Traceless Execution Support for Privacy Enhancing Technologies

Daniela Gorjão Lopes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Nuno Miguel Carvalho dos Santos

Examination Committee

Chairperson: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Supervisor: Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee: Prof. André Ventura da Cruz Marnoto Zúquete

October 2021

Dedicated to my sister,

Acknowledgments

First, I thank my advisor, Professor Nuno Santos, for supporting my ideas and always keeping me motivated and open to explore new challenges. I thank Diogo Barradas for all the guidance throughout this work, and for all the help in finding alternatives when needed. I thank my siblings for accompanying me and all the encouragement given during this journey. Lastly, I thank all my friends with whom I grew and evolved with as a person, specially those who still accompany me today and share the challenges that come, to continue this work of improvement, so that produce better and better outcomes.

Resumo

Muitas tecnologias de proteção de privacidade (PETs) permitem que os utilizadores que vivem sob regimes de censura autoritários realizem atividades online anonimamente. Existem propostas que tentam limpar os vestígios deixados pela execução destas ferramentas. No entanto, elas tendem a deixar padrões no disco, ou exigem componentes de hardware adicionais que podem sugerir o uso de técnicas anti-forenses.

Neste projeto, pretendemos desenvolver uma estrutura segura para a execução de PETs sem deixar rastos que ofereça garantias de negação plausíveis sempre que a máquina do utilizador for submetida a uma análise forense do disco. Apresentamos o Calypso, um sistema que oferece suporte para a execução de PETs numa partição sombra da máquina do utilizador. Uma partição sombra é um espaço de armazenamento anti-forense que "parasita" os blocos livres do disco de modo a ocultar dados confidenciais de forma plausivelmente negável. Isso é feito preservando as características de entropia inicial dos blocos livres, o que deixa margem para o uso de pegadas digitais plausivelmente negáveis que justificam mudanças no disco. Implementámos um protótipo robusto e totalmente funcional do Calypso como um módulo do kernel para a versão 5.4 do kernel do Linux. Avaliámos extensivamente o Calypso, demostrando que ele suporta a execução de vários PETs sem perturbar o sistema, enquanto reduz significativamente os vestígios de PETs deixados no armazenamento persistente. Demonstrámos ainda que o Calypso tem uma grande oportunidade de fazer alterações negáveis no disco, graças às semelhanças de entropia entre arquivos de multimédia e dados cifrados.

Palavras-chave: Tecnologias de Proteção de Privacidade, Pegada Digital, Armazenamento de Negação Plausível, Entropia, Interceção de pedidos, Ataque de múltiplas imagens

Abstract

Many privacy enhancing technologies (PETs) allow users living under strict censorship regimes to perform online activities anonymously. However, many users refrain from executing these tools due to fear of reprisals. Some systems attempt to clean up the traces left by the execution of PETs. However, they tend to leave patterns on the disk or require additional hardware components that may suggest the usage of anti-forensic techniques.

In this project, we aim at developing a secure framework for the traceless execution of PETs that offers plausible deniability guarantees whenever the user's machine is subjected to a full-disk forensic analysis. We present Calypso, a system that provides support for the execution of PETs on a shadow partition of a user's machine. A shadow partition is an anti-forensic storage space that "parasites" the free blocks of the disk to conceal sensitive data in a plausibly deniable fashion. This is done by preserving the initial entropy characteristics of the free blocks, which leaves a margin to the usage of forensic deniable footprints to justify changes in the disk. We implemented a robust and fully-functional prototype of Calypso as a kernel module for the Linux kernel v5.4. We have extensively evaluated Calypso, showing that it can support the execution of multiple PETs without disrupting the system, while significantly reducing the extent of PET traces left on persistent storage. Furthermore, we demonstrate that Calypso has a high opportunity of making deniable changes to the disk thanks to the entropy similarities between media files and encrypted data.

Keywords: Privacy Enhancing Technologies, Digital Footprint, Plausible Deniable Storage, Entropy, Requests Interception, Multi-Snapshot Attack

Contents

Acknowledgments . . . v

Resumo . . . vii

Abstract . . . ix

List of Tables . . . xv

List of Figures . . . xvii

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Challenges . . . 3

1.3 Contributions . . . 3

1.4 Thesis Outline . . . 4

2 Background and Related Work **5**

2.1 Background on Privacy Enhancing Technologies . . . 5

2.1.1 Privacy Enhancing Technologies . . . 5

2.1.2 Case Study: Tor Browser . . . 7

2.2 Digital Forensic Techniques . . . 9

2.3 Anti-Forensic Countermeasures . . . 11

2.3.1 Private Program Execution . . . 11

2.3.2 Deniable File Systems . . . 13

2.3.3 Defending from Multi-Snapshot Attacks . . . 14

2.4 Isolation through Containerization . . . 16

3 Design **19**

3.1 Overview . . . 19

3.2 Threat Model . . . 21

3.3 Traceless Bootstrap . . . 22

3.4 Block Allocation . . . 22

3.5 Monitoring Native Changes to Blocks . . . 24

3.6 Persisting Calypso Meta-Data . . . 25

3.6.1 Persisted Meta-Data . . . 25

3.6.2 Managing Meta-Data Blocks in the Native Partition . . . 26

3.7 Securely Encoding Data in Blocks . . . 27
3.7.1 Entropy and Entropy Differential . . . 27
3.7.2 Block Entropy and Entropy Threshold . . . 28
3.7.3 Multi-Snapshot Attacks and Plausible Deniable Footprints . . . 29

4 Implementation **31**
4.1 Introducing the Linux Kernel Block Layer . . . 31
4.1.1 The Linux Kernel Block I/O Stack . . . 31
4.1.2 I/O Requests and I/O Request Structures . . . 33
4.2 System Requirements . . . 34
4.3 Retrieving Usable Blocks for Storage . . . 35
4.4 Redirecting and Intercepting Block Requests . . . 35
4.4.1 Stacking Calypso on the Native Block Device and Redirecting Requests . . . 35
4.4.2 Hooking the Request Handling Function of the Native Partition . . . 37
4.4.3 Copying Calypso Blocks to Prevent Overwrites . . . 38
4.4.4 Cleaning Caches to Retrieve Updated Data . . . 39
4.5 Cryptography and Data Hiding . . . 40
4.5.1 Hiding Data Blocks . . . 40
4.5.2 Storing and Retrieving Meta-Data . . . 41
4.5.3 Meta-Data Retrieval Algorithm . . . 42
4.5.4 Meta-Data Hiding Algorithm . . . 43
4.6 Entropy and Floating-Point in the Kernel . . . 44

5 Evaluation **47**
5.1 Evaluation Goals . . . 47
5.2 Functionality . . . 48
5.2.1 Metrics and Methodology . . . 48
5.2.2 Results . . . 50
5.3 Isolation . . . 54
5.3.1 Metrics and Methodology . . . 54
5.3.2 Results . . . 55
5.4 Security . . . 58
5.4.1 Metrics and Methodology . . . 58
5.4.2 Results . . . 59
5.5 Performance . . . 63
5.5.1 Metrics and Methodology . . . 63
5.5.2 Results . . . 63

6 Conclusions **67**

Bibliography **69**

A PET Utilization Scripts 75

List of Tables

2.1 Taxonomy on currently used PETs and respective download URLs. 6

2.2 Examples of access files or directories during Tor's installation or execution on a Linux system, organized by relevant directories. 8

5.1 Blocks distribution in native partition being used by Calypso before and after executing each PET's utilization script. 53

5.2 Files accessed and respective classification during 10 minutes of auditing the system without Tor installed, while executing Tor in the native file system, and while executing Tor inside Calypso. 56

5.3 Top 10 files in disk accessed by Calypso when executing the Tor browser, by file type. . . 57

5.4 Statistics on the entropy variation in the blocks of 1 GB virtual disks simulating unused space, for multiple media file types. 60

5.5 Differential entropy statistics. 61

List of Figures

2.1 Residue-free computing architecture. 12
2.2 Artifice's generation of carrier blocks to hide user data. 15
2.3 Container engine. 16

3.1 Calypso architecture. 20
3.2 Calypso driver internals. 21
3.3 Calypso block mappings. 23
3.4 Calypso meta-data blocks structure. 26

4.1 Linux kernel block I/O stack. 32
4.2 I/O request structures. 33
4.3 Calypso request redirection. 36
4.4 Calypso request interception and handling overwrites. 39

5.1 Block utilization evolution with time for Tor browser executed with Calypso. 51
5.2 Block utilization evolution with time for Signal desktop executed with Calypso. 51
5.3 Block utilization evolution with time for Megasync desktop executed with Calypso. 52
5.4 Comparison of a part of the native partition state, before and after executing Megasync with Calypso, from block 200 to block 90000, as a matrix of 300x300. 53
5.5 Average block entropy and standard deviation in the blocks of 1GB virtual disks simulating unused space, for multiple file types. 60
5.6 Entropy differential PDF for multiple entropy threshold values. 62
5.7 Comparison of the latency of the native file system without and with Calypso loaded. 64
5.8 Comparison of the throughput of the native file system without and with Calypso loaded. 64
5.9 Comparison of the latency of the native file system without Calypso loaded and the file system in Calypso's shadow partition. 64
5.10 Comparison of the throughput of the native file system without Calypso loaded and the file system in Calypso's shadow partition. 64
5.11 Comparison of the latency of the file system in Calypso's shadow partition and a ramfs without Calypso. 65
5.12 Comparison of the throughput of the file system in Calypso's shadow partition and a ramfs without Calypso. 65

Chapter 1

Introduction

This thesis addresses the problem of the digital footprint left by the execution of privacy enhancing technologies (PETs) in users' machines. Users may face penalties for performing actions against repressive regimes when these actions can be observed through the forensic inspection of a machine's persistent storage. Therefore, it is important to provide users with a solution that helps them practice such activities without leaving persistent traces. To tackle this problem, our idea is to "parasite" on the free blocks of the native file system by deniably encoding in these blocks all the data involved in the execution of PETs. Informally, deniability can be defined as the ability of a user to deny having performed some activity. To achieve this property, we use entropy and forensic deniable footprints as decoy mechanisms to unlink the changes to the disk blocks from suspicious activities. Entropy measures the average level of unpredictability of a variable's outcome. In storage, it can be used to analyze the characteristics of the bytes stored in free space to determine the types of files erased. Lower entropy values may indicate the presence of files with less variability such as ebooks and other plaintext files, whereas higher entropy values are associated with compressed file types such as images or encrypted content. Forensic deniable footprints are sequences of steps that make changes to disk, mimicking the regular behaviour of the system or the user, thus justifying specific changes to disk. These range from activities like installing and uninstalling programs to streaming movies. This thesis introduces a practical solution to the aforementioned problem, without interfering with the regular operation of PETs or the system.

1.1 Motivation

Nowadays, Internet users face an increasing deterioration of privacy and freedom of speech. This situation is worsened by the rise of censorship and repressive regimes[1]. To preserve the right for users to freely access and publish content, several privacy enhancing technologies have been developed, which are commonly referred to as PETs. These applications target several use cases, such as circumventing censorship or preserving user anonymity within a network.

Despite of the wide availability of PETs, existing software still imposes several drawbacks on its

[1] https://www.lowyinstitute.org/publications/digital-authoritarianism-china-and-covid#_edn62

users. On the one hand, many tools tend to[2]: i) be hard to use by less technical users [1]; ii) require a considering amount of computation and storage resources in the local machine; and iii) require maintaining non-trivial configuration meta-data and cryptographic artifacts, such as keys. This means a user with little knowledge or computational power might not be able to properly take advantage of these tools, or choose not to use them if faced with circumstances that pose a threat to their life. To make matters worse, these tools may leave an extensive footprint on the machine, making them inappropriate for someone who may have the device periodically inspected, such as when crossing border controls in countries controlled by repressive regimes. Some of these countries do not make censorship-related laws explicit, which makes people insecure about which actions can lead to legal punishment. This creates a sense of social non-acceptance and self-censoring [2].

In fact, several studies are showing the forensic implications of using PETs on a machine, such as artifacts from privacy-enhanced web browsers [3, 4]. For instance, the Tor Browser[3] relies on numerous persistent files and configurations to preserve their functionality, usability and security. These changes leave a persistent footprint that can be easily identified by a forensic investigator using standard forensic analysis tools. Thus, we argue that users should have the possibility of installing and executing PETs without making observable changes to the persistent state of their machine so that they can plausibly deny having performed such activities.

Despite the extensive documentation on censorship techniques applied at the network level [5–7], there is a general lack of research regarding the forensic inspection of devices conducted by state-level censors. Censors may resort to state-of-the-art forensic techniques during their quest to uncover the activity of PETs in citizens' machines. Several existing approaches allow users to evade this kind of forensic analysis, e.g., steganographic file systems [8, 9]. However, these techniques cannot fully hide the anti-forensic system itself and are not designed to emulate the data entropy levels that would reflect the state of a disk before the execution of a PET tool. Consequently, they would make users vulnerable to disk entropy analysis techniques. So, for instance, if all the unallocated disk space is used up for storing the encrypted data and meta-data of an anti-forensic file system, a disk forensic analysis would immediately reveal an unusual presence of high-entropy disk blocks. Alternatively, some solutions can preserve the entropy levels [9, 10], but require additional hardware for file system isolation and security. Likewise, other projects [11][4] are very intrusive and perform many changes to the system. Some of these changes are quite unconventional, such as the creation of partitions schemes, modification to official program binaries, or extending the operating system.

Our goal is to build a system that allows users to execute PETs without leaving persistent traces on local disk storage. It aims to provide *plausible deniability* for the user's activities when subjected to forensic disk analysis while maintaining the usability, functionality, and efficiency of PETs. Our approach consists of creating a *shadow partition* to execute PETs and redirecting every access that may cause observable persistent changes to the blocks that constitute this partition, where the information is encoded in a deniable way. The encoded data in the shadow partition should only be available to

[2] https://www.adalovelaceinstitute.org/blog/privacy-enhancing-technologies-not-always-our-friends/
[3] https://www.torproject.org/
[4] http://truecrypt.sourceforge.net

a user possessing the right secret, who can then access data through the usage of existing steganographic and cryptographic techniques while maintaining the entropy of the disk and resilience of hidden data and maximizing storage capacity. Although we propose a solution that can be generally applicable across different operating systems and file systems, our implementation will be targeted to Linux platforms whose native partitions are formatted to the Ext4 file system.

1.2 Challenges

To develop a solution that can create a shadow partition on a user's computer, we need to overcome several challenges. Keep in mind that a shadow partition will take advantage of the free blocks in the native partition where it can store data and execute programs in a plausibly deniable way.

For instance, an important challenge involves retrieving and keeping track of the blocks to be used for the shadow partition. In particular, an initial issue relies on selecting native blocks of the Ext4 file system that will not disrupt the regular operation of the Linux kernel, operating system services, and applications. We need to allocate these blocks on-demand to accommodate the progression on the contents of the shadow partition, as well as manage the blocks that are already allocated. Since we are building this abstraction layer on top of the native partition, we will need to handle the mappings that support this. Then, we need to ensure we do not lose data that affects the functioning of the shadow partition, which can happen because the used blocks belong to the native file system. This means they can be changed at any time, raising complex technical challenges.

Another class of challenges is about encoding the data of the shadow partition in the native free blocks in a way that will not be observable to an adversary. Encoding data in the native blocks will produce changes in the disk. Making these changes non-observable entails determining the best way to encode the data and in which blocks is it safe to make the necessary changes, while accounting that the storage capacity of the shadow partition is limited. Additionally, only the rightful user should be able to retrieve the encoded data correctly, so we need to resort to the usage of cryptographic primitives.

Finally, all these challenges must be solved within the Linux kernel environment, which brings additional implementation difficulties that usually do not occur when developing userland programs. We highlight the complexity of the Linux kernel block I/O stack and its data structures which makes it hard to incorporate our changes without disrupting the system.

1.3 Contributions

To overcome the aforementioned challenges, this thesis proposes, implements, and evaluates techniques to map and encode blocks while hooking the native I/O request processing to trick the native file system into hiding data in its free blocks. Specifically, this thesis makes three core contributions:

- Presents the design of Calypso, a system that supports the traceless execution of PETS through the abstraction of a shadow drive that uses free disk blocks to support concealing a steganographic

file system. This way, users can store sensitive data and execute programs deniably, allowing private sessions with persistent state, while maintaining their functionality and performance.

- Implements a robust, fully-functional prototype of Calypso as a kernel module for the Linux kernel version 5.4.
- Delivers an extensive analysis that shows Calypso supports multiple PETs without affecting the system's operation or performance while reducing the extent of PET traces on storage. On top of this, Calypso has a high opportunity of making deniable changes to blocks due to the entropy similarities between media files and encrypted data.

This produces the following results: i) a full design specification of the Calypso system; ii) an implementation of Calypso for Linux systems; iii) an extensive experimental evaluation of the functionality, isolation, security and performance.

1.4 Thesis Outline

The rest of this document is organized in the following way. Chapter 2 starts by providing an overview of existing PETs and presents a case study on the Tor browser. Then, it describes the main body of literature related to our work, such as techniques to investigate digital traces and countermeasures, exploring systems with similar goals. Chapter 3 describes the design and proposed architecture of Calypso, while Chapter 4 details the implementation challenges of our prototype. Chapter 5 shows and analyses the results of the experimental evaluation conducted on Calypso. Lastly, Chapter 6 concludes this document by summarizing our main findings and presents suggestions to direct our future work.

Chapter 2

Background and Related Work

In this chapter, we provide some necessary background on existing PETs, offering also an empirical analysis of the forensically persistent traces left by Tor: a popular PET application for anonymous Internet communications (Section 2.1). From our analysis of Tor, we observe that these operations are extensive and hard to contain due to their diversity, making them observable during a forensic inspection. We then cover the related work. First, we present state-of-the-art forensic techniques that aim at analyzing software execution traces in a system (Section 2.2). Then we survey several anti-forensic techniques that prevent the analysis of digital artifacts from computers (Section 2.3). Lastly, we discuss potential approaches based on containers for improving isolation of PET execution (Section 2.4).

2.1 Background on Privacy Enhancing Technologies

In this section we start by providing an overview of existing PETs in use, both deployed and still in research phase, according to relevant categories already established by existing taxonomies. Then, we delve into Tor, one of the most well-known and used PETs. We further present a case study on the impacts that installing and executing these technologies can have on a user's machine.

2.1.1 Privacy Enhancing Technologies

PETs aim at achieving specific privacy or data protection functionality "by minimizing personal data use, maximizing data security and empowering individuals"[1] against the risks of privacy loss, possibly caused by identity disclosure or linking data traffic with identity. However, existing PETs do not mitigate all privacy harms alone as shown by Solove *et al.* [12]. In particular, they are not designed to withstand a forensic inspection. Thus, additional mechanisms are needed to make the execution of these tools robust against physical attacks and avoid their detection through analysis of the persistent storage.

