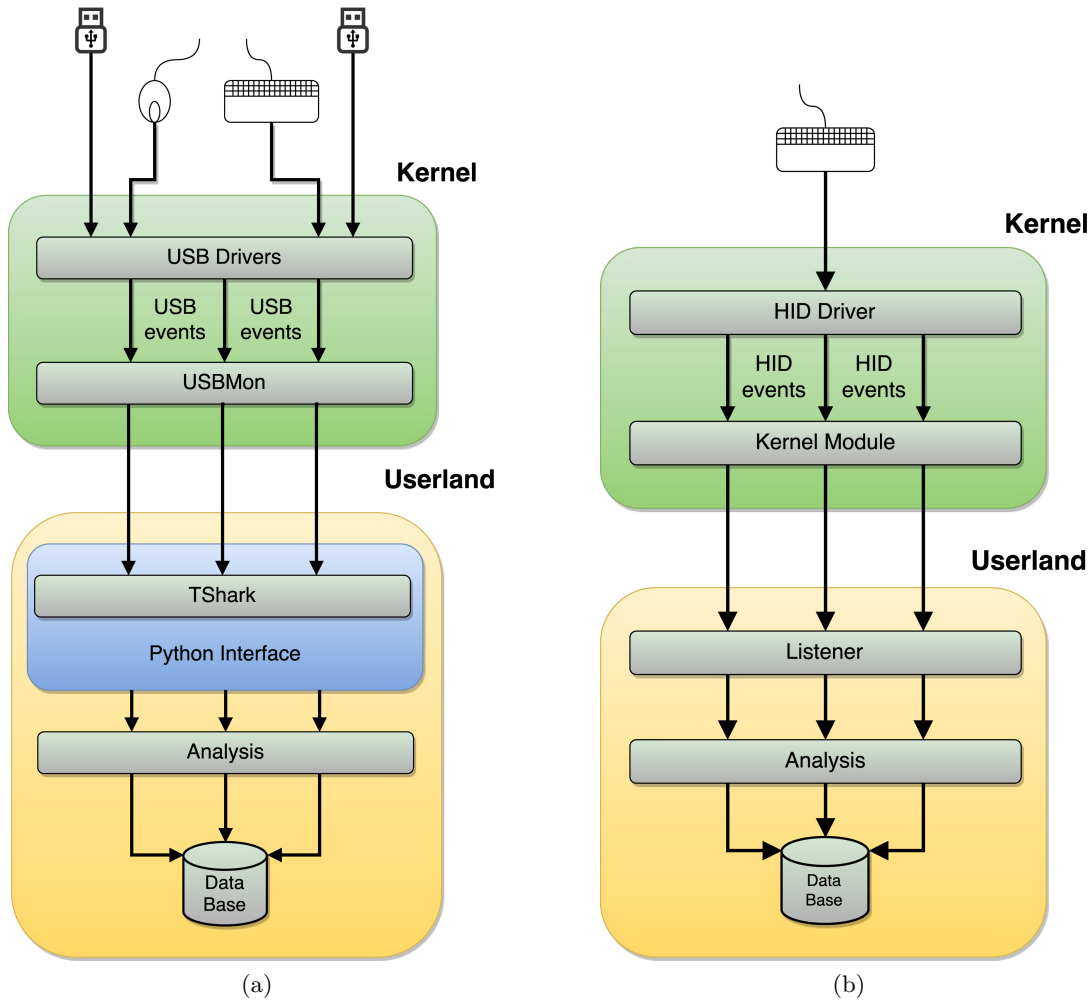


Figure 5.5: Architecture for monitoring infrastructure (a) evaluation of temporal variations and (b) evaluation of typing dynamics.

became aware of). Thus, the temporal variations in typing did not affect the operation of the prototype.

We collected in total timing information for more than 466,000 keypresses over more than 60 working days. Less than 1% of them were below the $t = 0.02$ second threshold, while the vast majority ranked quite higher as depicted in Figure 5.6. The median value of interarrival times was 0.10 seconds, while the average time was 0.21 seconds.

There was *no single case* of three or more consecutive keypresses with an interarrival time below the defined threshold. Thus, it was never the case that the prototype unbound the USB keyboard device driver, i.e., we experienced *no false positives*.

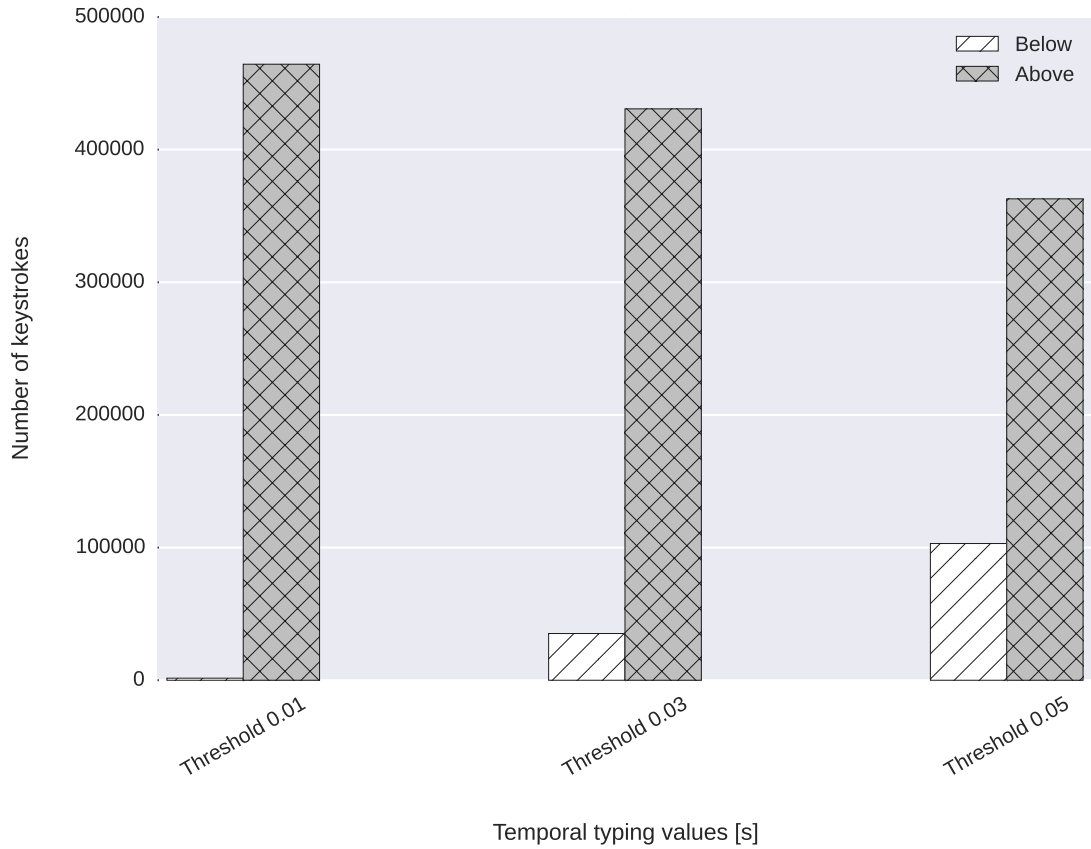


Figure 5.6: Interarrival times for the long-term typing experiment

5.3.3 Evaluation of Typing Dynamics

The second phase of the evaluation was a study on the effect of the typing dynamics. We designed a small-scale user study so as to collect evidence about typing patterns and compare them against the behavior of the Rubber Ducky and BadUSB attacks as well as published literature for typing dynamics.