In this section, we focus on surveying several tools that both provide censorship resistance and require a client-side application since these are expected to perform persistent changes to a user's machine. Below, we consider five main categories. We identified these categories based on prior

[1] https://en.wikipedia.org/wiki/Privacy-enhancing_technologies

Category	Name	URL
Secure messaging	Signal	https://signal.org
	Whatsapp	https://www.whatsapp.com
	Telegram	https://telegram.org
	Threema	https://threema.ch
	Wickr	https://wickr.com
	RetroShare	https://retroshare.cc
	Silence	https://silence.im
Virtual private networks	Psiphon	https://psiphon3.com
	Lantern	https://getlantern.org
	Bitmask	https://bitmask.net
	Riseup	https://riseup.net/vpn
	Wireguard	https://www.wireguard.com
	Mullvad	https://mullvad.net
Anonymizing networks	Tor	https://www.torproject.org
	I2P	https://geti2p.net
	Freenet	https://freenetproject.org
	Zeronet	https://zeronet.io/
Anti-tracking browsing	Ghostery	https://www.ghostery.com
	Disconnect	https://disconnect.me
	uBlock origin	https://github.com/gorhill/uBlock
	Privacy Badger	https://privacybadger.org
	NoScript	https://noscript.net
	AdBlockPlus	https://adblockplus.org
	Firefox's Private Browsing	https://www.mozilla.org
Digital footprint cleaning	Tails	https://tails.boum.org
	BitRaser	https://www.bitraser.com
	VeraCrypt	https://www.veracrypt.fr

Table 2.1: Taxonomy on currently used PETs and respective download URLs.

taxonomies proposed by other authors [13–15]. For each category, we present examples (and source URLs) of readily available PETs and others that are still in the research stage. Table 2.1 presents some examples of PETs according to each category:

1. Secure messaging: This class of PETs focuses on providing secure communication through instant messages, among two or more communicating parties. Generally, they prevent unauthorized third parties from accessing the contents of the communication. These commonly rely on the use of cryptographic protocols and algorithms to support end-to-end encryption.

2. Virtual private networks: VPNs can hide the user's real IP address in such a way that an outside observer can only see the IP address of the VPN service provider and bypass IP-based internet censorship. Their main limitation lies in the fact that the VPN provider is a point of failure since it can identify the user and respective online activity. Besides this, VPNs do not bypass more sophisticated censorship methods, so this may require using a combination of other services or tools. They may also be blocked and affect network performance. Protozoa [16] is an example of a research work that implements covert channels on multimedia streams on top of WebRTC, almost indistinguishable from legitimate streams. Similar to regular VPN, Protozoa connects itself to another machine, using it as a hop to access censored content.

3. Anonymizing networks: This special kind of network is designed to anonymize Internet communications to make it hard to link the communication parties. For instance, a third-party adversary should not be able to see determine which web pages a certain user is accessing. The main focus of anonymizing networks is to provide user anonymity, i.e., the property of not being identifiable within a set of potential subjects known as anonymity set. Generally, these tools are not infallible and may have some limitations in regards to providing full anonymity protection. Besides, they use protocols that can be easily fingerprinted and blocked, which may require combining them with other security mechanisms. They may also cause performance degradation. Some recent research projects designing such systems are Riffle [17] and Vuvuzela [18].

4. Anti-tracking browsing: Tools that aim to prevent the collection of information about an individual over time, including sensitive data. Some tracker mechanisms are cookies or more advanced device fingerprinting. Anti-tracking tools can have side effects like limiting the user's access to particular websites or content and generating performance issues such as slowing down page loading. Browser extensions in particular can make fingerprinting easier.

5. Digital footprint cleaning: These are tools that aim to erase or hide traces of program execution or file-related artifacts locally on a machine. It is relevant to note the work of Bock *et al.* [19], a purely server-side censorship evasion technique that does not require the user to download and run anti-censorship software, or even realize that they are being censored, by using packet manipulation strategies for evading nation-state censors.

2.1.2 Case Study: Tor Browser

Given the PET examples presented above, we selected Tor to perform an empirical study on the effect that installing and executing this software has on the persistent state of a system. We choose the Tor browser since it is a very complex and widely used tool for anonymity protection in the service of free speech. Some work has been done on the analysis of traces left by the Tor browser on Windows [20, 21] and additional, albeit outdated work, covering the three main OSs [22]. However, there is no up-to-date exhaustive forensic analysis on Linux. Hence, we focused our empirical study on Linux systems.

Methodology: Assuming Tor browser binaries already exist on our testbed virtual machine (VM), we created five different snapshots of the VM to capture its state: i) before installing Tor browser; ii) after having Tor browser installed; iii) during the execution of the Tor browser; iv) after executing the Tor browser; v) after uninstalling the Tor browser according to the recommendations on the website. Afterward, we performed *differential* analysis of the five stages to identify relevant differences between snapshots caused by installing and executing the Tor browser. We obtain the accessed files by processes using Volatility[2], as well by *diffing* the main accessed directories in the system and looking for information related to Tor, besides auditing and logging accesses to files using Linux Audit Framework [23]. We leveraged multiple Linux distributions that are widely used.

Evaluation of persistent traces: Table 2.2 shows examples of accessed files. We take into consid-

[2]`https://www.volatilityfoundation.org/`

Directory	Meaning	Examples
/home	User's files	.bash_history
/home/tor-browser	Tor main directory	Browser/.cache; Browser/.../Data/*; start-tor-browser.desktop
/home/.tor-browser	Tor hidden files	app/.../firefox.real; app/.../.config; app/.../.cache
/home/.local	User data and program state	share/.../application_state; share/.../tracker-store.journal
/home/.config	Configurations	pulse/cookie; IBus
/home/.cache	Non-essential data	mozilla/firefox/*; fontconfig/*
/dev/shm	Shared memory	org.chromium.*; org.mozzila.ipc.*
/etc	Host-specific configuration	NetworkManager/NetworkManager.conf; resolv.conf
/opt	Add-on apps	apparmor.d/disable; tor-browser/*
/run/user/1000/gnupg	Files by running processes	S.dirmngr; S.gpg-agent.browser; S.gpg-agent; S.gpg-agent.ssh
/tmp	Temporary files	firefox; mozilla_user0
/usr/share	Architecture independent	pixmaps/tor-browser.png; applications/tor-browser.desktop
/usr/bin	Executable commands	tor-browser; dirmngr; gpg-agent
/usr/lib	Libraries and packages	firefox/firefox; gvfsd-network; systemd/systemd-timesyncd
/usr/sbin	System binaries	NetworkManager
/var/log	Diverse log files	journal/*; kern.log; auth.log; syslog

Table 2.2: Examples of access files or directories during Tor's installation or execution on a Linux system, organized by relevant directories.

eration that read accesses also update the timestamps of a file's meta-data during Tor's installation or execution. We organized them by relevant directories since these might imply different traces left, regarding whether they are temporary files, files shared with other applications such as libraries, system logs or hidden files.

After analysing the table, we can conclude that there is a multitude of persistent traces left by installing and executing a PET such as the Tor browser. The main outcomes are: i) the existence of several temporary files spread across multiple directories, which means these will stay in non-allocated space until that space is overwritten, unless the user executes secure software removal, which may also leave observable traces; ii) it accesses several files and libraries that are shared with other applications, making it very difficult to isolate the accesses; and iii) hidden files are not going to be erased when the application is erased from the system, and a regular user might not notice them. Thus, even if Tor is erased every time after being used or booted from a live system, it is very hard to prevent a forensic analyst from being able to observe that Tor was indeed in that system, through techniques such as i) file carving to recover files from unallocated space; ii) analysis of patterns in the access of files by examining timestamps of files that are already known to be frequently accessed by the application; or iii) through direct observation of the files in the system, such as journaling and history files like *bash_history*.

Tor inevitably needs to store persistent files: Since the Tor browser is a very complex application, it is expected that it needs a big set of code files, libraries and saved state. Now, we are going to weigh whether these files are indispensable to maintain Tor's usability. The main Tor directory is not explicitly required for executing the Tor browser, which means the user could delete those files every time or configure Tor not to update them and transfer them again to the machine every usage, however, that lowers Tor usability and increases the attack surface. Tor [24] caches files according to the current Tor

Directory Protocol[3], and they are important because they maintain server descriptors with information on Tor relays, avoid a possible partition of clients from downloading a malicious set of network status documents and contain sensitive data like identity keys of directory authorities that need to be stored offline, speeding up the Tor node's reintegration in the network. However, not only Directory Protocol related files are persistent, there are also Firefox databases, configurations, keys and more.

Despite PETs serving various purposes in the fight for the freedom of speech, they inevitably require accesses to the file system, thereby leaving traces in persistent storage that are not easily covered. From our analysis of the accesses Tor makes to the file system, we concluded that these operations are extensive and hard to contain due to their diversity, making them observable during a forensic inspection.

In this section, we present state-of-the-art forensic techniques that aim at analysing software execution traces in a system. Such traces can be used by adversaries as supporting evidence for the past execution of PETs on the user's computer. Then we present anti-forensic techniques which put obstacles to the forensic tools presented in Section 2.1, hampering the analysis of digital artifacts located in raw memory or persistent storage. Lastly, we discuss container isolation techniques that isolate multiple environments on the same machine.

2.2 Digital Forensic Techniques

To analyse traces of program execution, we need to have available a set of tools to comprise the multitude of interactions a program can make that change the state of the system. We begin by exploring the tools that analyse these interactions in general. Then we can dive deeper into tools that analyse volatile or persistent traces in greater detail and also understand the current difference in development that exists between the last two. At last, we present some recent forensic analysis that describes the kinds of artifacts that can reveal program execution.

Versioning, logging and monitoring: First of all, several techniques need to be employed to determine which traces need to be hidden from a forensic investigation. For this, we need to analyse the differences in the system caused by installing or executing certain programs that the user may want to hide. Ext3cow [25] is a file system built on Ext3 that provides versioning, snapshot and auditability through "a time-shifting interface that permits a real-time and continuous view of data in the past".

Other systems apply logging and replay to recover from attacks. ReVirt [26] encapsulates the target system inside a VM and the Virtual Machine Monitor (VMM) kernel and hooks add log records with sufficient information to repeat an execution of the VM with instruction granularity. ReTrace [27] also takes advantage of virtualization to capture a minimal amount of information necessary to recreate an execution trace, optimizing time and space by capturing only non-deterministic events and uses deterministic replay to compress large trace files. Accountable Virtual Machines [28] audits the execution of a software system through logging and comparing to a known-good execution to check if the software is behaving as intended.

[3] https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt

VSyscall [29] uses system call interposition and virtualization to regulate and monitor program behaviour. Others like Eidetic Systems [30] track dependencies among groups of processes, enabling the analysis of the information flows, thus making it possible to regenerate information between groups, and allowing to replay only some subsets of processes instead of the whole system.

Memory and kernel analysis: Now we present the state-of-the-art tools that focus mostly on memory forensics. Although the focus of our work is on persistent trace analysis, memory analysis is relevant for various use cases, for instance, to deal with malware detection. In our present discussion, we provide a brief overview of some existing memory forensic techniques and tools.

Analyzing raw memory is very challenging due to the difficulty in parsing current kernel data structures. Isolating the programs that we want to analyse in a VM and allowing for access interception through the VMM is a standard way of analyzing program execution. So, Virtual Machine Introspection (VMI) can be a valuable forensic tool since it allows access to the memory of a running VM or from a physical memory snapshot. LibVMI [31][4] is an example of a library for VMI that offers integration with Volatility[5] framework for memory analysis. ProbeBuilder [32] is designed to support relevant information carving out of the guest memory, since VMI is only designed to access the memory, not to interpret it. MACE [33] performs memory analysis on closed-source OSs like Windows to obtain a more detailed view of the kernel data structures.

Other systems [34] aim at improving memory analysis through automatically generated heuristics instead of relying on humans to decide which part of the information is extracted from memory dumps, by building a graph of kernel objects that are tightly connected and also define a quality metric to guarantee the run-time structures are optimally traversed.

Persistent data analysis: These techniques focus on the analysis of the data that can be extracted from storage devices. These techniques can be divided into two major categories: *file system* or *disk analysis*, and *file carving*.

In the first category, we found the most notable to be The Sleuth Kit[6], which is an open-source library and collection of utilities for extracting data from file systems or disk raw images. These utilities perform analysis at several layers: i) file system, which allows analyzing the partitions of a disk and presents details such as last mounting time; ii) block level, that consists of the actual file content of the unallocated space; iii) meta-data, which describes files or directories; iv) file layer, which allows to display deleted files or find a file by its hash in case the name was changed, among others. Succinctly, it can be used to recover and summarize deleted files, search files by name or included keywords, or recover meta-data. It supports the most used file systems such as NTFS, Ext4 and FAT. Autopsy[7] is a graphical interface to The Sleuth Kit, capable of automating procedures and providing additional features such as integrity checks.

Garfinkel [35] developed a program for automatic disk forensic processing, based on The Sleuth Kit, XML and Python, that describes all of the partitions and files on a disk, as well as corresponding

[4] http://libvmi.com/
[5] https://www.volatilityfoundation.org/
[6] https://www.sleuthkit.org/sleuthkit/
[7] http://www.sleuthkit.org/autopsy/

meta-data. Aziz *et al.* [36] proposed methods of extracting and analyzing data from smartphone devices using The Sleuth Kit Autopsy. Later, Hilgert *et al.* [37] proposed extending The Sleuth Kit to support analysis of file systems with pooled storage, such as ZFS, a file system for large-scaled storage, cloud and virtualization environments. This file system model allows combining multiple volumes to form a pool, which can be accessed by multiple file systems.

Concerning file carving, it is a technique aimed at reconstructing deleted files from contiguous, unallocated disk space. For instance, Foremost[8] recovers files based on their headers, footers and internal data structures described in a configuration file. Similarly, there is Scalpel[9], which works based on patterns described in a configuration file that defines certain file types with binary strings or regular expressions. Bulk extractor [38] presents additional features such as processing compressed or corrupted data and creating histograms with the most common words found within the data, such as email addresses and URLs. On the other hand, EVTXtract[10] focuses on recovering Microsoft Event Logs, which are more complex cases due to their dependencies.

Program execution artifacts: Forensic analysis produces artifacts that may enable uncovering traces of previous actions in a computer. For instance, recent papers [39] show how Graphical User Interfaces (GUI) such as the Gnome Desktop Environment (GDE) can produce artifacts that can help in a forensics examination. It summarizes important files and what kind of artifacts can be found. Other studies [40] show how caching of apparently innocuous information such as *favicons* in modern browsers can be used to build utilization profiles to identify users across sessions, even with incognito mode activated.

2.3 Anti-Forensic Countermeasures

Anti-forensic techniques are designed to counteract forensic analysis by employing data hiding, artifact wiping, and trail obfuscation strategies. Some data hiding examples are encryption, steganography, or placing data in unused storage locations (e.g., the slack space of a disk partition).

The systems presented below focus on reducing the digital footprint caused by the execution of other programs. We start by presenting solutions that focus on enhancing the privacy of specific programs by isolating processes and concealing their data from the rest of the system without providing plausible deniability, culminating in the latest one, Residue-free computing. Then, we delve deeper into persistent systems with protections against *single-snapshot attacks*. Finally, we present the works focused on mitigating *multi-snapshot attacks*, highlighting Artifice.

2.3.1 Private Program Execution

These systems aim to provide privacy to program execution, by isolating and concealing all data generated by a program but are not designed to provide plausible deniability or protect a censored user against a physical inspector. Some systems introduce the concept of *private sessions* to provide pri-

[8] http://foremost.sourceforge.net/
[9] https://github.com/sleuthkit/scalpel
[10] https://github.com/williballenthin/EVTXtract

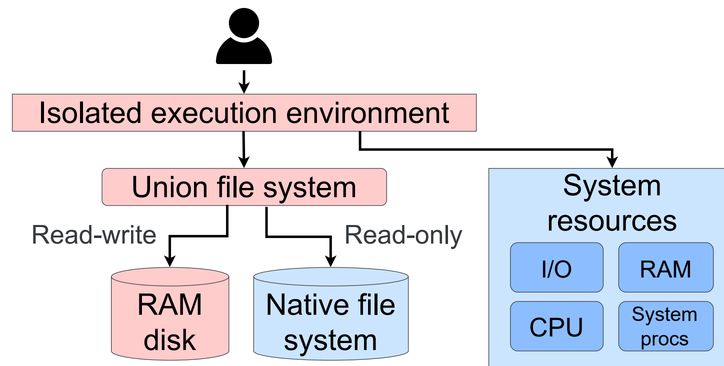

Figure 2.1: Residue-free computing architecture.

vacy concerning to the execution of specific programs on a computer by monitoring the digital trails saved in memory after the program is closed. For instance, Lacuna [41] creates ephemeral channels to avoid data leakage in communications with peripheral devices and runs private sessions inside a VM with an adapted kernel to protect applications that require many executables to communicate through Inter-Process Communication (IPC) like most recent web browsers. Man-in-the-machine [42] thoroughly analyses the privacy attacks that applications using IPC can suffer in the case of multi-user computers.

CleanOS [43] targets privacy loss in mobile devices in case the device gets lost or stolen. It features an Android OS that manages sensitive data rigorously with the use of trusted cloud-based services to evict unnecessary data from RAM and persistent storage, data that was otherwise hoarded by the application. Cloud is necessary since devices are not equipped with physical security. However, no system aims at creating lightweight private sessions that can execute using the native OS, focusing on usability and performance. Ultimately, these systems are not deniable by design.

Other solutions, such as PrivExe [44], create an OS service that provides a key to groups of processes that wish to execute privately. This key is used to encrypt all writes to the file system, as well as processes memory pages written to swap devices, and is discarded at the end of the session. In addition, it restricts IPC and employs containers to prevent leaks to public processes and is implemented by extending the Linux kernel. TpriVexeC [45] is an improvement in performance by using memory mainly. These projects present interesting concepts to incorporate in future work, however, extending the Linux kernel has the major drawback of being easily detectable by checking the kernel integrity against the corresponding official version. Besides this, encryption is used indiscriminately, which can produce undeniable changes to the disk.