We developed a research prototype system comprising a headless Raspberry Pi Model 2B running Ubuntu server 15.10, a USB keyboard, and a battery pack for autonomous operation. Figure 5.4a and Figure 5.4b depict our research prototype system and a close-up of the headless system. Similarly to the previous evaluation, we used a kernel module to collect the keypress timing information and send them to a userland application. The latter collected and aggregated the information prior to storing them into a database for later processing. The architecture of the monitoring infrastructure is depicted in Figure 5.5a.

We recruited 33 volunteers from our organization for this experiment. We visited each participant at their desk and asked them to type a short text in the comfort of their desk

using our research prototype system. We offered the participants the option to either plug in their keyboard or use the keyboard of our prototype. All but one participants opted to use our keyboard, as it felt more convenient for them not to unplug their keyboard, or because they were using laptops and docking stations.

We asked the participants to type the same, randomly-selected paragraph of the Bacon Ipsum text comprising 71 characters. Figure 5.7 depicts the distribution of the distance between each of the 71 recorded events (i.e., the keypresses interarrival time) produced by each participant in box-and-whisker diagrams. The diagrams indicate that each participant exhibited different typing patterns. Almost all diagrams contain a large number of outliers towards larger numbers. Delays of one or even three seconds between keypresses are noticed. The overall median value was 0.20 seconds and the average time reached 0.30 seconds. Despite the per-participant differences, there was *not a single case* where our research prototype detected erroneously a RES, i.e., once more, there were *zero false positives*.

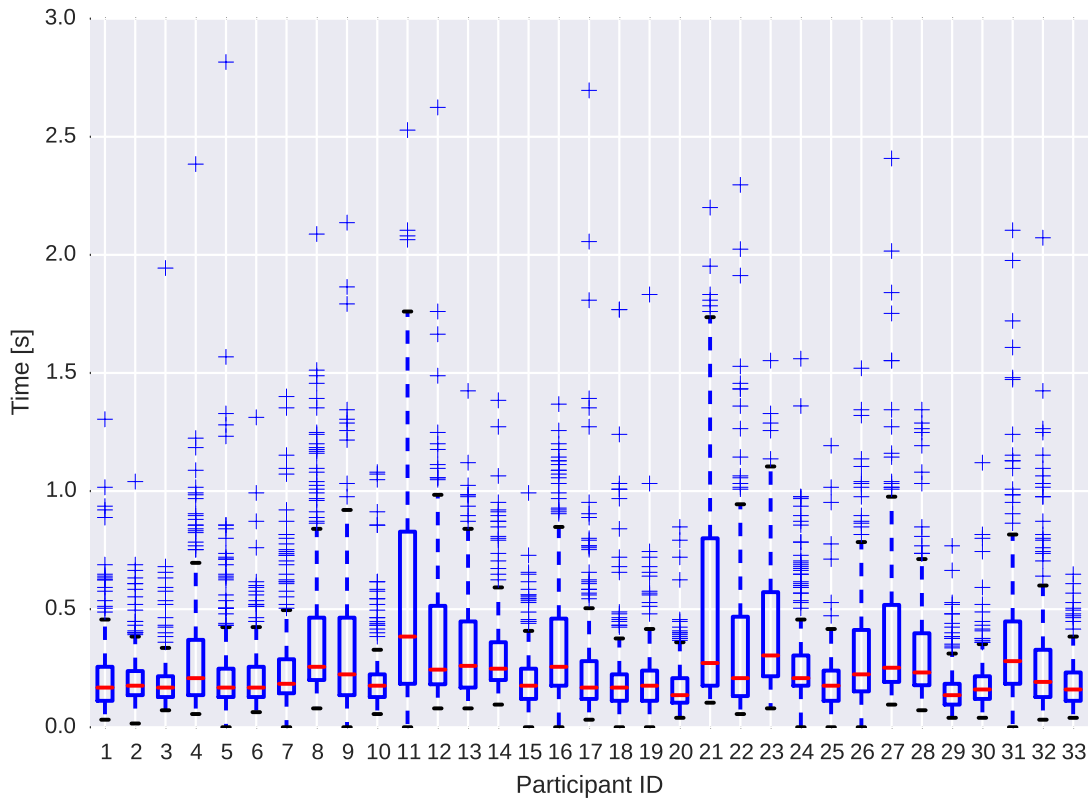


Figure 5.7: Temporal variations in the typing dynamics experiment.

Our typing dynamics analysis results are in alignment with published literature [144]. Hence and for the sake of research efficiency and economy, we opted not to expand the study to more participants.

The second part of the evaluation confirmed that typing dynamics are quite stable among users, despite some unavoidable differentiations. More importantly, even in short texts, the human typing dynamics are clearly above the detection threshold we have defined and clearly distinguishable from that of the Rubber Ducky and BadUSB attacks (cf. Figures 5.7 and 5.3). Thus, such information can serve as a heuristic for detecting and defending against the attacks.

5.4 Enriching Incident Response for BadUSB-like Attacks

A malicious USB firmware attack does not necessarily involve information transmission through a network. Rather, it may steal information from the attacked computer and store them in the USB drive or even hide them in filesystem metadata, as recently demonstrated [115]. In this setting, a computer antivirus is not able to detect the malicious piece of software, as it neither executes on the computer system nor it generates network traffic.

We now turn our focus to improving the incident response procedures. In an enterprise environment, should a BadUSB-like attack evades detection or an incident is indeed recorded, it is important to triage the available computing systems for further analysis. Efficient and effective incident handling relies on identifying quickly which systems are possibly exposed to the (unknown) threat. This is especially true if a USB-based attack is part of an advanced persistent threat campaign.

Towards this end, we devised a prototype analysis tool for automating the report of the spread of USB devices and tested it with a help of a company network that volunteered to participate in our study.

5.4.1 Reference Dataset Creation

Nowadays, multiple enterprise-grade system administration tools are available that allow for automated remote administration of multiple systems. For our case, the company IT administrators parsed the Microsoft Windows registry and collected the history of the USB device connected to 60 managed computers using the Microsoft System Center Configuration Manager. The whole procedure was fully scripted and automated, requiring minimal intervention by the administrators. Other equally suitable options could have been the Microsoft PowerShell and the GRR by Google [36]. The collected information were pseudo-anonymized to protect the innocent, before processing the data for the needs of our study.

The collected information includes: the USB device name; its product identifier; its vendor identifier; its serial number; the date of first and last connection (plug-in); the USB class; and the driver version. For the needs of our study it sufficed to use the tuple `<vendor ID>:<product ID>:<serial number>` so as to uniquely identify each device. As per definition, a serial number by itself should be a unique identifier. However, we observed that USB serial numbers are not unique across all devices and

5.4.2 Analysis of USB Device Propagation

Figure 5.8: Alternative visualization style for tracking connected USB devices. Visits of devices (red) to multiple clients (black).

However the default visualization style is depicted in Figure 5.9. In the first step, the IT administrators have an initial view of the offending computer that reported the incident. Then, the tool reports all the USB devices that were attached to this computer, including the one that caused the incident. The administrators can then view which systems this device has visited last in a timeline. Finally, they can have a view of all possibly infected computers. This gives a sense of the exposure scale, an indication for prioritizing their analysis efforts, and a means to build a chain of events for further investigation.

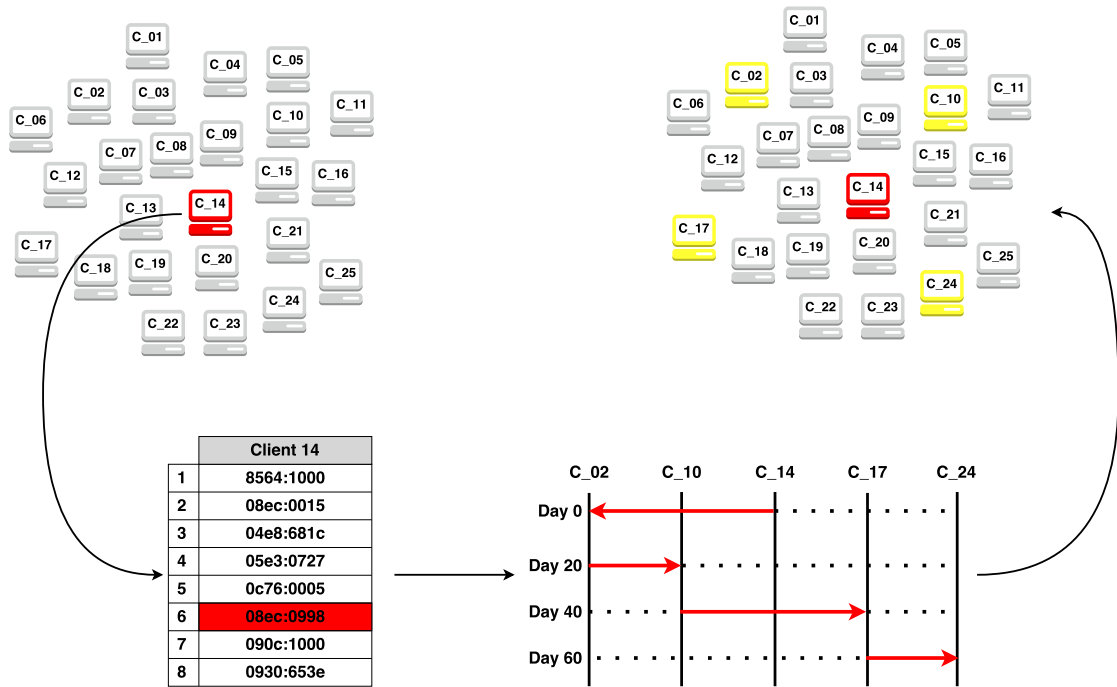


Figure 5.9: Prototype analysis tool. Visualize incident computer C_14 (1), report of connected USB devices (2), timeline of USB device visits (3), and complete view of triaged computers of company network (4)

5.5 Conclusions

BadUSB-like attacks are a realistic threat. Yet, system-level defenses cannot be realized in the form of malware analysis tools for USB firmware. USB device whitelisting can offer some protection in specific scenarios, but its applicability is hindered when scaling in enterprise-level networks. Proposed defenses in the literature mandate the user involvement in the trust decisions. This is a suboptimal, error-prone design choice.

In this chapter, we studied the *temporal* characteristics of BadUSB-like attacks. We proposed USBlock to block malicious USB packet traffic. Our proposal is extensible and can integrate additional features for coping with future, advanced attacks. Residing

on the side of the operating system, USBlock has an advantage over a malicious USB firmware payload executing on a peripheral device and interacting with the main system.

We implemented a proof-of-concept defense module for the Linux kernel. We evaluated its stability under different usage patterns for three months and studied the user temporal variations and typing dynamics in a small-scale study. The collected evidence suggest that our implementation caused no issues while human typing behavior was clearly distinguishable from that of the existing known attacks.

Our findings indicate that it is feasible to realize advanced defense mechanisms for BadUSB-like attacks by integrating system-level temporal characteristics and *without* involving the user in the trust decisions.

CHAPTER 6

Conclusions and Future Work

Computer-related crimes are increasing over the time. Not only in quantity but also in sophistication, thereby increasing the associated effort and accompanying monetary costs to defend against them. It is becoming widely acceptable that fully secure systems cannot be designed and engineered, especially when these systems do not exist in isolation but rather as crucial components of complex socio-technical systems including human beings and their actions. In such a *modus operandi*, the interest shifts towards reducing the associated risks and minimizing the attack surface.

In this thesis, we designed, engineered, and evaluated operating-system-level defenses against USB-based attacks, following a holistic approach, with the aim to increase the response capabilities of CSIRT's and related bodies in case of security incidents arising from cybercriminals. Our contributions revolve around three critical components.

We proposed a novel, file-based whitelisting approach, which utilizes available databases and cross-correlated files to exclude already known files from an investigative corpus. In our case study, we have been able to filter 78% of files known to be benign. Such an enormous reduction without any performance penalty significantly upgrades the operational capabilities of CSIRT's in terms of effectiveness and efficiency, both in time and space, allowing faster response times and improved allocation of available resources and focus towards potentially suspicious cases, which would go unnoticed otherwise.

To further reduce the investigative corpus of a forensic analysis, we proposed the use of sub-file hashes contained with every Torrent file. These readily-available information allow to exclude fast large portions of files-under-investigation. We constructed and openly-released datasets of more than 6 billion sub-file hashes. These hashes can be used by any CSIRT to properly identify and whitelist more than 60 million different files. The datasets are steadily growing and provided to interested researchers for further experimentation.

File contents are not the only place a malware author can hide malicious code or use as an exfiltration channel to bypass network perimeter and leak information outside an organization. We were the first to identify that file timestamps, a special form of filesystem metadata, in modern operating systems can be utilized to construct a leakage channel of steganographic strength. This channel can be further enhanced using error-correcting codes and modern encryption to further hinder prevention and detection techniques of examiners. The robustness of the steganographic channel was further validated in realistic usage scenarios spanning multiple weeks of filesystem operations and involving actions such as file reordering, deletions, and additions. This previously unreported channel can now be incorporated in standard CSIRT investigations to identify potential exfiltration, hence, improving their operational capabilities. Furthermore, we devised techniques that can significantly reduce the channel capacity, if properly integrated in day-to-day operations of an organization, thus, further reducing the attack surface and the associated risk.