Residue-free computing: The main goal of residue-free computing [46] is to isolate all interactions made by a program to memory only, preventing them from making persistent changes to the disk. This isolation is achieved through a Docker container. Figure 2.1 shows the main components that interact with an application executed in a residue-free session. Pink represents the residue-free components that can have access the application's data, while blue represents the native components that should not account for traces of the application's execution. Residue-free computing comprises three main components: an encrypted RAM disk, a union file system and an isolated execution environment. The RAM disk and the execution environment will both contain information from the private session that must

not be disclosed. The union file system combines the data in the RAM disk with the data in the native file system, allowing the application running in a residue-free session to have the illusion of free read and write access to the native file system. The union file system must have read-only access to the native file system. The isolated execution environment provides the interaction with the user and limits the accesses to the union file system.

This system has several shortcomings. First, it has the immediate disadvantage of not supporting persistence. This way, users cannot obtain progress on multiple sessions and applications cannot preserve meta-data that enhances the features of a given program. Additionally, it does not support the storage of sensitive files, which might be the motivation of several users.

Another downside is that the solution presented does not provide plausible deniability, since there is no attempt to hide the existence of this program and the other programs to be executed with its support. This is acceptable for users that only want to protect their sessions from an abusive cohabitant, for instance, but it is not appropriate in the context of censorship resistance in which users may be endangered or suffer penalties for the attempt to hide data from authorities.

Swapping is not prevented but the swapped data is encrypted, which does not disclose the contents of the private session, but still represents evidence of the attempt to hide application sessions.

Finally, the isolation performed with Docker containers depends on the characteristics of the application so that the right native resources can be allowed, causing a lack of support to some. Accessing the native resources will also cause changes, even at the file system level, even though it was specified that logging should be disabled.

2.3.2 Deniable File Systems

These systems are mostly concerned with deniable data hiding on persistent storage and preventing single-snapshot attacks. These attacks describe an adversary with access to a device at a single point in time. The first proposed scheme for a steganographic file system [47] lacked an implementation but inspired many authors of the following systems. It required the name of a file and a password to properly retrieve the system and an attacker that did not know such combination could not know whether such a file is present in the system.

A second scheme consists of storing blocks of the hidden file system within the unallocated space of the native file system. A variation of this design was implemented as StegFS [8], which comes to replace existing encryption systems with steganography for deniability. It is implemented as a file system driver in the Linux kernel 2.4 and uses a block allocation table to map encrypted data to unused blocks and a bitmap to track whether each block is free or allocated. However, there is no effort to hide these data structures. The authors argue that a user can disclose a less important hidden file and keep the others hidden. StegFS does not preserve the disk's entropy by writing random patterns to the hidden blocks, relying on weaker factors to increase deniability, such as executing on a multi-user environment. Also, it replicates blocks to achieve data resilience, which is not space-efficient. Ultimately, it coexists with the native file system in the same OS environment which can cause data leakage.

Other systems like Rubberhose[11], TrueCrypt[12] and VeraCrypt[13] use on-the-fly-encryption (OTFE) to provide easy-to-use volume encryption. However, none of these systems provides undetectable changes to the user's disk nor hide a user's capability of running the system from an adversary. Thus they do not entirely provide a secure deniable file system for traceless PET execution.

2.3.3 Defending from Multi-Snapshot Attacks

This class of systems focuses on multi-snapshot attacks prevention. These attacks encompass an adversary with access to a device at multiple points in time, thus can observe modifications that can indicate the existence of hidden data. The same authors from StegFS [48] proposed using oblivious RAM (ORAM) techniques to conceal data accesses but does not hide its presence. HIVE [11] and Datalair [49] propose a write-only oblivious RAM implemented as a Linux kernel block device. However, ORAM incurs in significant performance penalties, besides random disk write patterns and slower performance might indicate suspicious activity.

PM-DM [50] is a block device mapper that tries to improve on the performance implications of using ORAM techniques suffered by the previous attempts to hide the access patterns. For this, instead of focusing on generating random access patterns, it attempts to perform plausible modifications between snapshots, preserving data locality and performance. These plausible modifications consist on executing a structure of processes, such as an algorithm, that produces a trace of accesses in each snapshot, which empowers the user to plausibly deny hidden accesses by stating the observable changes were made to public volumes. However, the performance of write operations is still low enough to flag the existence of a hidden system.

Ever-Changing Disk *et al.* [10] was proposed as firmware design for writing data separately into hidden and public volumes of an SSD, the hidden data is written in combination with pseudorandom data in log format but the partition scheme and firmware constitute weaknesses to the design.

DEFY[14] was presented as a log-structured file system for mobile devices but does not protect against hidden data overwrite unless hidden volumes are constantly mounted. It requires all file system metadata to be stored in memory and does not prevent swapping-related data leakage.

Another system is Artifice [51], which we highlight separately below due to its relevance considering the similarity of the technical approach underpinning this system in relation to our work.

Artifice: More recently, Artifice [51] has been proposed as a deniable file system that prevents multiple snapshot attacks. It disguises data accesses to disk with deniable operations such as redundant writes. Artifice is meant for cases where malicious software is installed by an adversary to continuously track down user activity, or when the adversary has access to the machine on two separate occasions, for instance when passing through airport security. This system relies on combinatorial cryptography to protect data from being correctly retrieved by an adversary with no knowledge of the password. Additionally, it has good data resilience capabilities due to the use of erasure codes to make overwrites recoverable,

[11] https://web.archive.org/web/20100915130330/http://iq.org/~proff/rubberhose.org/
[12] http://truecrypt.sourceforge.net
[13] https://www.veracrypt.fr/en/Home.html
[14] https://calhoun.nps.edu/handle/10945/46375

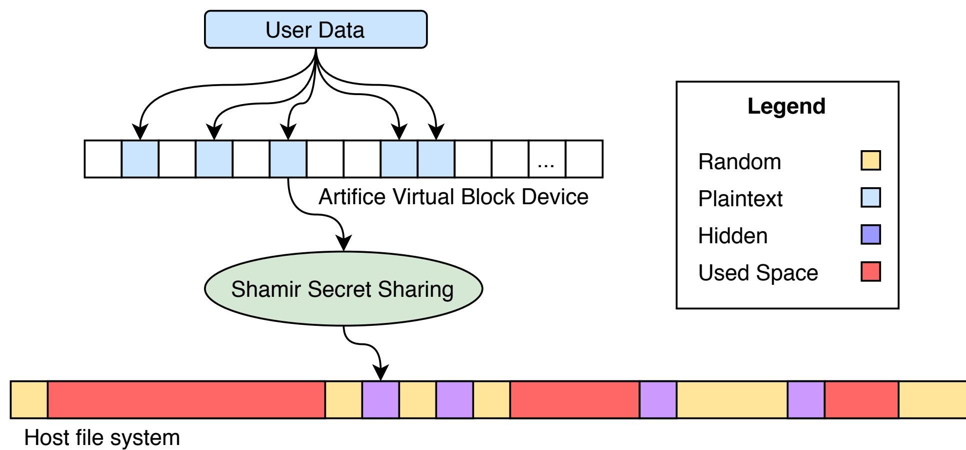

Figure 2.2: Artifice's generation of carrier blocks to hide user data. The block device is contained on removable media. Free space in the host file system is filled with pseudo-random blocks.

along with *checksums* and replication of the Artifice Map: the data structure containing the meta-data required to reconstruct the stored information while being space efficient.

Artifice inhibits an adversary from being able to distinguish the carrier blocks from random data but does nothing about the greater amount of random data present in the disk after the system is installed. So if a user passes an airport security control for the first time without Artifice installed, and then passes a second time with Artifice hiding several files in the same disk, the disk's entropy would reveal that information is being hidden on the disk. It provides isolation from the host OS by booting into an Artifice-aware OS through a patched Linux live USB that loads a kernel module containing a virtual block device driver. However, carrying an external hardware component increases the danger of exposing the system to an attacker. It also enlarges both memory and disk footprint by forcing reboot since most systems track these changes for important health management purposes such as tracking down the source of a problem in the system. There is still one more external hardware dependency to store the entropy blocks to be combined with data blocks separately but it is not explained in detail. Artifice offers no protection in the case of swapping and computer hibernation which can cause information to be written down to disk.

The same system is described in more detail in a follow-up paper [9], where the authors point out that external hardware dependencies need to be eliminated. They also present Shamir Secret Sharing as the algorithm to be used to distribute the carrier blocks so that no visible difference exists between other unused blocks in unallocated space, since the last are filled with pseudo-random blocks. The process of hiding data in the user's disk and the increase in random data is described in Fig. 2.2. This makes it harder for an attacker to brute force an attempt to reconstruct the Artifice volume but it completely changes the entropy of the disk, so the system's existence in a user's system stops being deniable. It also describes a possible improvement for data overwrite that consists of identifying "busy" locations in the native file system and placing Artifice writes in blocks that are least likely to be used in the future.

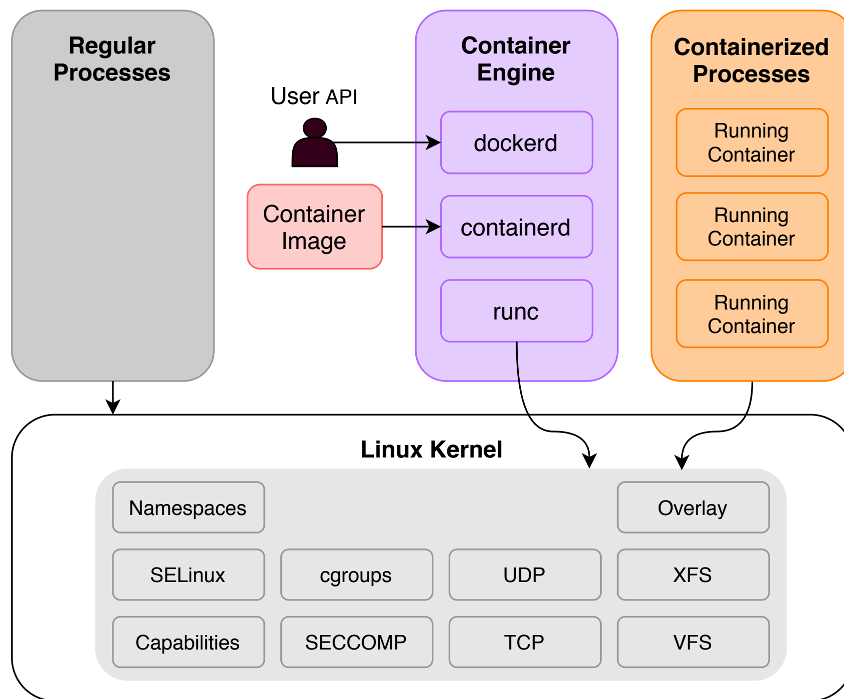

Figure 2.3: Container engine.

2.4 Isolation through Containerization

In this section, we provide related work on how containers can be used to further improve the isolation of mechanisms that support traceless program execution, one of our future ambitions. Containers are an abstraction at the application layer that includes everything needed to run an application. Containers share the host OS kernel, which makes them more lightweight than VMs, both in virtualization overhead and resource usage [52–54]. The popularity of containers has increased with tools such as LXC[15], Docker [55] and Kubernetes[16].

A more detailed view of a Docker container engine is described in Fig. 2.3. As shown, the container engine is composed by three daemon processes, *dockerd*, *containerd* and *runc*. Together, they provide an interface for the user to interact and manage the lifecycle of containers. They also transform container images into running containers by interacting with the kernel, which supports several security mechanisms that can be leveraged to further secure containers.

However, due to kernel sharing, containers provide less isolation than full virtualization and suffer from a greater attack surface, which makes isolation and sandboxing with containers a much more complex task. Next, we present relevant security vulnerabilities and isolation problems in current container solutions. Then, we dive into the security-enhancing mechanisms provided by the Linux kernel and show how in certain situations, they are not enough, either due to lack of customization or because of insufficient support by container technologies. Lastly, we present some systems that aim at complementing the Linux security mechanisms by adapting the Linux kernel.

Security vulnerabilities: Dua *et al.* [56] performed a study on how different container implementations

[15] https://github.com/lxc/lxc
[16] https://github.com/kubernetes/kubernetes

handle process, file system, and namespace isolation, concluding that these need to be improved across the different implementations. NCC [57] also described weaknesses in both privileged and unprivileged containers and discussed the importance of a good security configuration in container systems. Souppaya *et al.* [58] performed an analysis of all the risks according to the several components that belong to the container technologies architecture and countermeasures to target those risks.

Another problem with containers lies in the emergence of vulnerabilities in available images due to uncertain image provenance such as described by Tak *et al.* [59], which could be solved with better support for its tracking and by proposing practices that container users should adopt to limit vulnerabilities.

Bui [60] performed an analysis on the security level of Docker, focusing on the internal security and how Docker interacts with the security features of Linux kernel, such as SELinux and AppArmor, based on how these are supported by Docker, claiming they are fairly secure with default configurations, however, some information seems to be outdated and no real experimental analysis seems to have been conducted. Mohallel *et al.* [61] experimented Docker with LXC, concluding that using Docker increases the attack surface exposed by a given host when compared to an OS running the same services. Jian *et al.* [62] proposed checking the status of a process namespace as a defense mechanism against Docker escape attack, that consists of gaining root privilege of the host from the container.

Native Linux security mechanisms: Container isolation is provided by Linux namespaces and cgroups[17], which can be configured to finely sandbox an environment. Namespaces isolate and virtualize resources for a group of processes, which form a container. Cgroups provide a mechanism for partitioning groups of processes with controlled behaviours, limiting the resource usage of each container instance.

Linux security modules (LSM) [63] is a framework that targets Linux and provides a convenient set of kernel hooks placed at strategical points, composed of several modules such as SELinux [64] and AppArmor [65]. These modules can take appropriate actions matching policies defined for particular operations done by processes, such as file creation. Another alternative to enhance containers' security is Linux's secure computing mode (SECCOMP)[18]. Both focus on limiting how a process interacts with the system but where SECCOMP filters and limits the system calls a process can make at a lower level, LSM constrains access to kernel objects.

Problems with applying security mechanisms to containers: Despite all the existing security frameworks, they are generally not suitable for container defense. Belair, *et al.* [66] presented a taxonomy on container defense mechanisms and provided a use case to illustrate why these techniques can be required. However, tools like LSM cannot be easily adapted to containers to improve their security, since no solution solves the security issues by itself. There are several other limitations [67] with containers. Namespacing LSM is also challenging [68], meaning LSM modules need to handle namespacing by themselves to be useful for container security and there's no way to safely alternate between LSMs.

Enhancing existing security mechanisms: As we described in the paragraph above, some situations may require adapting existing security mechanisms. Gao *et al.* [69] studied several leaks that allow a container to access information from the host that should not be accessible and devised a two-stage

[17] `https://www.cs.ucsb.edu/rich/class/cs293b-cloud/papers/lxc-namespace.pdf`
[18] `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`

defense mechanism based on the Linux kernel security mechanisms. The first step consists of denying read access to leakage channels through security policies in AppArmor or mounting the pseudo-file "unreadable", which does not require changes to the kernel code. The second stage consists of the implementation of new namespaces to enhance the container's isolation model, which might require changing the kernel code to allow for a more concrete division of system resources.

More specific solutions have also emerged, such as X-Containers [70], which presented a paradigm for isolating containers in a cloud environment by leveraging *Xen paravirtualization* to turn Linux kernel into a LibOS without requiring hardware-assisted virtualization to maintain efficiency. Another system that aimed to enhance container security but only at the network level is Bastion [71], a sandbox for securing communications between containers using Linux kernel features to create a new network stack for container networks.

Summary

This chapter covered some necessary background and the related work. We have seen that forensic techniques for persistent data storage are very effective at retrieving traces from program execution. There have been several attempts at building deniable file systems in the past. However, they do not offer complete plausible deniability in the case of a forensic investigation, mainly due to visible changes in the disk that can be associated with hidden data. Even more recent approaches fail at this by relying on additional hardware components or by requiring to boot another OS for isolation, which constitutes points of failure for the deniability of the system. In the next chapter, we present a forensically-deniable storage system that aims to hide the installation and execution of PETs.

Chapter 3

Design

Calypso aims at establishing a secure storage partition for enabling the deniable execution of PET applications such as Tor. It is meant to be used by regular users without exceptional computer knowledge, which entails making it as simple and user-friendly as possible and not requiring extra hardware components or complex and unusual system specifications. The fact that Calypso must not disrupt the native operating system environment, combined with the complex Linux kernel realm, creates several design challenges that we address throughout this chapter.

In this chapter, we present the design of Calypso. First, Section 3.1 provides an overview of its architecture. Then, Section 3.2 describes the threat model faced by Calypso. Section 3.3 explains the initial setup and final clean up processes that need to be performed to prevent leaving traces of Calypso's module being inserted in the kernel. Section 3.4 details how Calypso chooses the blocks to be used, and Section 3.5 explains how changes to those blocks by the native system are handled. Finally, Section 3.6 describes the evolution of our solution to securely persist state across executions and Section 3.7 describes how Calypso's data is securely encoded in the blocks.

3.1 Overview

We propose Calypso, a system that provides a block device abstraction of a secondary file system composed of the free blocks of the native file system, which we refer to as *shadow partition*. This shadow partition encodes the persistent data employing data hiding techniques so that no visible changes are made to the persistent state of the machine. We consider *native free blocks* to be the blocks that are currently not used (are not linked to any file) by the native file system but may be allocated at any time.

This system is described in Figure 3.1, along with its main components coloured in pink. Its functionality is going to be implemented mainly by Calypso's block device driver, inserted in the system as a kernel module. We assume we have a disk with a main partition, formatted with a regular in-disk file system such as Ext4, from which we can obtain the numbers of all free blocks as a bitmap. We consider the *native file system* to contain the applications and data installed in the user's machine before Calypso. The Calypso driver exposes a block device (e.g., /dev/calypso0) that exports a shadow partition whose

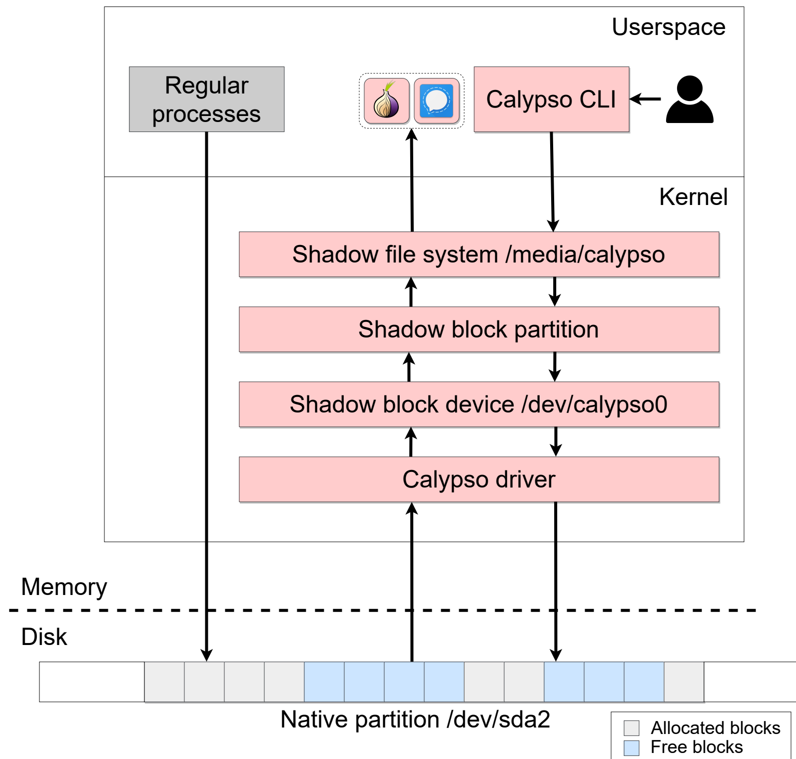

Figure 3.1: Calypso architecture.

size is determined by the user when installing Calypso. This partition can be formatted and mounted on a given folder, e.g., /media/calypso.