Files and filesystem metadata are not the only information carriers of malicious actions. Peripheral devices with USB interfaces contain firmware implementations of the USB protocol, which directly interacts with modern operating system at the kernel level bypassing all network perimeter defenses. We devised USBlock, a novel method to defend against keystroke-injection attacks launched by malicious USB firmware. USBlock is the very first approach in the literature to exclusively rely on system-level decisions; it does not involve the user at any trust decision for the prevention and detection of an attack. This not only allows more accurate reaction but also reduces the incident response time.

There are multiple directions of future research in the topics addressed by this thesis. More and more datasets and even whole virtual computing infrastructures producing these datasets are becoming publicly available as part of open science initiatives. At the same time cloud storage and large hard drives are becoming affordable to all consumers. We consider an interesting direction to explore the possibility of integrating such open sources in the whitelisting-corpora. Also, to explore use of novel data structures and constructions, including blockchains, to efficiently store and retrieve information sub-file hashes.

Another direction of research is towards new filesystem and related operating-system metadata without excess capacity, thus, removing the possibility of new steganographic channels and eliminating existing, undisclosed ones. In parallel, more generic anomaly detection techniques, based on recent advances in machine learning and artificial intelligence, are needed to identify abnormal or unexpected distributions of characteristics of files and filesystems. Such techniques can provide early warnings signals and fast triage of artifacts to be analyzed for information leakage via unknown steganographic channels.

Keypress-injection attacks are just one class of potential threats that modified USB firmware can deliver. As USB directly interacts with the most trusted parts of the operating system kernel, further research on security threats and low-latency countermeasures is necessary. Along this line of research, it is interesting to explore low-overhead in-kernel

isolation of USB protocol parsing and interaction processing. The USB protocol variants and newer versions do not incorporate system security in their design. It is insecure by design. Sooner or later, USB will be replaced by a next-generation protocol interface, with a physical or wireless connector. It is of utmost importance to research ahead of time the protocol characteristics so that it provides both “Security-by-Design” and “Privacy-by-Design” and not as an afterthought.

List of Figures

1.1	Response phase and timeliness vs. estimated security impact; thesis focus points.	3
2.1	Improved Steps for the Forensic Process	14
2.2	File reduction in the reduced working copy	19
2.3	Deduplication rate of the corpus databases to NSRL RDS	21
2.4	Deduplication rate of each corpus database to the remaining databases . .	22
2.5	Average over the incremental deduplication rate	23
3.1	File content in a torrent file	28
3.2	Chunk hashes	29
3.3	Distribution of sub-file hash duplicates	34
4.1	How file timestamps are displayed to the users: (a) Ubuntu Linux, (b) Microsoft Windows, (c) Apple OS X.	42
4.2	Overview of storing data in the nanoseconds part of the timestamp fields.	46
4.3	Creation time (C), Synthetic, HDD, no delay, Python	61
4.4	Creation time (C), Synthetic, HDD, no delay, Powershell	62
4.5	Creation time (C), Synthetic, eSATA, no delay, Python	63
4.6	Creation time (C), Synthetic, eSATA, no delay, Powershell	64
4.7	Histogram of C-bins and A-bins for S2	66
4.8	Histogram of C-bins and A-bins for A1	67
4.9	Histogram of C- and A-bins in enterprise volume 9	68
4.10	Histogram of C- and A-bins in enterprise volume 15	68
4.11	Histogram of C- and A-bins in enterprise volume 18	69
4.12	Histogram of C- and A-bins in enterprise volume 40	69
4.13	Histogram of C- and A-bins in enterprise volume 42	70
4.14	Histogram of C- and A-bins in enterprise volume 45	70
4.15	Histogram of C- and A-bins in enterprise volume 48	71
4.16	Histogram of C- and A-bins in enterprise volume 60	71
4.17	Histogram of C- and A-bins in enterprise volume 65	72
4.18	Histogram of C- and A-bins in enterprise volume 67	72
5.1	USB packet sequence diagram (malicious behavior)	78

5.2	A Rubber Ducky USB drive featuring an additional 128 MB SD card (left) and a Toshiba USB drive featuring a Phison PS2251-03 chip (right) . . .	83
5.3	Interarrival times of KEY_DOWN events in collected traces (outliers excluded for the sake of clarity)	85
5.4	Pictures of the research prototype system used for the user study	90
5.5	Architecture for monitoring infrastructure (a) evaluation of temporal variations and (b) evaluation of typing dynamics.	91
5.6	Interarrival times for the long-term typing experiment	92
5.7	Temporal variations in the typing dynamics experiment.	93
5.8	Alternative visualization style for tracking connected USB devices. Visits of devices (red) to multiple clients (black).	95
5.9	Prototype analysis tool. Visualize incident computer C_14 (1), report of connected USB devices (2), timeline of USB device visits (3), and complete view of triaged computers of company network (4)	96

List of Tables

2.1	Overview of devices and storage capacities	16
2.2	File extraction distribution per source in corpus.	17
2.3	Corpus Details	20
3.1	Results of data collection for 2.3 million torrent files	33
4.1	Characteristics of filesystem timestamps.	44
4.2	Time to embed and extract information on filesystem.	55
4.3	Average and standard deviation of the occurrences of unique timestamps in the synthetic dataset (delayed creation)	59
4.4	Time in minutes to create a set 200,000 files for the synthetic dataset (no delay)	60
4.5	Average and standard deviation of the occurrences of unique timestamps in the synthetic dataset (creation without delay)	61
4.6	Number of unique timestamps (bins) of files in the consumer-grade dataset filesystems	63
4.7	Average and standard deviation of the occurrences of unique timestamps in the consumer-grade dataset (aggregated)	65

Bibliography

- [1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.
- [2] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)*, 3(3):9, 2007.
- [3] M. Akgül and M. Kırıldoğ. Internet censorship in Turkey. *Internet Policy Review*, 4(2), 2015.
- [4] R. Amirtharajan, J. Qin, and J. B. B. Rayappan. Random image steganography and steganalysis: Present status and future directions. *Information Technology Journal*, 11(5):566–576, 2012.
- [5] B. Anderson and B. Anderson. *Seven deadliest USB attacks*. Syngress, 2010.
- [6] D. Anderson. Splinternet behind the great firewall of China. *Queue*, 10(11):40:40–40:49, 2012.
- [7] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *Information Hiding*, pages 73–82. Springer, 1998.
- [8] R. J. Anderson and F. A. Petitcolas. On the limits of steganography. *IEEE Journal on Selected Areas in Communications*, 16(4):474–481, 1998.
- [9] S. Angel, R. S. Wahby, M. Howald, J. B. Leners, M. Spilo, Z. Sun, A. J. Blumberg, and M. Walfish. Defending against malicious peripherals with Cinch. In *USENIX Security Symposium*, 2016.
- [10] E. Antsilevich. Capturing timestamp precision for digital forensics. Technical Report JMU-INFOSEC-TR-2009-002, Department of Computer Science, James Madison University, USA, 2009.
- [11] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

- [12] J. Aycock and D. M. N. de Castro. Permutation steganography in FAT filesystems. In *Transactions on Data Hiding and Multimedia Security X*, pages 92–105. Springer, 2015.
- [13] K. Bailey and K. Curran. An evaluation of image based steganography methods. *Multimedia Tools and Applications*, 30(1):55–88, 2006.
- [14] V. Berk, A. Giani, and G. Cybenko. Detection of covert channel encoding in network packet delays. Technical Report TR2005-536, Department of Computer Science, Dartmouth College, USA, 2005.
- [15] G. Bonetti, M. Viglione, A. Frossi, F. Maggi, and S. Zanero. A comprehensive black-box methodology for testing the forensic characteristics of solid-state drives. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 269–278. ACM, 2013.
- [16] F. Breitinger and H. Baier. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In *Digital forensics and cyber crime*, pages 167–182. Springer, 2013.
- [17] F. Breitinger, B. Guttman, M. McCarrin, V. Roussev, and D. White. Approximate matching: definition and terminology. *NIST Special Publication*, 800:168, 2014.
- [18] F. Breitinger and V. Roussev. Automated evaluation of approximate matching algorithms on real data. *Digital Investigation*, 11:S10–S17, 2014.
- [19] D. Brezinski and T. Killalea. Rfc 3227: Guidelines for evidence collection and archiving. *Internet Engineering Task Force*, 2002.
- [20] D. C. Brock and G. E. Moore. *Understanding Moore’s law: four decades of innovation*. Chemical Heritage Foundation, 2006.
- [21] A. Burghardt and A. J. Feldman. Using the HFS+ journal for deleted file recovery. *Digital Investigation*, 5:S76–S82, 2008.
- [22] J. Camenisch and G. M. Zaverucha. Private intersection of certified sets. In *Financial Cryptography and Data Security*, pages 108–127. Springer, 2009.
- [23] B. Carrier. *File system forensic analysis*. Addison-Wesley Professional, 2005.
- [24] B. Carrier. The sleuthkit (tsk), 2015. URL: <http://www.sleuthkit.org/sleuthkit/>.
- [25] A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev. Face: Automated digital evidence discovery and correlation. *Digital Investigation*, 5:S65–S75, 2008.

- [26] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohn, M. Green, N. Heninger, R.-P. Weinmann, E. Rescorla, and H. Shacham. Where did i leave my keys?: Lessons from the juniper dual ec incident. *Commun. ACM*, 61(11):148–155, Oct. 2018.
- [27] A. Cheddad, J. Condell, K. Curran, and P. Mc Kevitt. Digital image steganography: Survey and analysis of current methods. *Signal processing*, 90(3):727–752, 2010.
- [28] P. Chen, L. Desmet, and C. Huygens. A study on advanced persistent threats. In *IFIP International Conference on Communications and Multimedia Security*, pages 63–72. Springer, 2014.
- [29] T. M. Chen and J.-M. Robert. The evolution of viruses and worms. *Statistical methods in computer security*, 1, 2004.
- [30] R. Chirgwin. Big mistake by big blue: Storwize initialisation usbs had malware. *The Register*, 2017.
- [31] G.-S. Cho. A computer forensic method for detecting timestamp forgery in NTFS. *Computers & Security*, 34:36–46, 2013.
- [32] K.-P. Chow, F. Law, M. Kwan, and P. Lai. The rules of time on NTFS file system. In *Second International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE 2007)*, pages 71–85. IEEE, 2007.
- [33] B. Cohen. The bittorrent protocol specification, bep-3. Online at http://www.bittorrent.org/beps/bep_0003.html.
- [34] F. Cohen. Computer viruses: theory and experiments. *Computers & security*, 6(1):22–35, 1987.
- [35] M. Cohen. Pyflag—an advanced network forensic framework. *Digital investigation*, 5:S112–S120, 2008.
- [36] M. Cohen, D. Bilby, and G. Caronni. Distributed forensics and incident response in the enterprise. *Digital Investigation*, 8:S101–S110, 2011.
- [37] K. Conger and T. Hatmaker. The shadow brokers are back with exploits for windows and global banking systems. *TechCrunch*, 2017.
- [38] I. Cox, M. Miller, J. Bloom, J. Fridrich, and T. Kalker. *Digital watermarking and steganography*. Morgan Kaufmann, 2007.
- [39] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security*, pages 143–159. Springer, 2010.
- [40] X. Ding and H. Zou. Reliable time based forensics in NTFS, 2010. Available at <https://www.acsac.org/2010/program/posters/ding.pdf>.

- [41] X. Ding and H. Zou. Time based data forensic and cross-reference analysis. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 185–190. ACM, 2011.
- [42] R. Dingleline and N. Mathewson. Anonymity loves company: Usability and the network effect. In *Workshop on the Economics of Information Security (WEIS)*, 2006.
- [43] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Symposium on Security and Privacy*, pages 326–343. IEEE, 1989.
- [44] Europol. 2015 Internet Organized Crime Threat Assessment (IOCTA). Online at <https://www.europol.europa.eu/content/internet-organised-crime-threat-assessment-iocta-2015>.
- [45] Europol. The Internet Organised Crime Threat Assessment (IOCTA) 2018. <https://www.europol.europa.eu/activities-services/main-reports/internet-organised-crime-threat-assessment>, 2018.
- [46] K. D. Fairbanks. An analysis of ext4 for digital forensics. *Digital Investigation*, 9:S118–S130, 2012.
- [47] C. Fidas, A. Voyiatzis, and N. Avouris. When security meets usability: A user-centric approach on a crossroads priority problem. In *14th Panhellenic Conference on Informatics (PCI)*, 2010. Tripoli, Greece, September 10–12.
- [48] E. Franz, A. Jerichow, S. Möller, A. Pfitzmann, and I. Stierand. Computer based steganography: How it works and why therefore any restrictions on cryptography are nonsense, at best. In *Information Hiding*, pages 7–21. Springer, 1996.
- [49] J. Fridrich and R. Du. Secure steganographic methods for palette images. In *Information Hiding*, pages 47–60. Springer, 2000.
- [50] S. Garfinkel. Digital forensics xml and the dfxml toolset. *Digital Investigation*, 8(3):161–174, 2012.
- [51] 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*, 7:S13–S23, 2010.
- [52] S. L. Garfinkel. Digital forensics research: The next 10 years. *Digital Investigation*, 7:S64–S73, 2010.
- [53] S. L. Garfinkel. Digital media triage with bulk data analysis and bulk_extractor. *Computers & Security*, 32:56–72, 2013.