The interaction with the user is done through a command-line application (CLI) that allows executing the bootstrap system that inserts Calypso's module in the system, with the option to recover previous data or reset it, or the option to clean up Calypso and remove it from the kernel, always taking into account that no signs of Calypso should be left on the system. It is also through the Calypso CLI that the user should be able to install repackaged PETS such as the Tor browser or Signal, or directly copy data into the shadow partition through the shadow file system.

The heart of the system is the Calypso driver. It is composed of two main parts, shown in Figure 3.2: i) a block mapper and a block encoder, which handle the requests to the shadow partition, hiding Calypso's data on the free blocks of the native file system; and ii) a native requests hook and a read hook, which handle the intercepted requests to the native partition, monitoring and handling possible changes to the native partition that may compromise Calypso's hidden data. We refer to *virtual blocks* as the block numbers from the Calypso block device since those blocks are just an abstraction for the actual physical blocks, and *native blocks* as the blocks belonging to the native block device under Calypso.

Private sessions: Our work contemplates a scenario where a user launches Calypso, executes a PET in the Calypso shadow partition, closes Calypso, and returns to the regular usage of the machine. We refer to *private session* as the set of activities performed in the Calypso shadow partition while Calypso is loaded. The ultimate goal of a private session is to ensure the absence of trances on persistent storage that can reveal the execution of the said PET application if the computer is subjected to a forensic disk analysis after the termination of a private session. We believe there is a wide range of scenarios where

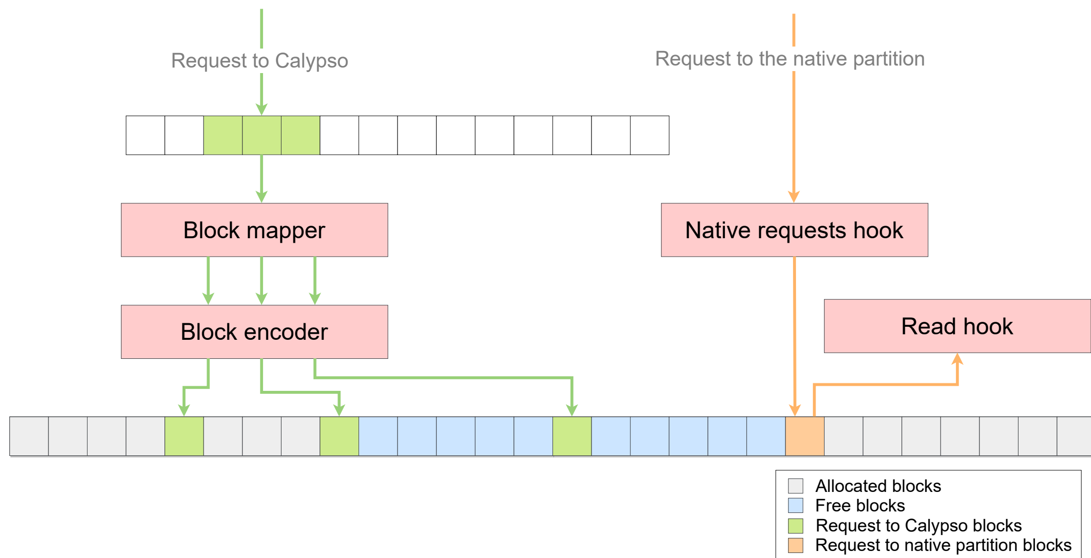

Figure 3.2: Calypso driver internals.

Calypso's private sessions would benefit a user, for instance in the following cases:

- *Protection against censorship agents:* Users enduring censored regimes may be afraid of using PET applications due to possible legal reprisals. In particular, journalists reporting issues such as human rights violations may face danger if their activities are uncovered. Calypso was designed to preserve security against experienced forensic inspectors, as long as under the conditions specified in Section 3.2.

- *Privacy from abusive cohabitants:* Other people have physical access to a user's computer, such as an abusive partner, who attempts to monitor the user's computer usage. Even if the abusive partner has advanced computer knowledge, they should not be able to monitor the user's activities, as long as the user uses Calypso only when the abusive partner is not near.

- *Privacy in public computers:* Several people in the world cannot afford to have a private computer, so they need to use public computers from places such as libraries or cybercafes. This means other users or even the computer administrators may try to monitor the computer usage, which should not be possible if the user resorts to Calypso.

3.2 Threat Model

The goal of the adversary is to detect that Calypso is installed in the system, or that a PET was executed with Calypso's support. The adversary might also attempt to retrieve Calypso's meta-data and data to recover more information about the user's hidden activities.

We consider the adversary only has physical access to the machine after Calypso is closed, but has a forensic set of tools to proceed with a full analysis of the persistent state of that machine. Additionally, the adversary knows the details of the Calypso implementation. In these circumstances, the adversary

should not learn any information about Calypso's existence in the machine or about anything executed or stored in Calypso's shadow partition, from inspecting the machine.

We assume the adversary does not have the means of analysing anything that is not persistently stored. We presume such investigations can occur when crossing borders or when a user is under suspicion by a repressive regime that applies censorship, such as the case of a journalist reporting humanitarian causes to a foreign country.

Security properties: More specifically, Calypso aims to provide **plausible deniability**. This property consists of the ability to deny knowledge or responsibility so that users can plausibly deny having our system installed on their devices. In the context of our project, this is achieved when a forensic analyst is not capable of identifying persistent changes in the disk caused by executing Calypso in the system. Concretely, we aim to offer plausible deniability by providing the following two sub-properties:

- **Isolation:** Guarantees no information leaks from the Calypso processes into the native file system.
- **Non-observability:** An adversary should not be able to observe the presence of hidden data on the native free blocks.

3.3 Traceless Bootstrap

To make Calypso deniable, we need to hide: i) the Calypso's kernel module and evidence that it was inserted, i.e, commands issued, lists of modules inserted in the kernel and journaling logs; ii) the bootstrap script that aims at hiding Calypso's kernel module traces; iii) the data blocks to support applications executed with Calypso's support; and iv) the meta-data blocks to persist Calypso.

Bootstrap is the initial process required to execute Calypso without leaving detectable traces in the machine. Concealing this process is a crucial aspect. To illustrate, if an adversary finds the bootstrap script or manages to correctly retrieve some of the Calypso meta-data, our system loses its deniability.

Discussion of possible bootstrap hiding techniques: Several rootkit techniques aim at storing and executing the bootstrap program without leaving traces on the native file system. There is also the possibility of storing the bootstrap program with steganographic techniques, such as encoding it in a PDF file [72]. Alternatively, we could write directly to a native free block and recommend the user to remember the number of the block, or even split the bootstrap file among several blocks using Shamir Secret Sharing [73].

3.4 Block Allocation

This section details the challenge of selecting and keeping track of blocks to accommodate Calypso's data. For this, Calypso needs to keep track of the native blocks that may potentially be used to store information in the shadow partition. The free blocks do not contain data in use by the native file system, thus overwriting them would not cause data loss. Another reason is that changes to these blocks

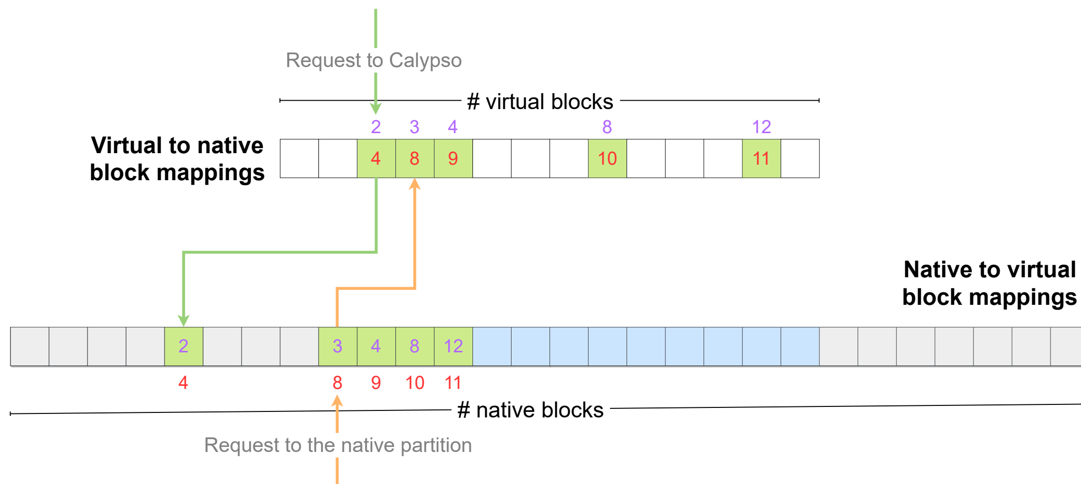

Figure 3.3: Calypso block mappings.

might have several explanations, unlike using unallocated space from the disk, so there is no reason to suspect the existence of Calypso. As the storage demand increases, Calypso should use those blocks to compose the shadow partition, where each block in use needs to be consistently mapped to a native block, or the data in the shadow partition may become corrupted.

When Calypso is first executed, the user inputs the maximum number of blocks it should occupy. For Calypso to allow users to execute programs and store data in the shadow partition, it needs to determine which blocks to use, in such a way that it does not interfere with the native system. For this, we have two data structures that map the Calypso's block number to the respective native block number and vice-versa, in which we store the mappings of all blocks allocated by Calypso. Whenever Calypso receives a request to a block that does not have a mapping, it looks for the next free block in the native blocks bitmap, assigns a new mapping, and redirects the request to that native free block. When Calypso receives a request to a block that is already mapped, that request is instantly redirected to the mapped native block. More specifically, these data structures have the following composition and updating semantics:

Block mappings: To know which virtual block corresponds to which native block, we made two arrays, as represented in Figure 3.3: i) one for mapping virtual to physical blocks, of length equal to the number of virtual blocks, in which the index is the virtual block number and the content is the native block number; and ii) another array for reverse mapping of the native to virtual blocks, of length equal to the number of native blocks, in which the index is the native block number and the content is the corresponding virtual block number so that we can issue requests to the right blocks.

This redundancy was done for simplicity and direct access times since we need to map blocks both ways due to: i) receiving requests to Calypso, which contain the virtual block number that needs to be translated into a native block number; and ii) intercepting requests to the native block partition, such requests containing the number of a native block that needs to be translated into a virtual block number. This way, we can test whether that block is being mapped by Calypso and if mapped, it gives us the virtual block number directly.

On the other hand, this solution has the downside of growing linearly in space as the number of native and virtual blocks grow, so there are better alternatives that consume less memory, such as having two hash maps instead of two arrays, with the key as the array index and the value as the content, allowing for direct access likewise, but the space occupied would only be proportional to the number of blocks being used by Calypso, making it a much more efficient solution in terms of memory.

Used blocks bitmap: The used blocks bitmap is a contiguous array containing 1 bit for each native block. If the bit equals 0, then it is not set and the respective block is free, otherwise, it is set and the block is allocated. This bitmap allows Calypso to know which blocks can be used next by looking for the next unset blocks.

Tracking Calypso allocated blocks: Whenever Calypso receives a request, it first checks if the virtual block is mapped to a native block using the virtual to native block mappings data structure. If it is mapped, the block number is translated and the request proceeds to the native device. On the other hand, when the block is not mapped, we look for the next unset block in the used blocks bitmap, set that bit and assign new mappings for the respective block. Setting the bit is essential to prevent assigning the same native block to multiple virtual blocks. Performing a sequential block allocation is a simple solution that optimizes the request processing, however, we should study its implications in the observability of Calypso's hidden data and block usage patterns in future work.

3.5 Monitoring Native Changes to Blocks

This section targets the additional challenges caused by using the free blocks of the native file system. Firstly, Calypso needs to update the state of the used block bitmap as the native system allocates new blocks, which need to become ineligible for Calypso to allocate. Furthermore, free blocks are volatile since they may be overwritten at any time the native file system needs more blocks. Overwriting data in a block mapped by Calypso can cause corruption that can lead to, i.e., irreparable damage to vital meta-data structures of the file system installed on the shadow partition. To prevent this, Calypso will monitor its mapped blocks and take the appropriate action when a potential overwrite is detected.

When monitoring changes to blocks, we only want to account for requests that were not issued by Calypso since the requests issued by Calypso should not corrupt its allocated blocks. Whenever there is an intercepted write, if the corresponding bit is not set in the bitmap, we set it, preventing Calypso from allocating this block. We only do this for writes, because, for instance, if we read all the blocks in the native partition with a tool such as 'dd', we would be setting all the bits in the bitmap, which does not mean they are all in use.

Preventing block overwrites while Calypso is loaded: When Calypso intercepts a request to a block mapped by Calypso, it means a Calypso block is about to be overwritten. To prevent losing data, Calypso should block the current write request until the content of the block about to be overwritten is copied into the next free block. Copying the block requires applying a read hook, shown in 3.2, that allows Calypso to issue a write operation when the result from reading the block about to be overwritten is available.

Dealing with block overwrites while Calypso is not loaded: The persisted used blocks bitmap gives Calypso the necessary information to find out which Calypso mapped blocks suffered changes by the native file system while Calypso was inactive and unable to prevent them. This is done by comparing the bitmap retrieved from the file system and the persisted bitmap. If a block's bit is different in both bitmaps, and that block was being mapped by Calypso, then its contents may have changed, and we should resort to existing data recovery techniques based on redundancy, such as error-correcting codes, described in Section 2.3.2.

3.6 Persisting Calypso Meta-Data

All Calypso data structures that support the shadow partition live in memory. This means that once Calypso gets unloaded, all its data structures, and consequently all the contents of the shadow partition are going to be lost. We want users to be able to continue the usage of the shadow partition from the state of previous private sessions. To do this, Calypso stores its meta-data in native free blocks before being unloaded, so that when it is reloaded, it can populate the memory data structures as they were in the previous session, which presents significant challenges.

3.6.1 Persisted Meta-Data

Calypso needs to persist the meta-data required to resume the state of previous private sessions, which consists of: i) the data structures that allow Calypso to know which blocks to use and when to allocate a new block, specifically the block mappings; and ii) the data structures that allow Calypso to know which blocks were corrupted while it was not loaded in the system, namely the used blocks bitmap.

Block mappings: Persisting the block mappings data structures allows Calypso to know which blocks were already allocated, to which native free block. For instance, if we have a file in the shadow partition that we want to recover when reloading Calypso, we need to know the native blocks it occupies, and the virtual blocks that have the shadow partition's file system meta-data, so that the file system can be remounted and it maps a file, that when read is actually accessing the underlying native blocks.

Used blocks bitmap: Persisting the used blocks bitmap allows Calypso to find out which blocks might have been corrupted while Calypso was inactive. Then, we set the mapped bits on the bitmap retrieved from the file system to contain information on the blocks mapped by Calypso so the system will choose the next free blocks correctly and does not map repeated native blocks. The latter could also be done with the block mappings, but having the previous bitmap makes it easier and faster to change the values in the retrieved file system bitmap due to the higher complexity of individually iterating a bitmap, in comparison to applying bit-wise operations such as *xor*.

Meta-data blocks structure: Since we need the virtual to native block mappings and the used blocks bitmap, we cannot expect these to always fit in a single block. So, this meta-data needs to be split into several parts. The visual representation of this process and the meta-data block structure can be observed in Figure 3.4. All meta-data content is represented in orange. To know the number of the next

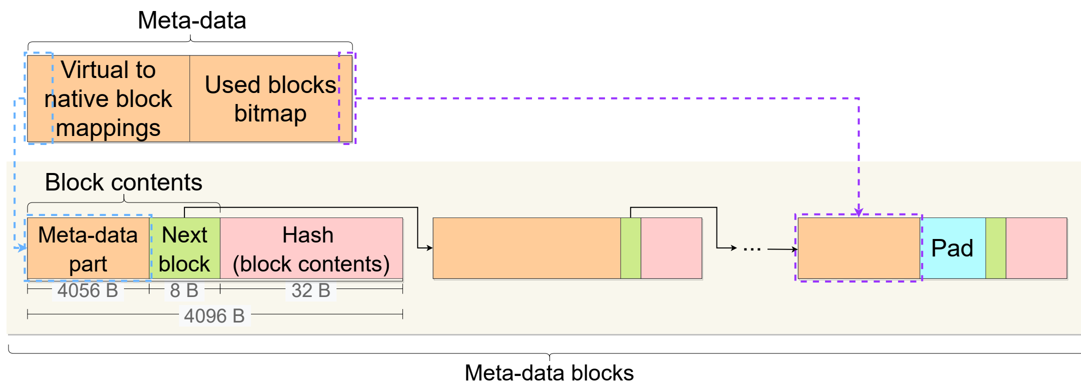

Figure 3.4: Calypso meta-data blocks structure.

meta-data block, we need to have an additional field with the size of a long, represented in green.

Additionally, we need to take into account that, while Calypso is not loaded, the meta-data blocks might be corrupted by the native file system. Thus, we need to have a mechanism to check the integrity of the meta-data block contents, along with a method to check whether that block is actually a meta-data block written by Calypso or a random block. Both situations can be solved by implementing a 32-byte hash of the block contents (the meta-data part and the next block fields), represented in pink, which needs to be checked every time Calypso attempts to retrieve a meta-data block from the native partition. This is accomplished by hashing the first 4064 bytes (the first 4056 from the meta-data part, and the remaining 8 bytes from the next block field), and comparing the result with the retrieved hash, which consists of the last 32 bytes of each block.