- [54] S. L. Garfinkel and M. McCarrin. Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. *Digital Investigation*, 14:S95–S105, 2015.
- [55] A. Giani, V. H. Berk, and G. V. Cybenko. Data exfiltration and covert channels. In *Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense V*. International Society for Optics and Photonics, 2006.
- [56] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert messaging through TCP timestamps. In *Privacy Enhancing Technologies*, pages 189–193. Springer, 2003.
- [57] D. Gilbert. International space station infected with usb stick malware carried on board by russian astronauts. *International Business Times*, 2013.
- [58] D. Goodin. Nsa-leaking shadow brokers just dumped its most damaging release yet. *Ars Technica*, 2017.
- [59] G. Greenwald. *No place to hide: Edward Snowden, the NSA, and the US surveillance state*. Macmillan, 2014.
- [60] F. Griscioli, M. Pizzonia, and M. Sacchetti. USBCheckIn: Preventing BadUSB attacks by forcing human-device interaction. In *14th Annual Conference on Privacy, Security and Trust (PST)*, pages 493–496. IEEE, 2016.
- [61] A. Gupta, P. Kuppili, A. Akella, and P. Barford. An empirical study of malware evolution. In *First International Conference and Workshops on Communication Systems and Networks*, pages 1–10. IEEE, 2009.
- [62] 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)*, pages 849–864, 2015.
- [63] S. Han, W. Shin, J. Kang, J.-H. Park, H. Kim, E. Park, and J.-C. Ryou. IRON-HID: Create your own bad USB (white paper). HITBSecConf 2016 - Amsterdam. The 7th Annual HITB Security Conference in the Netherlands, <http://conference.hitb.org/hitbsecconf2016ams/materials/Whitepapers/Seunghun%20Han%20-%20IRON-HID%20-%20Create%20Your%20Own%20Bad%20USB%20Device.pdf>, 2016.
- [64] Herjavec Group. Official Annual Cybercrime Report. <https://www.herjavecgroup.com/the-2019-official-annual-cybercrime-report/>, 2019.
- [65] U. Hermann. Access Time Update. Online at http://forensicswiki.org/wiki/MAC_times#Access_Time_Update.

- [66] A. Hoog. *Android forensics: investigation, analysis and mobile security for Google Android*. Elsevier, 2011.
- [67] N. Hopper, L. von Ahn, and J. Langford. Provably secure steganography. *IEEE Transactions on Computers*, 58(5):662–676, 2009.
- [68] M. Huber, M. Mulazzani, M. Leithner, S. Schrittwieser, G. Wondracek, and E. Weippl. Social snapshots: Digital forensics for online social networks. In *Proceedings of the 27th annual computer security applications conference*, pages 113–122. ACM, 2011.
- [69] M. Hussain and M. Hussain. A survey of image steganography techniques. *International Journal of Advanced Science and Technology*, 54:113–124, 2013.
- [70] T. Ith. Microsoft’s photodna: Protecting children and businesses in the cloud, 2015. Online at <https://news.microsoft.com/features/microsofts-photodna-protecting-children-and-businesses-in-the-cloud/>.
- [71] P. Johnson, S. Bratus, and S. Smith. Protecting against malicious bits on the wire: Automatically generating a USB protocol parser for a production kernel. In *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017*, pages 528–541. ACM, 2017.
- [72] M. Kang. Usbwall: A novel security mechanism to protect against maliciously reprogrammed usb devices. Master’s thesis, University of Kansas, 2015.
- [73] S. Katzenbeisser and F. Petitcolas. *Information hiding techniques for steganography and digital watermarking*. Artech House, 2000.
- [74] K. Kent, S. Chevalier, T. Grance, and H. Dang. Guide to integrating forensic techniques into incident response. *NIST Special Publication (SP) 800–86*, 2006.
- [75] J. O. Kephart and S. R. White. Directed-graph epidemiological models of computer viruses. In *Proceedings on Research in Security and Privacy*, pages 343–359. IEEE, 1991.
- [76] M. Kerrisk. *The Linux programming interface*. No Starch Press, 2010.
- [77] H. Khan, M. Javed, S. A. Khayam, and F. Mirza. Designing a cluster-based covert channel to evade disk investigation and forensics. *Computers & Security*, 30(1):35–49, 2011.
- [78] D. Kholia and P. Wegrzyn. Looking inside the (drop) box. In *7th USENIX Workshop on Offensive Technologies (WOOT)*, 2013.
- [79] I. Kolochenko. Cybercrime: The price of inequality. *Forbes*, 2016.
- [80] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

- [81] M. Kryder and C. S. Kim. After hard drives – what comes next? *IEEE Transactions on Magnetics*, 45(10):3406–3413, Oct 2009.
- [82] D. Kushner. The real story of stuxnet. *Spectrum*, 50(3):48–53, 2013.
- [83] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3):49–51, 2011.
- [84] R. Langner. To kill a centrifuge: A technical analysis of what stuxnet’s creators tried to achieve. 2013.
- [85] K. S. Lee, H. Wang, and H. Weatherspoon. PHY covert channels: can you see the idles? In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 173–185. USENIX, 2014.
- [86] Y. Lee, H. Lee, K. Lee, and K. Yim. Cognitive countermeasures against bad usb. In *International Conference on Broadband and Wireless Computing, Communication and Applications*, pages 377–386. Springer, 2016.
- [87] J. Lessard and G. Kessler. Android forensics: Simplifying cell phone examinations. 2010.
- [88] B. Li, J. He, J. Huang, and Y. Q. Shi. A survey on image steganography and steganalysis. *Journal of Information Hiding and Multimedia Signal Processing*, 2(2):142–172, 2011.
- [89] M. H. Ligh, A. Case, J. Levy, and A. Walters. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014.
- [90] Y. Liu, C. Corbett, K. Chiang, R. Archibald, B. Mukherjee, and D. Ghosal. Sidd: A framework for detecting sensitive data exfiltration by an insider attack. In *42nd Hawaii International Conference on System Sciences (HICSS)*, pages 1–10. IEEE, 2009.
- [91] E. L. Loe, H.-C. Hsiao, T. H.-J. Kim, S.-C. Lee, and S.-M. Cheng. Sandusb: An installation-free sandbox for usb peripherals. In *3rd World Forum on Internet of Things (WF-IoT)*, pages 621–626. IEEE, 2016.
- [92] B. Martini and K.-K. R. Choo. An integrated conceptual digital forensic framework for cloud computing. *Digital Investigation*, 9(2):71–80, 2012.
- [93] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans, 2007. Online at <https://www.kernel.org/doc/mirror/ols2007v2.pdf>.
- [94] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

- [95] W. Mazurczyk and J. Lubacz. LACK –a VoIP steganographic method. *Telecommunication Systems*, 45(2-3):153–163, 2010.
- [96] McAfee. The Economic Impact of Cybercrime. <https://www.mcafee.com/enterprise/en-us/solutions/lp/economics-cybercrime.html>, 2018.
- [97] A. D. McDonald and M. G. Kuhn. StegFS: A steganographic file system for Linux. In *Information Hiding*, pages 463–477. Springer, 2000.
- [98] S. Mead. Unique file identification in the national software reference library. *Digital Investigation*, 3(3):138–150, 2006.
- [99] T. Micro. EternalRocks Emerges, Exploits Additional ShadowBroker Vulnerabilities, 2017.
- [100] Microsoft. Data execution prevention (dep). Online at <http://support.microsoft.com/kb/875352/EN-US/>.
- [101] Microsoft. Description of NTFS date and time stamps for files and folders. Online at <https://support.microsoft.com/en-us/kb/299648>.
- [102] Microsoft Developer Network. File Times. Online at <https://msdn.microsoft.com/en-us/library/ms724290/>.
- [103] T. Morkel, J. H. Eloff, and M. S. Olivier. An overview of image steganography. In *The Firth Annual Information Security South Africa (ISSA 2005)*, pages 1–12. electronically, 2005.
- [104] P. Mueller and B. Yadegari. The stuxnet worm. <http://www.cs.arizona.edu/~collberg/Teaching/466-566/2012/Resources/presentations/2012/topic9-final/report.pdf>, 2012.
- [105] M. Mulazzani, S. Neuner, P. Kieseberg, M. Huber, S. Schrittwieser, and E. Weippl. Quantifying windows file slack size and stability. In *Advances in Digital Forensics IX*, pages 183–193. Springer Berlin Heidelberg, 2013.
- [106] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. R. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security Symposium*, pages 65–76. San Francisco, CA, USA, 2011.
- [107] S. Murdoch and S. Lewis. Embedding covert channels into TCP/IP. In *Information Hiding*, pages 247–261. Springer, 2005.
- [108] National Crime Agency. Cyber crime assessment. 2016.

- [109] C. Neasbitt, R. Perdisci, K. Li, and T. Nelms. Clickminer: Towards forensic reconstruction of user-browser interactions from network traces. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1244–1255. ACM, 2014.
- [110] M. Neugschwandtner, A. Beitler, and A. Kurmus. A Transparent Defense Against USB Eavesdropping Attacks. In *Proceedings of the 9th European Workshop on System Security*, EuroSec '16, pages 6:1–6:6. ACM, 2016.
- [111] S. Neuner, M. Mulazzani, S. Schrittwieser, and E. R. Weippl. Gradually improving the forensic process. In *10th international conference on Availability, Reliability and Security*, pages 404–410. IEEE, 2015.
- [112] S. Neuner, M. Schmiedecker, and E. R. Weippl. Effectiveness of file-based deduplication in digital forensics. *Security and Communication Networks*, 15(9):2876–2885, 2016.
- [113] S. Neuner, M. Schmiedecker, and E. R. Weippl. PeekTorrent: Leveraging P2P Hash Values for Digital Forensics. *Digital Investigation*, 18(7):149–156, 2016.
- [114] S. Neuner, A. G. Voyiatzis, S. Fotopoulos, C. Mulliner, and E. R. Weippl. Usblock: Blocking usb-based keypress injection attacks. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 278–295. Springer, 2018.
- [115] S. Neuner, A. G. Voyiatzis, M. Schmiedecker, S. Brunthaler, S. Katzenbeisser, and E. R. Weippl. Time is on my side: Steganography in filesystem metadata. *Digital Investigation*, 18(7):76–86, 2016.
- [116] S. Neuner, A. G. Voyiatzis, M. Schmiedecker, and E. R. Weippl. Timestamp hiccups: detecting manipulated filesystem timestamps. In *12th International Conference on Availability, Reliability and Security*, pages 33:1–33:6, 2017.
- [117] T. H. News. Newly Found Malware Uses 7 NSA Hacking Tools, Where WannaCry Uses 2, 2017.
- [118] N. Nissim, R. Yahalom, and Y. Elovici. USB-based attacks. *Computers & Security*, 70(Supplement C):675–688, 2017.
- [119] K. Nohl and J. Lell. BadUSB - On Accessories that turn Evil. <https://srlabs.de/badusb/>, 2014.
- [120] H. Pang, K.-L. Tan, and X. Zhou. StegFS: A steganographic file system. In *Proceedings of the 19th International Conference on Data Engineering*, pages 657–667. IEEE, 2003.
- [121] M. Perklin. ACL steganography: Permissions to hide your porn. Online at <https://www.defcon.org/images/defcon-21/dc-21-presentations/Perklin/DEFCON-21-Perklin-ACL-Steganography-Updated.pdf>.

- [122] D. V. Pham, A. Syed, and M. N. Halgamuge. Universal serial bus based software attacks and protection solutions. *Digital Investigation*, 7(3):172–184, 2011.
- [123] S. Poeschel and J.-H. Gim. Btrfs on-disk format. Online at https://btrfs.wiki.kernel.org/index.php/On-disk_Format#Basic_Structures.
- [124] PwC. Global economic crime and fraud survey. 2018.
- [125] G. G. Richard III and J. Grier. Rapid forensic acquisition of large media with sifting collectors. *Digital Investigation*, 14:S34–S44, 2015.
- [126] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.
- [127] O. Rodeh and A. Teperman. ZFS –a scalable distributed file system using object disks. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2003)*, pages 207–218. IEEE, 2003.
- [128] M. K. Rogers, J. Goldman, R. Mislán, T. Wedge, and S. Debroya. Computer forensics field triage process model. In *Proceedings of the conference on Digital Forensics, Security and Law*, page 27. Association of Digital Forensics, Security and Law, 2006.
- [129] V. Roussev. Data fingerprinting with similarity digests. In *Advances in digital forensics vi*, pages 207–226. Springer, 2010.
- [130] V. Roussev. An evaluation of forensic similarity hashes. *Digital Investigation*, 8:S34–S41, 2011.
- [131] V. Roussev, C. Quates, and R. Martell. Real-time digital forensics and triage. *Digital Investigation*, 10(2):158–167, 2013.
- [132] N. C. Rowe. Testing the national software reference library. *Digital Investigation*, 9:S131–S138, 2012.
- [133] N. C. Rowe and S. L. Garfinkel. Finding anomalous and suspicious files from directory metadata on a large corpus. In *Digital Forensics and Cyber Crime*, pages 115–130. Springer Berlin Heidelberg, 2012.
- [134] RSA. Current state of cybercrime. 2018.
- [135] R. Russon and Y. Fledel. Ntfs documentation, 2004.
- [136] J. Rutkowska. The implementation of passive covert channels in the Linux kernel. In *Chaos Communication Congress, Chaos Computer Club eV*, 2004.
- [137] J. Rutkowska. Introducing stealth malware taxonomy. *COSEINC Advanced Malware Labs*, pages 1–9, 2006.