For the last block, the remaining meta-data part might not occupy the whole 4056 bytes reserved for it. So we need to apply padding to the remaining bytes, as represented in blue.

The size of the meta-data to persist depends on the number of virtual blocks assigned to Calypso for the virtual to native block mappings, and on the number of native blocks because of the used blocks bitmap. We only need to persist the smaller mapping data structure because we can reconstruct both block mappings from a single one since we can obtain the number of physical blocks from the bitmap. The space occupied by the block mappings can be improved by storing only the active mappings.

3.6.2 Managing Meta-Data Blocks in the Native Partition

The meta-data persisted by Calypso is encoded in the blocks mapped by Calypso itself. However, these mappings are in memory, so they are lost once Calypso is unloaded. So, the first non-trivial challenge is to find the location of the first meta-data block. We will begin with a single assumption: the meta-data fits on a single block. We start by obtaining deterministic pseudo-random numbers using a seeded pseudo-random number generator. When encoding meta-data, the first number to correspond to a free block decides where the meta-data is going to be. When retrieving the meta-data, we obtain the same sequence of pseudo-random numbers using the same seed and read the associated block sequentially. The first block with a matching hash corresponds to the first block of meta-data.

Now, we assume we are going to have multiple meta-data blocks since the meta-data is not likely to

fit into a single block. To know the number of the following blocks containing meta-data, we simply need to have a “next block” field in each block and the last block is going to contain an invalid number like -1.

Finally, we need to encode meta-data in the blocks. Otherwise, a forensics inspector could simply inspect the contents of the blocks in the disk and retrieve the Calypso meta-data in plaintext. Calypso encrypts data to be written in each meta-data block and uses only blocks that already had a similar entropy to make these changes deniable. The remaining details and cryptography primitives are described in further detail in Section 4.5.

Preserving Meta-Data Mappings in Memory: Calypso needs to know in which blocks to store meta-data. Calypso could look for the next free block each time it was about to save meta-data. However, updating the used blocks bitmap accordingly to unset blocks previously used to store meta-data is complex and error-prone, so when Calypso retrieves the meta-data once it is loaded, it will store the respective mappings from meta-data blocks to free native blocks, similar to previously described mappings data structures, and reuse them the next time Calypso is unloaded.

3.7 Securely Encoding Data in Blocks

At this point, Calypso is working and supports executing PETs on the shadow partition. However, anyone with physical access to the machine can read the blocks mapped by Calypso and understand what's being executed in there. This does not ensure plausible deniability. The data that Calypso writes on its mapped blocks needs to be encoded so that its contents are not directly observable. Even if Calypso encrypts all the data to be written, these changes may still be observable, especially if the adversary has access to multiple snapshots of the disk since that block pattern may be associated with data hiding.

To contradict this, Calypso encodes the blocks according to their characteristics, i.e, an encrypted block has high entropy and can be replaced for encrypted data, whereas a plaintext block has lower entropy, so only a few bits might be encoded to maintain the entropy level. For simplicity, we decided to establish an entropy threshold, so that all blocks with entropy values equal to or above that threshold can be used by Calypso to hide its encrypted data. Other algorithms can be used to encode data in blocks with lower entropy values, but we left that for future work.

3.7.1 Entropy and Entropy Differential

The *entropy* of a block is the variability of the bytes in it. For instance, a block with an equivalent count of every possible byte has the highest entropy possible, while a block only with zeroes is going to have the lowest possible entropy. This measurement is calculated using the Shannon entropy [74, 75] for each block, described in Equation 3.1.

$$H(X) = -\sum_{i=1}^{n} P(x_i) log_2 P(x_i) \tag{3.1}$$

In our case, we consider each block has 4096 bytes, and each byte has 8 bits, so each byte has $2^8 = 256$ different possibilities. First, we need to count the frequency with which each possibility occurs

in the 4096 bytes, obtaining the probability $P(x_i)$. The lowest value for the entropy of a block is 0, which occurs for instance in a block that had its content replaced by zeroes during the utilization of tools for secure software deletion. This value is due to $P(x_i) = 1$ for a single case and $P(x_i) = 0$ in all others, so $P(x_i)\log_2 P(x_i) = 0$ for all values of i in the summation. Contrarily, the highest value possible is 8, because it happens when the frequency of all the possible variations of the value of a byte is the same. Encrypted or random sequences of bytes are the closest to this value, which happens for $P(x_i) = \frac{16}{4096}$. Since we have 256 possible values for each byte, $n = 256$ and $-\sum_{i=1}^{256} \frac{16}{4096}\log_2 \frac{16}{4096} = 8$. Therefore, we translate the entropy of a block through Equation 3.2.

$$H(X) = -\sum_{i=1}^{256} P(x_i)\log_2 P(x_i), 0 \leq H(X) \leq 8 \quad (3.2)$$

The *entropy differential* is the measure we established to quantify how much a block has been changed, thus how observable it is that there is a hidden mechanism in the system such as Calypso. It consists of the absolute value of the difference between two entropy values, taken in two distinct moments in time, as shown in Equation 3.3. These values are normalized to be between 0 and 1.

$$\triangle H(X) = \frac{|H(x_1) - H(x_0)|}{max(H(X))}, 0 \leq \triangle H(X) \leq 1 \quad (3.3)$$

Using entropy as a decisive factor to change the contents of free blocks allows Calypso to encode data in the free blocks while maintaining their characteristics. For instance, if a block contains remains of a plaintext file, we should only change it in a way that it presents the same entropy characteristics of a plaintext file in the end. Likewise, if a block contains remains of an encrypted file, the changes made by Calypso should still reflect an encrypted file in the end.

This way, Calypso remains deniable because the user can claim those are regular changes that occur due to temporary files or are a consequence of the user's habit of doing some activities like watching streaming movies. Contrarily, if Calypso changed blocks indiscriminately, for instance, by encrypting a big amount of consecutive blocks, a forensics inspector in the power of a previous snapshot could observe that these changes are not part of the user's regular utilization.

3.7.2 Block Entropy and Entropy Threshold

The plausible deniability of Calypso depends on making only deniable changes to the native blocks, which is achieved by choosing only blocks with a value of entropy equal or above the established entropy threshold to store the encrypted data of Calypso. However, this entails classifying the entropy of all blocks that may be potentially used by Calypso. This is a challenge since, for each block, it requires: i) issuing a read request to obtain the block contents; and ii) classifying the entropy of the read contents, based on Equation 3.1. To mitigate the performance penalties of repeating this process whenever Calypso allocates a new block, we classify the native free blocks' entropy when Calypso is loaded, right after retrieving the usable blocks bitmap.

To avoid having an additional data structure, we adapted the existing used blocks bitmap by unsetting

only the blocks with the entropy value on the established entropy threshold, and the remaining blocks are going to be seen as unusable.

The optimal entropy threshold: The optimal entropy threshold value is going to depend on: i) the characteristics of the native free blocks; ii) the amount of storage capacity the user needs; and iii) the level of security the user needs and the danger they are facing.

The characteristics of the native free blocks depend on a variety of factors, i.e, the amount of free space the user has, how old the disk is, if any cleaning tool is used, the workload of the services and programs installed and frequently executed. For instance, if the user requires a big amount of space for storing pictures and will not have the computer inspected by a censoring authority, then the entropy threshold can be the lowest to allow maximal native free blocks utilization. On the other hand, if the user is a journalist reporting government illicit activities under a repressive regime, and only needs to use a whistleblower submission system such as SecureDrop[1] with the Tor browser, then the highest entropy threshold is advisable. The minimum entropy threshold will ensure every free block can be used by Calypso, whereas a maximum entropy threshold will ensure only blocks with entropy equivalent to the entropy of encrypted blocks.

Using steganography to increase storage capacity: Relying on multiple data hiding techniques such as steganography, other than encryption, will help increase the amount of native free blocks that are usable by Calypso when using higher entropy threshold levels because we can deniably change blocks with entropy level below the required by the threshold.

We complement this section with an experimental evaluation performed in Section 5.4.

3.7.3 Multi-Snapshot Attacks and Plausible Deniable Footprints

Multi-snapshot attacks happen when an adversary has access to a machine in multiple moments in time, and each time is able to take a snapshot of the persistent state of the machine. This implies the adversary can analyse all the differences that happened in the machine from one point in time to another, such as the changed disk blocks.

Resistance to multi-snapshot attacks entails that the adversary cannot observe changes between multiple snapshots, or that the observable changes cannot be linked to a specific questionable activity such as executing Calypso. However, establishing a single level of changes that would be acceptable between two disk snapshots is a disputable matter. A file system environment is in constant change for several reasons, i.e, installing and removing software, compilation of programs, temporary files written to disk such as video streams.

We attempt to make all changes made by Calypso non-observable by limiting the usable blocks to the native free blocks with an entropy value above a customizable threshold. For instance, we can limit the system to use only previously encrypted blocks to replace with Calypso's encrypted data, thus maintaining the characteristics of the free space. So, we cannot state no changes were made to those blocks, but we can claim these changes were originated from a regular utilization, such as temporary

[1] `https://securedrop.org/`

keys that regularly get generated and erased. From this idea, we can expand to higher disk workloads, to sustain more deniable storage capacity. We named this concept *plausible deniable footprints*.

We can request the user to watch a movie through a streaming service such as Netflix, which would be the plausible deniable footprint, before executing other programs in Calypso's shadow partition. This will change several blocks temporarily allocated, later becoming free for Calypso to use. An additional Calypso component called *access monitor* should be loaded during the execution of the plausible deniable footprint to record the used blocks that can now be taken advantage of by Calypso. The access monitor is responsible for recording the number of all blocks written to by the shadow partition while it is loaded, and sharing that information with Calypso via memory mechanisms.

Summary

This chapter specified the architecture and design of Calypso, a novel support system for the traceless execution of PETs that parasites the free blocks of the native file system and the notion of block entropy to encode hidden data deniably. The primary concerns lie in: i) block allocation and associated data structures; ii) monitoring of the used blocks; iii) persisting the state of the shadow partition; and iv) securely encoding data in blocks. All these matters allow Calypso and everything installed and executed in the shadow partition to cohabit the native partition without disrupting or being disrupted by it while remaining deniable to a forensics inspector with physical access to multiple disk snapshots. The next chapter details the implementation of Calypso's prototype and the technical challenges faced.

Chapter 4

Implementation

This chapter discusses the implementation details of the Calypso prototype, which is targeted for Linux platforms running the Linux kernel version 5.4. Section 4.1 presents an overview of the Linux kernel block layer, introducing relevant concepts to the reader. Section 4.2 establishes the requirements to execute Calypso on a machine. Section 4.3 describes how Calypso parses the Ext4 data structures to retrieve the bitmap identifying the native free blocks. The mechanisms used for Calypso to handle requests using the native free blocks and to monitor accesses to these blocks are described in Section 4.4. Lastly, Section 4.5 details the cryptographic primitives used to ensure Calypso's data is securely hidden within the native system, and only the rightful user can access it.

4.1 Introducing the Linux Kernel Block Layer

This section provides the reader with the necessary background on the Linux kernel mechanisms related to block I/O request processing. This introduction is necessary because the implementation of Calypso relies heavily on these mechanisms. First, we introduce the basic concepts necessary to proceed to the analysis of Figure 4.1, which we explain throughout this section. Then, we explain the structures that support processing I/O requests, delving deeper into the block layer[1,2,3,4,5,6].

4.1.1 The Linux Kernel Block I/O Stack

The block I/O stack is the set of the independent kernel components that support the processing of data transfer requests to block devices, such as read and write operations. Each of the main components is illustrated in Figure 4.1, which we will explain in the next paragraphs.

Storage units: The smallest unit of data transfer from disk is the *sector*. It generally consists of a group

[1] http://byteliu.com/2019/05/10/Linux-The-block-I-O-layer/

[2] http://byteliu.com/2019/05/21/What-is-the-major-difference-between-the-buffer-cache-and-the-page-cache-Why-were-they-separate-entities-in-older-kernels-Why-were-they-merged-later-on/

[3] https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram

[4] https://xuechendi.github.io/2013/11/14/device-mapper-deep-dive

[5] https://students.mimuw.edu.pl/ZSO/Wyklady/13_IOschedulers/io_schedulers.pdf

[6] https://myaut.github.io/dtrace-stap-book/kernel/bio.html

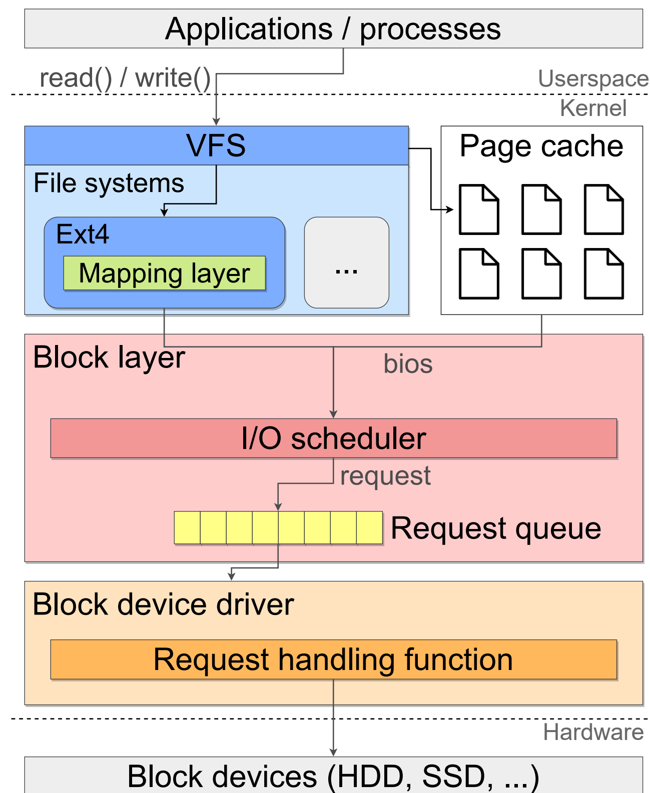

Figure 4.1: Linux kernel block I/O stack.

of 512 adjacent bytes. A *block* is the smallest addressable unit of a file system. It consists of a group of adjacent sectors, generally of 4096 bytes, equivalent to 8 sectors. A *page* is a memory unit belonging to the virtual memory mechanisms and serves as a buffer for intermediate file system operations. Normally, each page has 4096 bytes size and maps to file system blocks[7].

Virtual file system (VFS): The VFS is an abstract layer that exposes a uniform interface for applications to interact with the file system, without depending on the implementation of each file system. This mechanism allows Linux to support multiple file systems. Each file system implements the operations to be performed on files, which are going to be called by the VFS. This component is also responsible for checking if the requested data is already mapped into memory in the page cache, or if needs to be read from the disk. In case it needs to be read from the disk, the VFS activates the mapping layer.

Mapping layer: This layer is in charge of computing the location of the data, based on file system-specific information, determining the block numbers in the disk containing the target file.

Page cache: It caches file data from a disk to make subsequent I/O faster. The page cache is addressed when reading and writing file data. In the case of a read, if it contains the desired block contents, that data is copied to the user, otherwise, a page fault is triggered and the contents are fetched from the disk. During writes, the contents are initially written to the page cache, later written to disk.

Block layer: This layer is responsible for managing input/output operations performed on block devices. It receives the request to retrieve the desired data, not necessarily adjacent on disk. Because of this, this layer might start several I/O operations, each represented by a bio structure.

[7]https://medium.com/databasss/on-disk-io-part-1-flavours-of-io-8e1ace1de017

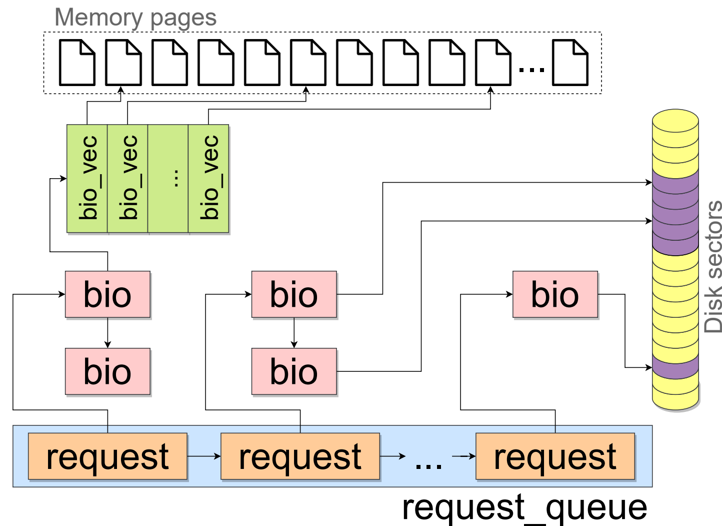

Figure 4.2: I/O request structures.

I/O Scheduler: An I/O scheduler manages a block device's request queue to reduce seek times by merging and sorting requests. For instance, it can collect multiple bios to obtain a smaller number of requests, but these must be physically adjacent to the ones already in the request.

Block device drivers: These contain the code that handles the I/O requests to each block device. They receive file system requests and issue the corresponding I/O operations to the block device.

Block devices: These devices are characterized by random access to data, making them ideal to be used as storage devices, such as our native partition. They are complex to ensure performance.

4.1.2 I/O Requests and I/O Request Structures

An I/O request works as follows. A process accesses the disk via a read or write system call, which invokes the respective VFS operations. The VFS checks the page cache and if the data is not present, it calls the mapping layer to compute the location of the data in the disk. Then, the block layer creates a list of *bio* structures, each representing a block I/O operation to be submitted to the disk. Subsequently, the block layer translates bios to requests. Finally, the block layer dispatches the requests to the device driver, where the actual data transfer happens. The request queue allows the block layer to pass requests to the device driver. It contains a linked list of requests and pointers to functions that handle requests just before the submission to the respective device. Each device is generally associated with a private request queue that can only be accessed by the respective driver. It contains the available requests for that block device, managed by the I/O scheduler, responsible for reordering requests, i.e, making the operations semi-sequential to speed up mechanical disks such as HDDs.

To understand how I/O requests are organized and processed by the multiple kernel subsystems, we should understand the structures that describe it[8,9,10], as illustrated in Figure 4.2.

[8]https://www.halolinux.us/kernel-architecture/request-structure.html
[9]https://lwn.net/Articles/738449/
[10]http://sylab-srv.cs.fiu.edu/lib/exe/fetch.php?media=paperclub:lkd3ch14.pdf

request_queue: Request queues maintain the pending requests to a block device. The block device driver grabs requests from the request queue and submits them to the associated block device.

request: The request structure describes each request to a block device and it is the unit that composes a request queue. The request structure's fields indicate the exact position of the data to be transferred, such as the start sector, the number of sectors to transfer for the current request, and the number of requests still pending. Each request can be composed of multiple bios to optimize request processing.

bio: The bio structure represents the block I/O request that pertains to a set of consecutive sectors in the disk. It is associated with a vector with each position, a *bio_vec* structure mapping to a contiguous chunk of a memory page. The vector of bio_vec structures allows performing block I/O operations from multiple locations in memory. These pages are used to receive data from and send data to a block device. It contains other fields such as the target block device, the number of the sector in the target block device, flags that identify the operation to be performed, and the total size of the request in bytes.