- [138] D. Sanger. Shadow brokers leak raises alarming question: Was the nsa hacked. *New York Times*, 27:2016, 2016.
- [139] M. Steiner, T. En-Najjary, and E. W. Biersack. Long term study of peer behavior in the kad dht. *IEEE/ACM Transactions on Networking (TON)*, 17(5):1371–1384, 2009.
- [140] D. Stodle. Ping tunnel. Online at <http://www.cs.uit.no/~daniels/PingTunnel/>.
- [141] M. Suiche. Shadow brokers: The insider theory. *Comae Technologies*, 2017.
- [142] Sun Microsystems Inc. ZFS on-disk specification, 2006.
- [143] U. S. C. I. R. Team. Alert (TA17-132A): Indicators Associated With WannaCry Ransomware, 2017.
- [144] P. S. Teh, A. B. J. Teoh, and S. Yue. A survey of keystroke dynamics biometrics. *The Scientific World Journal*, 2013, 2013.
- [145] The Telegraph. NHS cyber attack: Everything you need to know about 'biggest ransomware' offensive in history, 2017.
- [146] D. J. Tian, A. Bates, and K. Butler. Defending Against Malicious USB Firmware with GoodUSB. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 261–270. ACM, 2015.
- [147] D. J. Tian, A. Bates, K. R. Butler, and R. Rangaswami. ProvUSB: Block-level provenance-based data protection for USB storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, pages 242–253, New York, NY, USA, 2016. ACM.
- [148] D. J. Tian, N. Scaife, A. Bates, K. Butler, and P. Traynor. Making USB great again with USBFILTER. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 415–430, Austin, TX, 2016. USENIX Association.
- [149] M. Tischer, Z. Durumeric, S. Foster, S. Duan, A. Mori, E. Bursztein, and M. Bailey. Users really do plug in usb drives they find. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 306–319. IEEE, 2016.
- [150] Tuxera Inc. Open source: Ntfs-3g. Online at <http://www.tuxera.com/community/open-source-ntfs-3g/>.
- [151] D. Umphress and G. Williams. Identity verification through keyboard characteristics. *International journal of man-machine studies*, 23(3):263–273, 1985.
- [152] U.S. Department of Justice. How to protect your networks from ransomware. 2016.

- [153] J. Von Neumann, A. W. Burks, et al. Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, 5(1):3–14, 1966.
- [154] C. Wang, J. C. Knight, and M. C. Elder. On computer viral infection and the effect of immunization. In *16th Annual Conference on Computer Security Applications (ACSAC)*, pages 246–256. IEEE, 2000.
- [155] Z. Wang and A. Stavrou. Exploiting smart-phone USB connectivity for fun and profit. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 357–366. ACM, 2010.
- [156] S. Wendzel, W. Mazurczyk, L. Caviglione, and M. Meier. Hidden and uncontrolled—on the emergence of network steganographic threats. In *ISSE Securing Electronic Business Processes*, pages 123–133. Springer, 2014.
- [157] A. Williams. Transfer data via YouTube. Online at <https://hackaday.com/2015/08/23/transfer-data-via-youtube/>.
- [158] S. Wolchok and J. A. Halderman. Crawling bittorrent dhds for fun and profit. In *4th USENIX Workshop on Offensive Technologies (WOOT)*, 2010.
- [159] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems*, pages 260–269. IEEE, 2003.
- [160] B. Yang, D. Feng, Y. Qin, Y. Zhang, and W. Wang. TMSUI: A trust management scheme of USB storage devices for industrial control systems. In *17th International Conference on Information and Communications Security*, pages 152–168. Springer International Publishing, 2016.
- [161] J. Young, K. Foster, S. Garfinkel, and K. Fairbanks. Distinct sector hashes for target file detection. *IEEE Computer*, (12):28–35, 2012.
- [162] P. Zaitcev. The usbmon: USB monitoring framework. In *Linux Symposium*, page 291, 2005.
- [163] C. Zhang, P. Dhungel, D. Wu, and K. W. Ross. Unraveling the bittorrent ecosystem. *Transactions on Parallel and Distributed Systems*, 22(7):1164–1177, 2011.
- [164] E. Zielińska, W. Mazurczyk, and K. Szczypiorski. Trends in steganography. *Communications of the ACM*, 57(3):86–95, 2014.

Curriculum Vitae

Work Experience

Security Engineer Google Switzerland, Zurich, Switzerland	December 2017 – Current
Software Engineering Intern Google Switzerland, Zurich, Switzerland	July 2017 – October 2017
Visiting Researcher Northeastern University, Boston, USA	August 2014 – January 2015
Security Researcher SBA Research, Vienna, Austria	June 2011 – July 2017

Education

Doctor of Technical Sciences (Dr. techn.) Vienna University of Technology, Vienna, Austria <i>Doctoral Programme in Technical Sciences</i>	2019
Master of Science (Dipl. Ing.) Vienna University of Technology, Vienna, Austria Major: <i>Security Engineering for Enterprise Environments</i>	2018
Bachelor of Science in Engineering (BSc.) University of Applied Sciences Hagenberg, Hagenberg, Austria Major: <i>Secure Information Systems</i>	2012
Ingenieur (Ing.) Höhere Technische Lehranstalt (HTL) Anichstraße, Innsbruck, Austria Major: <i>Industrial Engineering</i>	2008

Program Committee Service

International Conference on Digital Forensics and Cyber Crime (ICDF2C)	2017
International Conference on Availability, Reliability and Security (ARES)	2016, 2017
International Workshop on Cyber Crime (IWCC)	2016

List of Publications

Neuner, S., Voyiatzis, A. G., Fotopoulos, S., Mulliner, C., and Weippl, E. R. "USBBlock: Blocking USB-based Keypress Injection Attacks" 32nd Annual Conference on Data and Applications Security and Privacy. IFIP, 2018.

Neuner, S., Voyiatzis, A. G., Schmiedecker, M., and Weippl, E. R. "Timestamp hiccups: Detecting manipulated filesystem timestamps on NTFS" 12th International Conference on Availability, Reliability and Security. ACM, 2017.

Kieseberg, P., Neuner, S., Schrittwieser, S., Schmiedecker, M., and Weippl, E. R., "Real-time Forensics through Endpoint Visibility"; 9th International Conference on Digital Forensics & Cyber Crime. EAI, 2017.

Merzdovnik, G., Huber, M., Buhov, D., Nikiforakis, N., Neuner, S., Schmiedecker, M., and Weippl, E. R., "Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools"; European Symposium on Security and Privacy. IEEE, 2017.

Neuner, S., Voyiatzis, A. G., Schmiedecker, M., Brunthaler, S., Katzenbeisser S. and Weippl, E. R., "Time is on my side: Steganography in filesystem metadata"; Digital Investigation 18, S76-S86, 2016.

Neuner, S., Schmiedecker, M. and Weippl, E. R., "PeekaTorrent: Leveraging P2P Hash Values for Digital Forensics"; Digital Investigation 18, S149-S156, 2016.

Neuner, S., Schmiedecker, M. and Weippl, E. R., "Effectiveness of file - based deduplication in digital forensics"; Wiley Security and Communication Networks 9.15, S2876-2885, 2016.

Neuner, S., Mulazzani, M., Schrittwieser, S. and Weippl, E. R., "Gradually Improving the Forensic Process"; International Workshop on Cyber Crime, 2015.

Kadluba, C., Mulazzani, M., Zechner, L., Neuner, S., and Weippl, E. R., "Windows Installer Security"; Sixth ASE International Conference on Privacy, Security, Risk and Trust, 2014.

Neuner, S., Van der Veen, V., Lindorfer, M., Huber, M., Merzdovnik, G., Mulazzani, M., and Weippl, E. R., "Enter Sandbox: Android Sandbox Comparison"; Mobile Security Technologies Workshop. IEEE, 2014.

Mulazzani, M., Neuner, S., Schrittwieser, S., Weippl, E., Kieseberg, P., and Huber, M., "Quantifying Windows File Slack in Size and Stability"; International Conference on Digital Forensics. IFIP, 2013.