4.2 System Requirements

This section presents the conditions to execute Calypso on a machine without unpredictable behaviour.

File systems and block size: Our solution depends on the file system installed on the native partition to obtain the free blocks. However, it can be extended to support multiple file systems. This is due to the VFS not having abstractions for obtaining the free blocks of any file system. That functionality is file system dependant and we need to iterate through Ext4 specific structures to retrieve that information, as explained in Section 4.3.

We currently assume a block size of 4096 bytes, which is the default for Ext4, but the block size can be retrieved via VFS. Calypso can work with any file system block size, as long as it remains the same as the one in the first execution of Calypso, where the association with the native file system and partition, and definition of data structures is performed.

Any file system can be installed on top of Calypso's shadow file system, as long as its block size is the same as the native file system's. Nevertheless, Calypso can be extended to support dealing with multiple block sizes, and it should be a feature if it is extended to support more native file systems.

Calypso's storage capacity: The storage capacity is a parameter passed by the user during the initial execution, limited by the number of free blocks in the native file system. If the storage capacity in Calypso and the blocks to store meta-data surpass the amount of native free blocks, Calypso may stop functioning correctly. However, only the blocks already allocated by Calypso are occupying native free blocks. If a user specifies a much higher storage capacity than the one required, it is not going to need to occupy those remaining free blocks as long as they are not required.

Kernel version: We are working on the 5.4 Linux kernel version (5.4.0-42-generic) and our Calypso module depends on it to function properly, so our code may not even compile correctly in other versions. A kernel module is a piece of code that can be loaded and unloaded in the kernel on-demand, extending its functionality. However, it depends on the kernel code that is already compiled and installed, which

differs depending on the kernel version.

4.3 Retrieving Usable Blocks for Storage

Assembling a set of blocks that we could use to store the data is vital to ensure Calypso's functionality and persistence. Since we do not want to disrupt the native system by overwriting needed blocks and we do not dispose of a partition just for Calypso, we decided to use the free blocks that are not allocated. This can be technically challenging since the VFS does not have functions to retrieve the free blocks as mentioned in 4.2. So, we need to parse the Ext4 file system data structures[11], iterating all block groups and combine all block bitmaps from each group as a single contiguous bitmap, to simplify the operations to perform on the bitmap, such as looking for the next free block. This contains a bit for every block in the native partition, so we can know if the block is in use or not.

In more detail, using VFS's structures defined in `kernel/include/fs.h` and Ext4's source code[12], we obtain the VFS super block instance of the `struct super_block`. Then, we get the number of Ext4 groups from the super block using the function `ext4_get_groups_count()`, and for each group, we get the respective instance of a group descriptor, `struct ext4_group_desc`, using the function `ext4_get_group_desc()`. Each group descriptor contains the location of the block bitmap within that group in the field `bg_block_bitmap_hi`. We read each group block bitmap from disk by calling the function `ext4_read_block_bitmap()`. Subsequently, we get the number of the first block of this group using the `ext4_group_first_block_no()` function to know the offset that we have to apply to the blocks represented in the group bitmap. Then, we iterate the group bitmap read from disk and set each bit with value 1 in Calypso's bitmap, taking into account the determined group block offset.

4.4 Redirecting and Intercepting Block Requests

In this section we describe the initial challenges faced for Calypso to start using the blocks of the native shadow partition, which entails: i) redirecting the received requests to the native partition and all the challenges that come with adapting the request structures so that it is mapped to the correct blocks; ii) intercepting the native requests to monitor and avoid changes to mapped blocks.

4.4.1 Stacking Calypso on the Native Block Device and Redirecting Requests

The first challenge we faced was for Calypso to start issuing the requests received to disk instead of only to memory. For this, Calypso needs to redirect the requests to the native block device associated with the disk's main partition. This is further detailed in Figure 4.3. Calypso should act as the front end or a device mapper such as RAIDs, receiving the requests and adapting them to be processed by the underlying device. The processing of Calypso's requests is initialized with `blk_queue_make_request()`, defined

[11] https://metebalci.com/blog/a-minimum-complete-tutorial-of-linux-ext4-file-system/
[12] https://elixir.bootlin.com/linux/v5.4/source/fs/ext4

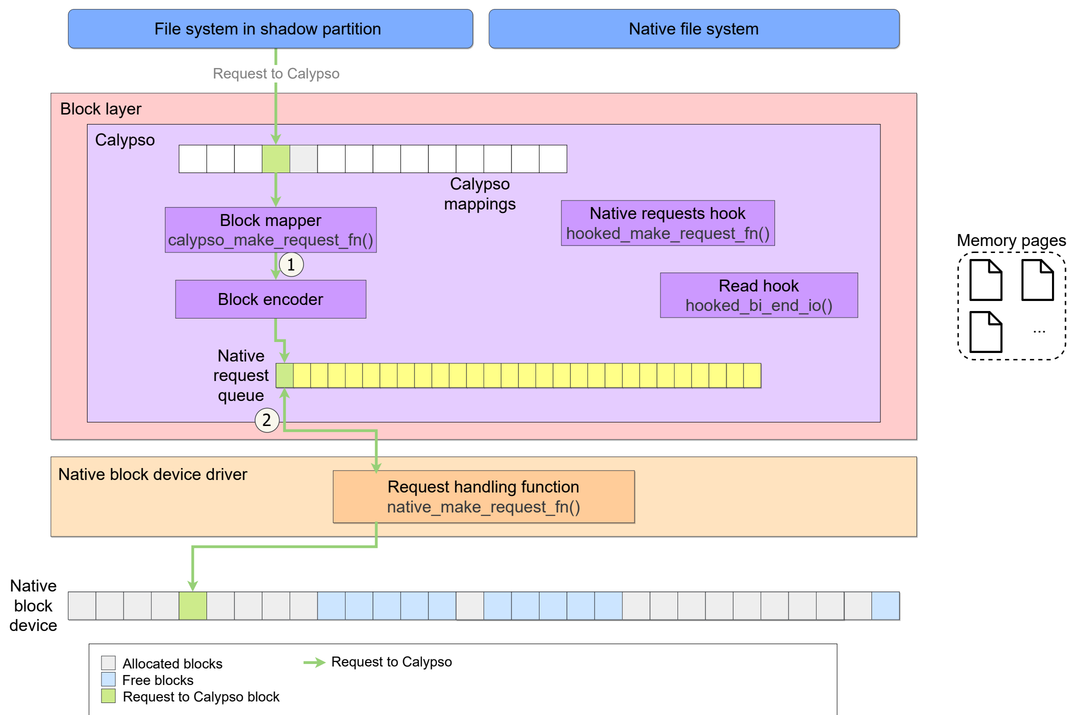

Figure 4.3: Calypso request redirection. The green arrow shows how a request to the shadow partition is handled. Once we get to Calypso, the virtual block associated with the request is mapped to the corresponding native block by the block mapper, as shown in 1), using the Calypso mapping structures. Then, the request data is encoded and the request is placed in the native request queue. Finally, in 2) the request is fetched by the native request handling function and gets redirected to the native block.

in `kernel/block/blk-settings.h`[13] so that requests are passed directly to Calypso's driver instead of going to a request queue. We based this part of our code in a small project called "stackbd"[14,15]. To redirect Calypso's requests to the native device, we need to open the native block device without exclusive mode to obtain its pointer to a `struct block_device`, declared in `kernel/include/linux/fs.h`. Then, for every request, we need to change the following fields in the instance of the `struct bio` (listed in `kernel/include/linux/blk_types.h`): i) the target sector `bio->bi_iter.bi_sector`; ii) and the target device through the macro `bio_set_dev()` defined in `kernel/include/linux/bio.h`; and finally send the request to the native device's request queue using the `generic_make_request()` function, defined in `kernel/block/blk-core.c`.

Splitting multi-block requests into multiple single block requests: Since we do not ensure that Calypso's blocks have contiguous mappings, requests redirected to the native partition cannot target more than one block. Otherwise, it is assumed the blocks in a single bio are contiguous, as explained in Section 4.1, which would result in requests to the wrong native blocks.

To prevent this, we continuously split the bios 8 by 8 sectors, creating new bios, each equivalent to a request to a single block at most, and updating the original bio until it has one block left at most. After splitting, we need to update the resulting bios with the right physical sectors, obtained from multiplying the number of sectors in each block per the mapped native block number and sum the offset in the

[13] `https://www.kernel.org/doc/htmldocs/kernel-api/API-blk-queue-make-request.html`
[14] `https://orenkishon.wordpress.com/2014/10/29/stackbd-stacking-a-block-device-over-another-block-device/`
[15] `https://github.com/OrenKishon/stackbd`

original bio, the latter only performed for the last remaining bio. Then, we submit each bio to the native device's request queue, one at a time, using the `generic_make_request()` function defined in `kernel/block/blk-core.c`.

This way, we ensure no incorrect mappings are issued, and the requests to the native partition issued by Calypso are still going to be optimized once they get to the request queue, by the scheduling algorithm that takes care of merging requests to contiguous sectors of the native partition.

4.4.2 Hooking the Request Handling Function of the Native Partition

To monitor and prevent changes to native blocks mapped by Calypso, our module needs to add functionality to the native request handling function in the respective driver. However, changing the code of the native partition is not an option. So our objective is to hook the function that handles the native requests. Previous versions of the kernel allow us to override the pointer to this function, `make_request_fn`, once we obtained the request queue of the native block device. To obtain this structure, we need the instance of `struct block_device`, which we already obtained when stacking the native block device.

When replacing the native function pointed to by the field `make_request_fn` belonging to the `struct request_queue` in `kernel/include/linux/blkdev.h`, we need to be careful, since any disruption that prevents native requests from being correctly processed may invalidate the entire system. First of all, we need to store the pointer to the original function so that we can call and restore it when Calypso is unloaded. The substitute function should ensure that at some point, the original function gets called.

Another challenge in hooking this function is that the driver for this partition is the same for the entire disk, so we intercept the requests to all the partitions in the disk. To prevent this, we limit the interception to only perform additional functionality if the bio's target sector is within the sector range of the native partition, which we need to obtain from the physical device structure.

The drawback of this solution is that it can no longer be implemented in more recent kernel versions because the structure that contains the block operations of a request queue was made immutable and the kernel assumes a memory violation when we try to override the pointer to the request handling function. There is a project[16] that discusses this problem and suggests an algorithm to surpass this, but it was not implemented yet. Finding a way to hook the request handling function in more recent kernel versions will be left for future work.

Differentiating requests by Calypso from other requests to the native block device: We began by investigating the fields in the `struct bio`, defined in `kernel/include/linux/blk_types.h` responsible for describing a request to a block device. Every field was used and could impact the functionality of the request processing, so we analysed the fields associated with flags and found that `bi_opf` has 32 bits, from which only the first 27 are assigned to a specific meaning. As a result, we defined a new flag `REQ_CALYPSO` that uses the value of the 29th bit to mark requests issued by Calypso. Then, when requests are intercepted, all that needs to be done is to check if the bio's field contains that flag checked.

[16] https://github.com/CodeImp/blk-filter

4.4.3 Copying Calypso Blocks to Prevent Overwrites

```
1 void new_bio_read_page(sector_t sector, struct block_device *
      physical_dev, struct page *pending_page,
2     void (*bio_end_io_func)(struct bio *))
3 {
4      struct bio *bio = bio_alloc(GFP_NOIO, 1);
5      struct page *page = alloc_page(GFP_KERNEL);
6      pending_page = page;
7      bio_set_dev(bio, physical_dev);
8      bio->bi_iter.bi_sector = sector;
9      bio->bi_opf = REQ_OP_READ;
10     bio->bi_opf |= REQ_CALYPSO;
11     bio_add_page(bio, page, 4096, 0);
12     bio->bi_end_io = bio_end_io_func;
13     generic_make_request(bio);
14 }
```

Listing 4.1: Function that issues read request to get block contents to copy

```
1 void new_bio_write_page(struct page *pending_page, struct
      block_device *physical_dev, sector_t sector)
2 {
3      struct bio *bio = bio_alloc(GFP_NOIO, 1);
4      bio_set_dev(bio, physical_dev);
5      bio->bi_iter.bi_sector = sector;
6      bio->bi_opf = REQ_OP_WRITE;
7      bio->bi_opf |= REQ_CALYPSO;
8      bio_add_page(bio, pending_page, 4096, 0);
9      generic_make_request(bio);
10 }
```

Listing 4.2: Function that issues write request to copy block

Calypso is using the free blocks of the native file system, which can be overwritten at any time if the native file system requires allocating more space. Since Calypso intercepts all requests made to the native block device, it detects when the native system is about to overwrite a Calypso block. Overwrites happen when a write request, issued by the native file system, targets a native block mapped by Calypso. In that case, we need to freeze that request while data is being copied as follows:

1. Issue a read request to retrieve the data in the original block, shown in 4.1. To issue it directly to the block layer, we allocate a new bio with a new page and store them in Calypso's device structure, avoiding extra indirection and overhead. The stored page is later used to resume the write, since we may not be able to access the page field in the bio structure once the request has been processed. Then, we hook the `bi_end_io()` function, generically defined as `bio_endio()` in `kernel/block/bio.c`, by overriding its pointer in `struct bio`, allowing us to intercept the bio's termination, so we can issue the following write that copies the data to the target block.

2. Once the `bi_end_io()` function is called, it is going to advance to the write that copies the data to the next free block, shown in 4.2. To issue this write, we proceed the same way as for the read, but we copy the data in the stored page from the read.

3. After issuing the write request that copies the block, we can call the original request handling function through its pointer stored in memory, since we have the data to copy in a different memory page. Then, we can proceed to update the mappings.

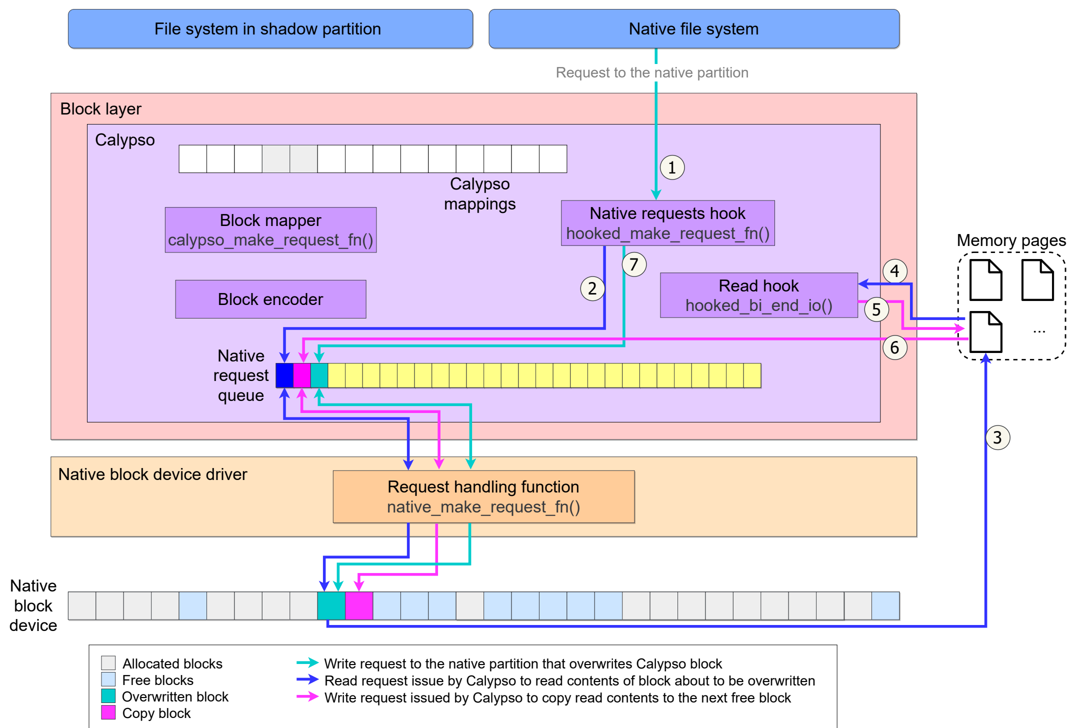

Figure 4.4: Calypso request interception and handling overwrites. The cyan arrow shows how a write request to a block mapped by Calypso is handled to avoid overwrite. In 1), the request to the native file system is intercepted. This request is put on hold, and in 2), Calypso issues a read to the block to be overwritten. That read copies the contents of the block to a page in memory as shown in 3). In 4), Calypso intercepts the termination function of the read request, activating a second write request, represented in blue, to copy the contents in the memory page, pinned by 5). Step 6) represents the second write request being placed in the native requests queue, where it is going to be fetched by the native block device to be processed. Since we already issued the copy request and its data is on a different memory page, we can resume the initial write request identified by 7).

Figure 4.4 presents a visual representation explaining how requests to the native block device are intercepted and the process that needs to be performed in case of an imminent overwrite.

To perform the described procedure, we need to store the following structures for each copy request: i) the pointer to the `struct bio` instance of the original request, so that we can resume it later; ii) the pointer to the `struct page` instance that we purposely allocated for the read request, for Calypso to reuse it in the write request that copies the data. This way, the data we want to copy is already on the page. We would need to store the contents of the read page otherwise since the contents might no longer be available after the read request terminates; Finally, iii) we need to store the number of the block about to be overwritten so that we know which block the subsequent requests should target.

4.4.4 Cleaning Caches to Retrieve Updated Data

When testing Calypso's ability to copy blocks about to be overwritten, we noticed that when we write to a native block that Calypso is mapping, and we read the next free block where the overwritten data was supposed to be copied on, did not return the copied data. For instance, if we write the string "hello world" to Calypso's block device and overwrite that same block by writing the string "hi cat" to the native block device, when we the next free block, we won't get "hello world" until we flush the page cache.

This happens because we are reading the contents of the cache instead of the contents in the disk, and Calypso bypasses the page cache when it issues the write to the native device charged with writing the copied contents. We are only going to be able to read the right value when the page cache is flushed. The dirty pages are written to disk and a new read request is going to generate a page fault that fetches that block from the disk. This approach is acceptable since that page would only be marked as dirty if the block was overwritten, so we won't lose data.

4.5 Cryptography and Data Hiding

To make Calypso secure and to prevent a forensics inspector from being able to correctly retrieve Calypso's data or meta-data, it needs to rely on several cryptographic primitives. These support concealing all data written to the native partition and retrieving meta-data blocks correctly without having their mappings in memory when Calypso is loaded.

To enable Calypso to properly use the cryptographic primitives, the user needs to insert a password of their choosing each time the Calypso is inserted. This password is going to be used as: i) the initial value to generate a key deterministically; and ii) a seed to feed to a pseudo-random number generator.

4.5.1 Hiding Data Blocks

So far, Calypso writes content to the native blocks as plaintext, which means an adversary with access to the disk can simply observe the data written by Calypso. To hide the Calypso data in the native partition, we need to use encryption or other data hiding techniques such as steganography. For simplicity, we only implemented encryption, which entails challenges since we cannot store the key on the machine. This means we need to generate the same key every time Calypso is loaded.

Why not use an encryption file system: Some systems, like Residue-free computing [46], mount an encrypted file system (ecryptfs [76]) on top of the file system with the contents to be hidden. However, encryption at the file system level changes leads to sensitive leaks, i.e. changes in timestamps and directory tree structure and unencrypted metadata that can be observed directly through raw disk analysis, besides increasing the size of files[17]. To preserve the deniability of the system, using encryption at the block layer is ideal, such as dm-crypt[18], a disk encryption subsystem that uses the Linux Kernel Crypto API[19]. In terms of performance, dm-crypt proved to be closer to a system with no encryption than ecryptfs[20,21]. Our encryption solution is based on dm-crypt and the Linux Kernel Crypto API.

Encrypting and decrypting data blocks: When Calypso is making changes to the requests structure to redirect them to the native device, two possible actions can be made: i) in case of a write request, it replaces the data to be written with the encrypted version of that same data; and ii) in case of a read request, it replaces the read encrypted data with its decrypted contents. This way, the native device will

[17] `https://mikemabey.com/blog/2017/08/ecryptfs_filenames.html`
[18] `https://elixir.bootlin.com/linux/v5.4/source/drivers/md/dm-crypt.c`
[19] `https://www.kernel.org/doc/html/v5.10/crypto/index.html`
[20] `https://www.phoronix.com/scan.php?page=article&item=ext4-crypto-418&num=1`
[21] `https://www.mayrhofer.eu.org/post/ssd-linux-benchmark/`

only have access to the encrypted data, while the user only has access to the plaintext data. We are using the kernel's implementation of the symmetric block cipher AES, in Cipher Block Chaining mode (CBC), which requires using the same initialization vector for encryption and decryption, thus we use a hardcoded value since it does not need to be a secret. However, we need the encryption and decryption key to be generated deterministically with a secret from the user. Otherwise, the key generated to encrypt the data would not be the same as the key generated to decrypt the data.

Generating an encryption key: To allow only the rightful user to deterministically encode and retrieve blocks using encryption, the password that the user inputs to the system is going to be used to generate an encryption key. To do this, we used the kernel's implementation of the *HMAC-based extract-and-expand key derivation function* (HKDF) [77] of the fscrypt file system[22], based on a hardcoded salt and SHA512 HMAC, which generates a 512-bit key.

4.5.2 Storing and Retrieving Meta-Data

Calypso stores the meta-data in its own mapped blocks. When Calypso is loaded, it can easily know the location of the mapped blocks due to its data structures in memory, so data blocks are easily stored and retrieved using encryption and decryption, respectively. However, once Calypso gets unloaded, those structures are lost. We need to provide a way for Calypso to determine the first block of meta-data every time it is reloaded. This way, it can reconstruct the memory structures that allow mapping blocks. The second challenge presented lies in detecting whether the retrieved blocks actually represent Calypso's meta-data and if they were written by Calypso, for this user. Some parts of this challenge are already solved because: i) Calypso blocks can only be decrypted by the right key, thus if Calypso did not issue the blocks, the decrypted contents would not make sense; and ii) Calypso establishes a specific structure for each block of meta-data, as detailed in Section 3.6, so Calypso can determine whether the decrypted contents follow that structure. The only part that remains to solve is knowing whether the block was corrupted, which we accomplish by checking the integrity of every meta-data block.

Deterministic generation of the first meta-data block number: Passing a *seed* to a pseudo-random number generator (PRNG) allows obtaining the same sequence of pseudo-random numbers every time. We exploit this concept to generate the numbers of the blocks in which we are going to attempt to store and retrieve from the first block of meta-data. PRNGs have a period, so the sequence will eventually start to repeat itself, which we contradict by increasing the seed and feeding it again to the PRNG. The period of the PRNG depends on the length of the initial seed. The number of the block to use is going to be the module between the pseudo-random number and the number of blocks in the native partition. We did not use the kernel's source code because we only found a strong random number generator that uses sources of randomness from the computer's environment. We required a PRNG that accepts a seed to generate a deterministic sequence of pseudo-random numbers, so we used an implementation of the algorithm "Mersenne Twister" [78][23], which we minimally changed to fit the kernel environment.

[22] `https://elixir.bootlin.com/linux/v5.4/source/fs/crypto/hkdf.c`
[23] `https://github.com/ESultanik/mtwister`

Checking the meta-data block's integrity and hash: To ensure every block has integrity, we need to perform the following actions: i) when we are retrieving the blocks, we should calculate the hash of the block's contents until the byte where the hash starts and compare that with the hash field of the block; and ii) in the case where we are encoding the meta-data in the blocks, we should calculate the hash of the entire block contents until the byte where the hash field starts. Then, copy the calculated hash into the bytes corresponding the hash fields. To compute these hashes, we applied the SHA256 algorithm using the Linux Kernel Crypto API.

4.5.3 Meta-Data Retrieval Algorithm

Every time Calypso is loaded, it needs to check if there is shadow partition's state to recover. Otherwise, all the data previously in the shadow partition could be lost, invalidating the user's previous sessions. This state can be recovered by retrieving the contents of the Calypso meta-data blocks previously encoded in the free native blocks. However, this is a complex process: i) generating the right decryption key; ii) finding the first block containing meta-data, which entails decrypting the contents of a sequence of pseudo-random blocks until we find one with the correct hash; iii) retrieving the remaining meta-data blocks and reconstituting the previous state of the shadow partition from the retrieved meta-data. A simplified version of the algorithm for retrieving Calypso's meta-data is described in Algorithm 1 and the main procedure in line 26. The meta-data fields are detailed in Figure 3.4. It is organized as follows:

- *Setup to find initial block and decoding the contents:* It starts by generating the encryption key using the HKDF algorithm that generates keys from the user password. After this, it finds the first block of meta-data (procedure described in line 1) by generating a pseudo-random number sequence by passing a seed to the PRNG.

- *Finding the first meta-data block:* This sequence is generated until a block's hash is verified or until the number of iterations reaches the hardcoded parameter that limits the number of times we can check for meta-data, otherwise there was the possibility of never finding the initial meta-data block which would exhaust the kernel resources. To verify the first meta-data block's hash, we issue a read to the respective block and decrypt its contents. Then, we hash the first 4064 bytes of the block and check if the hash matches the last 32 bytes of the block, as described in line 13.

- *Retrieving remaining blocks:* If a valid first block was found, we proceed to retrieve the remaining meta-data blocks to reconstruct the meta-data. As we retrieve meta-data blocks, we will populate the meta-data to native block mappings with the values from the next block field so that these mappings can be reused when Calypso is unloaded.

- *Recovering state from retrieved meta-data:* If the procedure to retrieve a block returns the first parameter as -1, then it means we found a corrupted block and it will not be possible to reconstruct the meta-data and a new install of Calypso is required. Otherwise, it will continue to loop until the next block field is -1. Finally, we can recover the state by populating the block mappings and comparing the bitmaps to check if any of the mapped blocks might have been corrupted. This

Algorithm 1 Meta-data retrieving algorithm

```
1: procedure FINDFIRSTBLOCKTORETRIEVE(virtualToNativeBlockMapping, bitmap)
2:     prng ← SEEDRAND(seed)
3:     metadata ← ""
4:     ret ← 0
5:     while ret ≠ 0 and iter < MAX_ITERS do
6:         randomNum ← GENRANDLONG(prng)
7:         randomBlock ← randomNum mod totalNativeBlocks
8:         ret, _, metadataSplit ← RETRIEVEBLOCK(iter, virtualToNativeBlockMapping, bitmap)
9:         metadata_to_physical_block_mapping[0] ← randomBlock
10:        firstBlockNum ← randomBlock
11:    end while
12: end procedure
13: procedure RETRIEVEBLOCK(curBlock, virtualToNativeBlockMapping, bitmap)
14:    metadataSplit ← READPAGE(curBlock)
15:    metadataSplit ← DECRYPTBLOCK(metadataSplit)
16:    hash ← HASHBLOCK(metadataSplit)
17:    if hash ≠ metadataSplit[4064] then
18:        return −1, −1, metadataSplit
19:    end if
20:    nextBlock ← metadataSplit[4056 : 4065]
21:    if nextBlock ≠ −1 then
22:        metadataToNativeBlockMappings[blockIndex + 1] ← nextBlock
23:    end if
24:    return 0, nextBlock, metadataSplit
25: end procedure
26: procedure RETRIEVEMETADATA(virtualToNativeBlockMapping, bitmap)
27:    encryptionKey ← HKDF(password)
28:    ret ← FINDFIRSTBLOCKTORETRIEVE(virtualToNativeBlockMapping, bitmap)
29:    nextBlock ← 0
30:    metadataSplit ← ""
31:    metadata ← ""
32:    while nextBlock ≠ −1 do
33:        ret, nextBlock, metadataSplit ← RETRIEVEBLOCK(virtualToNativeBlockMapping, bitmap)
34:        metadata ← metadata + metadataSplit
35:    end while
36:    if ret ≠ 0 then
37:        RECOVERSTATE(virtualToNativeBlockMapping, bitmap)
38:    end if
39:    COMPAREBITMAPS(virtualToNativeBlockMapping, bitmap)
40: end procedure
```

solution for detecting block corruptions is merely illustrative to test Calypso's ability to recover state from previous sessions. Other systems like Artifice [9] proposed effective schemes for a self-repair process that determines whether a block encoding hidden data has been overwritten while the system was not inserted in the kernel, through the usage of a checksum for each block. The overwritten blocks are then reconstructed and remapped to new locations on the disk.

4.5.4 Meta-Data Hiding Algorithm

Every time Calypso is unloaded, it needs to persist the state in the shadow partition, so that the session can be recovered once Calypso gets reloaded. To do this, the contents need to be split into several

blocks, and each block should contain enough information to ensure the block's integrity and to point to the next block where meta-data continues. This entails determining in which native free blocks we should encode each meta-data block. A simplified version of the algorithm for hiding Calypso's meta-data is described in Algorithm 2 and the main procedure in line 25. The meta-data fields are detailed in Figure 3.4. The algorithm is structured in the following way:

- *Setup to encode first meta-data block:* The first step is combining the two data structures to persist. Then, we generate the encryption key from the user's password using the HKDF algorithm. Subsequently, in line 3, we test if we already have a mapping to the first block to encode meta-data.
- *Finding the location of the first meta-data block:* Subsequently, we test if we already have a mapping to the first block to encode meta-data, described in line 3. If we don't, we need to find the first block to encode (using the procedure in line 1) using the seeded pseudo-random number generated. This function generates a sequence until the modulus between the pseudo-random number and the total native blocks corresponds to the number of a free native block.
- *Encoding meta-data in the blocks:* Next, we combine each meta-data split, along with the next block number and the hash of those fields, encrypt the resulting block and issue a write request, as shown in line 15. The latter is executed until there is no more meta-data to persist.

In sum, encoding and decoding meta-data is a vital part for Calypso, consisting of solving several different challenges that ensure the shadow partition progresses across multiple Calypso executions.

4.6 Entropy and Floating-Point in the Kernel

Calypso inevitably changes blocks belonging to the native file system to accommodate the shadow partition. However, we can limit these changes to make them plausibly deniable. For instance, there are several regular changes to the native free blocks that regular users may not even notice, i.e. temporary system files, installing and removing software, watching streamed content. In this section, we will discuss the implementation challenges of classifying the entropy of the native free blocks, which is the chosen metric to limit the blocks that can be used to encode Calypso's data.

We started by implementing the Shannon entropy formula described in Equation 3.1, within the algorithm to classify the entropy of disk blocks, as a user-level C program. This way, it is easier to test and it can be used as a standalone tool to classify the blocks of any disk in a Linux system.

This implementation consists of reading a 4096-byte block at a time from a given disk, counting the number of occurrences of each possible value for a byte in each block. The number of occurrences divided by the amount of bytes in the block corresponds to the probability of the value for that byte to occur within the block and apply the Shannon entropy formula described in Equation 3.2.

The main drawback of this implementation is the overhead, so it is not appropriate for big disks. To mitigate this, we used multiple threads, which is not problematic in terms of concurrency to the disk file since we have one file descriptor per thread. Nonetheless, only the native free blocks are going to be classified when Calypso is loaded, which is not problematic if it takes longer.

Algorithm 2 Meta-data hiding algorithm

```
1: procedure FINDFIRSTBLOCKTOENCODE(virtualToNativeBlockMapping, bitmap)
2:     randomBlock ← metadataToNativeBlockMapping[0]
3:     if randomBlock not assigned then
4:         prng ← SEEDRAND(seed)
5:         randomNum ← GENRANDLONG(prng)
6:         randomBlock ← randomNum mod totalNativeBlocks
7:         while not TESTBIT(randomBlock, bitmap) do
8:             randomNum ← GENRANDLONG(prng)
9:             randomBlock ← randomNum mod totalNativeBlocks
10:        end while
11:        metadataToNativeBlockMapping[0] ← randomBlock
12:        UPDATEBITMAP(bitmap, randomBlock)
13:    end if
14: end procedure
15: procedure ENCODEBLOCK(metadataSplit, virtualToNativeBlockMapping, bitmap)
16:    nextBlock ← metadataToNativeBlockMapping[block_index + 1]
17:    if nextBlock = −1 and nextBlock ≠ lastBlock then
18:        nextBlock ← GETNEXTFREEBLOCK(bitmap)
19:        UPDATEBITMAP(bitmap, nextBlock)
20:    end if
21:    HASHBLOCK(metadataSplit)
22:    ENCRYPTBLOCK(metadataSplit)
23:    WRITEPAGE(metadataSplit)
24: end procedure
25: procedure ENCODEMETADATA(virtualToNativeBlockMapping, bitmap)
26:    metadata ← virtualToNativeBlockMapping + bitmap
27:    encryptionKey ← HKDF(password)
28:    firstBlockNum ← FINDFIRSTBLOCKTOENCODE(virtualToNativeBlockMapping, bitmap)
29:    while metadata not empty do
30:        metadataSplit ← metadata[: 4056]
31:        ENCODEBLOCK(lastBlock, metadataSplit, virtualToNativeBlockMapping, bitmap)
32:        metadata ← metadata[4056 :]
33:    end while
34: end procedure
```

Another aspect to take into account is that we are not classifying entropy at the file level, which limits our options of using steganography. One other problem created by this is that, on a formatted disk, if the file does not fulfill all the blocks it is assigned to, for instance, an encrypted file that only takes up half of its last assigned block, that last assigned block can be detected as a low entropy or plain text block. Or worse, it can take a bigger part of the last block and still be detected as high entropy level block, however, the entropy is not even among the entire block. Nonetheless, we assume this situation cannot cause visible deviations of the entropy, at least for higher entropy thresholds.

Then, the second approach is to take the user-level program to the kernel to classify only the unused blocks of the native file system so that Calypso uses only these to hide the data blocks. The initial challenge of the latter is due to being difficult and non-advisable to use floating-point in the kernel, which is required for a direct application of Shannon's entropy formula in Equation 3.2.

Not all CPUs were designed with a floating-point unit, and since floating-point operations are seldom used and expensive, even CPUs with floating-point unit keep it logically separated[24]. For efficiency, the

[24] http://porterchen-note.blogspot.com/2011/03/linux-kernel-and-floating-point.html

floating-point unit states are not saved during every context switch (floating-point state is relatively bigger compared to integer state). So, code containing floating-point operations can be interrupted midway, causing errors. Some architectures[25] require wrapping all floating-point operations inside the kernel with `kernel_fpu_begin()` and `kernel_fpu_end()` macros, which allow restoring the floating-point context and prevents its corruption by disabling preemption, but brings extra overhead [79]. Furthermore, recent kernel versions no longer export these macros[26] due to possible complications that can occur when using floating-point arithmetic, which can be easily avoided through the use of fixed-point arithmetic.

So, we adapted the calculation of the entropy to be able to perform it with integer values, through the usage of fixed-point arithmetic using a scaling factor that allows representing entropy values as integers [80, 81], which also has better performance than floating-point arithmetic.

Summary

This chapter detailed the implementation of the Calypso prototype and the technical challenges faced to achieve the desired result. We start by giving an overview of the kernel I/O request processing mechanisms, followed by Calypso's requirements to be correctly executed. After this, we explained what needs to be done to retrieve the native free blocks from the Ext4 file system data structures. Then, we focused on demonstrating how Calypso handles the received requests by remapping them to the native block device and how it intercepts the requests to the native block device. Finally, we present the cryptography algorithms used to support hiding Calypso's data. The next chapter presents a comprehensive experimental evaluation of the developed prototype.

[25] https://yarchive.net/comp/linux/kernel_fp.html
[26] https://lwn.net/Articles/643235/

Chapter 5

Evaluation

This chapter describes the experiments that we performed to evaluate the proposed solution. It starts by explaining what we want to accomplish along with four evaluation domains (Section 5.1). Then, it describes the methodology and respective experimental results regarding whether the executed programs within Calypso have their *functionality* preserved (Section 5.2). Section 5.3 discusses the *isolation* properties of our system in terms of how it prevents changes to the native file system. Section 5.4 assesses the *security* of Calypso by measuring how the entropy of free blocks is affected as a result of running our system. Lastly, Section 5.5 evaluates the *performance* of our system by measuring the latency and throughput of file system accesses by resorting to a benchmark suite.

5.1 Evaluation Goals

Before presenting our experiments and results, we start by clarifying the domains that we focus on to assess whether Calypso achieves the originally intended design goals. These domains are as follows:

1. **Functionality** consists of showing that Calypso can sustain the execution of PET applications and ensure that these applications can progress as expected with regular user utilization. We expect to observe no file system errors resulting from the Calypso block device where the PET applications are installed and running from.

2. **Isolation** ensures that no persistent observable traces leak from Calypso's execution environment to the native file system. In other words, all accesses by a PET application to the Calypso block device should not cause any alterations in the allocated data structures (both data and metadata) belonging to the native file system (e.g., files' timestamps and content blocks). Therefore, whenever a forensics inspector observes these data structures from disk snapshots, they cannot obtain evidence of what is installed and executed from Calypso.

3. **Security** measures the ability that Calypso has to modify free blocks of the native file system while making these changes unobservable in the eyes of a forensics inspector. Keep in mind that Calypso is going to parasite on the blocks that are currently not in use by the native file system.

Therefore, by analysing such blocks, the forensics inspector should not be able to distinguish between free blocks that store Calypso data and meta-data, from blocks that do not.

4. **Performance** measures the file system overheads incurred by PET applications when running from Calypso partitions. Achieving good performance is a decisive factor so that Calypso can be a usable tool. Yet, given that the main focus of Calypso is the protection of the user, some slowdown is to be expected. Ideally, we expect to make the following observations: i) Calypso does not significantly affect the native file system access times; and ii) the performance of the PET applications does not suffer notable overhead.

We purposely do not study several other aspects of our system, such as resilience to block overwrites. The reason is that some of these aspects were not the focus of our solution; these are problems already solved in previous work and described in Chapter 2. The following sections describe the methodologies used to evaluate Calypso in the above four domains and present our findings.

5.2 Functionality

We test the Calypso functionality by monitoring the execution of representative PET applications and assess if Calypso can provide the storage resources to accommodate the installation and execution of these applications. Next, we clarify our relevant metrics and methodology, and then present our results.

5.2.1 Metrics and Methodology

Metrics: To analyse whether the functionality of a PET application is maintained when executed on Calypso, we measure the *disk usage* evolution of Calypso when executing a given PET. Another part of understanding how the tools are using the disk is understanding how the block allocation is done and which of the native free blocks are being used. Then, we compare the allocated blocks with the size in megabytes (MB) that each tool occupies and represent the number of blocks as percentages out of all the blocks in the native partition so that the reader can have a view closer to the units that users are used to observing of Calypso's impact in the usage of each PET.

PET selection: To evaluate our system, we selected three PETS that: i) are widely used, ii) are recommended for adoption by privacy-conscious users[1,2], and iii) can be fully configured or recompiled, such as open-source software. We also want them to be representative of different disk workloads and purposes of use. For these reasons, we chose the Tor browser[3], Signal[4] and Megasync[5]. This way, we cover a browser that downloads internet resources such as HTML pages, a messaging application that keeps messages, and a file storage and sharing system.

[1] https://ssd.eff.org/en/module/understanding-and-circumventing-network-censorship#3
[2] https://ssd.eff.org/en/playlist/want-security-starter-pack#communicating-others
[3] https://www.torproject.org/
[4] https://signal.org/
[5] https://mega.nz/

Collecting disk usage information: This operation involves checking how many blocks are used when installing each PET in Calypso's shadow partition, how many more blocks are required when we execute the PET, and where that number stabilizes over multiple utilization steps. To obtain a log of all the blocks being used by Calypso for the shadow partition, Calypso's module issues a log with a specific pattern to the "journald" logs. Whenever a request is made to Calypso for a new block, the number of that block is stored in a data structure with a limited capacity of 100 and logs all block when it is full, so that it does not lose logs due to the incapacity of producing logs at higher rates. Besides this, to reduce the error, we implemented an ioctl call to log the block numbers that are pending logging when we finish each step of the utilization script.

Experimental setup: To simplify this evaluation, we used a smaller native partition formatted with an Ext4 file system, of 3.5 gigabytes total size, from which 1.8 gigabytes are used and the remaining 1.6 gigabytes are free. This is equivalent to 937,500 total blocks, from which 479,605 are allocated blocks and the remaining 457895 are free blocks. We consider a block as being equivalent to 4096 bytes, which is also the default block size in the Ext4 file system. These values were taken with the 'df' Linux utility that shows free and available space in file systems and 'dumpe2fs', another Linux utility that prints the superblock and blocks group information on the present Ext4 file system. Then, Calypso is initialized with 300,000 blocks, approximately equivalent to 1.1 gigabytes, which does not take into account the number of blocks to write the Calypso meta-data; for this number of blocks, this meta-data takes 621 blocks. So, in total, the Calypso's shadow partition can occupy at most 300,621 blocks, which always needs to be smaller than the number of free blocks from the native partition, in this case, 457,895.

Emulating PET user actions: To construct the utilization script for each of the three pre-selected PET applications, we had to take into account the characteristics of each application and the workloads that each interaction would produce on the underlying file systems. Next, we explain these characteristics in more detail. (In Appendix A, we provide the full list of user actions that we emulate for each PET.)

- **Tor browser:** We needed to take into account a regular browser utilization by a censored user. A browser takes advantage of multiple client-side persistent storage methods on the user's disk to enhance user experience, like local databases, caches and cookies. Browsers also store cryptographic information such as keys locally for increased security, since they are more vulnerable in servers, and other sensitive data for privacy reasons. When running the Tor browser on Calypso, these files will be stored on a shadow partition.

  Whether the user accesses static or dynamic web pages will also affect the resulting filesystem workload. Static web pages are composed of files that are delivered to all users in the same way, whereas dynamic web pages render content depending on the user's interactions, such as performing a login. These differences can have implications on the amount of data that needs to be fetched for a web page to load. For instance, a static web page will load all resources independently of whether the user wants to access it specifically, while a dynamic page will mostly fetch the resources that are explicitly requested by the user.

  So we needed to choose a set of relevant web pages that are either on the top of accessed

pages by Tor users or that are censored, or both. The OpenNet Initiative (ONI)[6] is a collaborative partnership to investigate Internet filtering and surveillance practices across different countries in the world [82]. We chose the China profile due to having a very sophisticated and widely studied censorship scenario. The selected web pages should also represent a balanced mix of static and dynamic web pages with multiple types of content to be fetched.

- **Signal desktop:** Private messaging apps are in high demand and produce a variety of workloads, such as video calls and voice messages. In particular, Signal is very complex. It claims to perform end-to-end encryption and prioritizes storing data locally instead of servers. This means it is going to manage several local cached data items and keys, increasing its disk footprint.
- **Megasync desktop:** File sharing tools are widely used and privacy dependant since they can be used for several sensitive matters, such as sharing journalists' sensitive reports. Furthermore, they can introduce various file system workloads depending on the nature and quantity of the files to be synced to and from the local machine. We chose to use this with a smaller workload of two folders with a few jpeg images to demonstrate the block evolution with finer granularity. We chose Megasync specifically due to its free plans and availability of the binaries.

5.2.2 Results

In this subsection, we present our main findings after testing each of the pre-selected PET applications on a Calypso shadow partition.

Respectively, each of the following plots shown in Figures 5.1, 5.2, and 5.3 reflect how the block utilization evolves with multiple different actions being executed with each PET, i.e., Tor browser, Signal, and Megasync. Each number of blocks represented in the y axis should be interpreted as $1\text{k}= 1 \times 10^3$ blocks. Annotation labels highlight some of the most interesting steps of the utilization script. Each one contains the action performed in the respective utilization step, along with the exact cumulative number of allocated blocks, specified between parentheses.

Calypso can execute multiple PETs with different disk overloads: In all cases, we observe an initial spike during Calypso setup and tool installation, which can be explained by three reasons. First, the need to format Calypso's shadow partition with the ext4 file system, which allocates blocks for the superblock and its backups, group tables, inode tables and other accounting file system information. Second, installing each tool in the shadow partition requires allocating blocks for them. Third, each tool when first executed stores initial meta-data for usability purposes, such as user accounts or preferences. This is followed by a stabilization in smaller increases in the number of used blocks during a regular utilization of the tool, ending in a final increase at the end of Calypso's execution, where Calypso stores meta-data that ensures persistence across multiple Calypso executions.

In more detail, Figure 5.1 shows the evolution of the block allocation in Calypso during the execution of the **Tor browser**. There is a sharp rise in the number of used blocks during the initial stages of setup and Tor execution, up until searching on search engine. This can be justified by the size and

[6] https://opennet.net/research/profiles/china-including-hong-kong

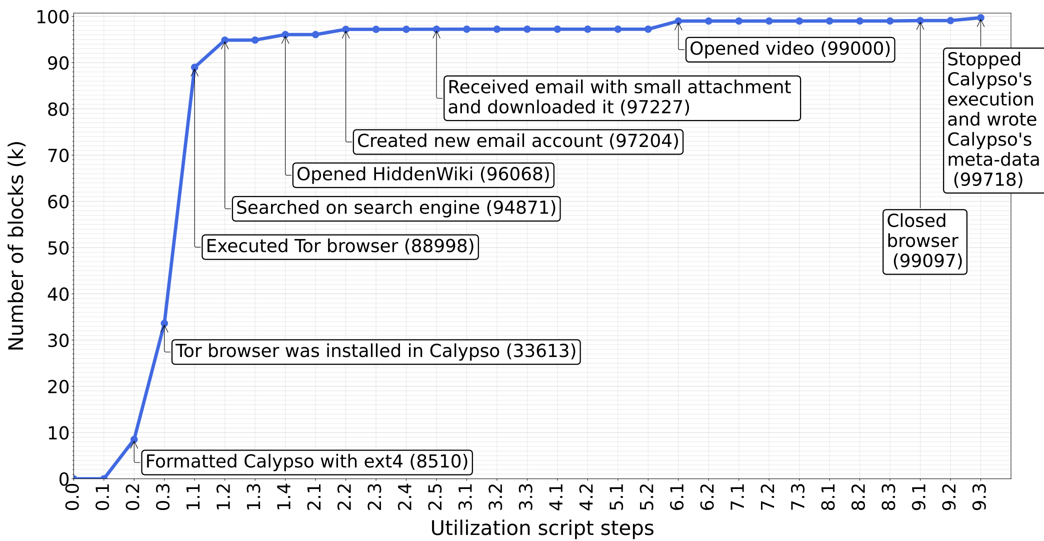

Figure 5.1: Block utilization evolution with time for Tor browser executed with Calypso.

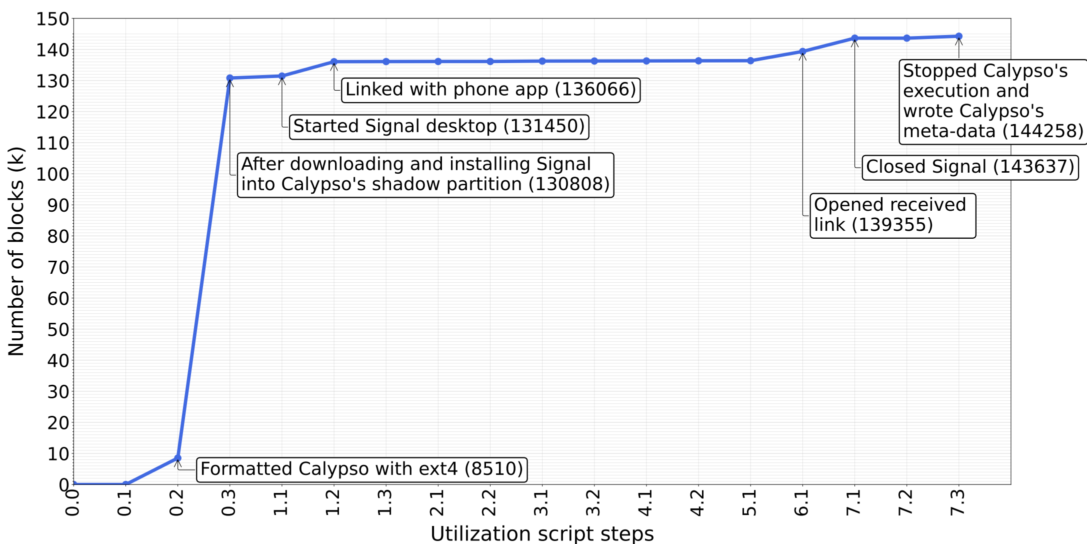

Figure 5.2: Block utilization evolution with time for Signal desktop executed with Calypso.

big number of files occupied by the Tor browser folder. When first executed, the browser needs to get server descriptors with information on Tor relays[7], so Calypso will expand the partition size due to the creation of new files. Then, the evolution in the number of blocks steads in small increases, resulting in two more accentuated ones, namely after opening HiddenWiki, after creating a new email account and after opening a link to a youtube video that requires streaming the video into the machine.

As for the **Signal** desktop app (see Figure 5.2), the execution also developed into an expected high rise when the Signal files were moved into the shadow partition. Starting the Signal app caused a smaller increase in the blocks since it needs to update configurations. Linking with the phone app increases the number of blocks since it needs to download all the data associated with Signal from that device, such

[7] `https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt`

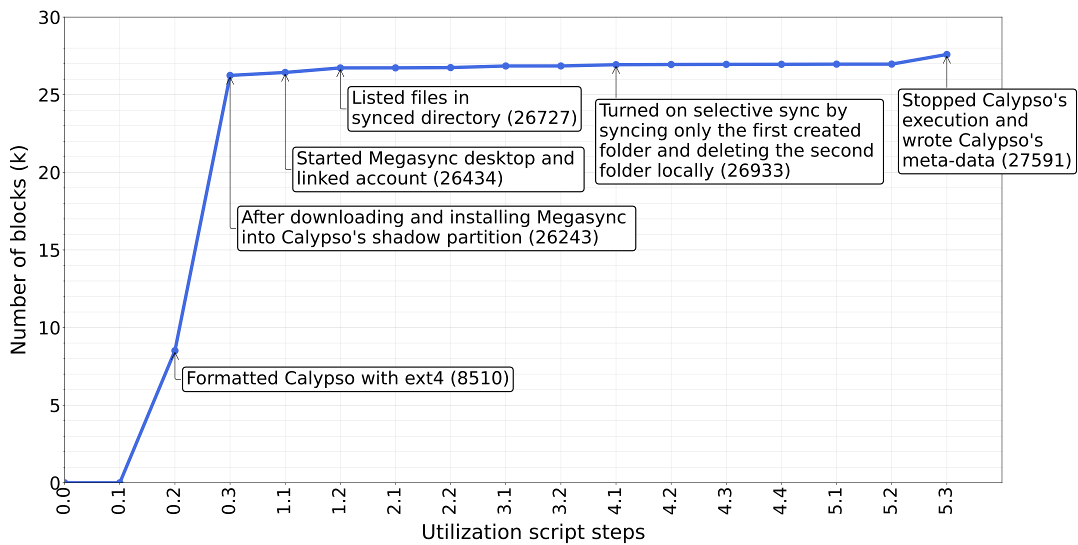

Figure 5.3: Block utilization evolution with time for Megasync desktop executed with Calypso.

as contacts, user configurations and previous conversations, heading towards a smaller steady increase in most successive steps. The final pikes occur when opening a received link in the browser and when Signal is closed, probably due to persisting cached memory information onto files.

Finally, Figure 5.3 represents the block usage evolution using the **Megasync** desktop app. Similarly, it shows two noticeable increases besides formatting the partition at the initial stages, the first being after the Megasync files being moved to the shadow partition, and the second after starting the Megasync app and linking it with the account, the latter because of them needing to download account information from the server. Since we did not have synced folders in this account, it was not a very significant increase. Then, the increase steads in smaller growths, in which two synced folders were created with a few jpeg files inside, one locally and the other one remotely. These lead to a stabilization when selective syncing was turned on by syncing only the first created folder and deleting the second folder locally. The reason why there was not a decrease in the number of blocks is that Calypso's shadow partition works much like a dynamically allocated virtual machine disk, which will not automatically shrink. After this, the next steps did not translate into an increase in the number of blocks because they consisted of adding changes to the unsynced folder remotely.

Calypso allocates blocks using only the native free blocks: We elaborate a visual representation of the blocks allocated by Calypso before and after executing a PET utilization script. Figure 5.4 pictures the initial segment of the native partition state before executing Calypso on the left, and those same blocks after executing Megasync's utilization script, on the right. Both segments of the native partition are represented in the form of a matrix of 300x300 dots, where each dot represents a block in a color elusive to its purpose: i) orange blocks are the native allocated blocks that cannot be used by Calypso; ii) purple blocks represent free blocks of the native ext4 file system; iii) pink blocks act as the Calypso file system's meta-data blocks; iv) cyan blocks are used to store the meta-data that allows Calypso to be persistent; and v) yellow blocks are occupied with data generated by Megasync. The first block is

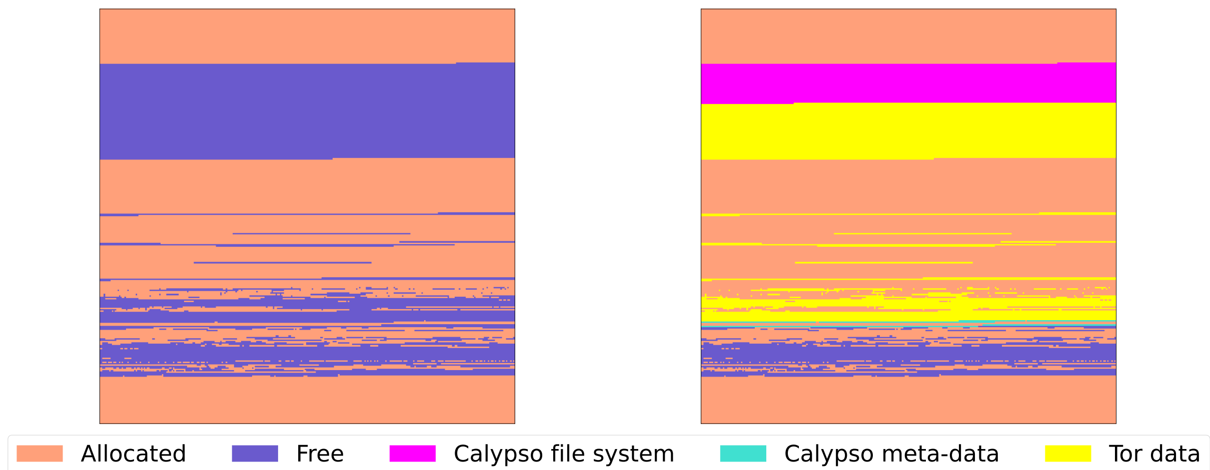

Figure 5.4: Comparison of a part of the native partition state, before and after executing Megasync with Calypso, from block 200 to block 90000, as a matrix of 300x300.

	Initial state		Tor browser		Signal		Megasync	
	%	MB	%	MB	%	MB	%	MB
Allocated blocks	51.16	1873	51.16	1873	51.16	1873	51.16	1873
Free blocks	48.84	1789	38.2	1399	33.45	1225	45.89	1681
Calypso file system blocks			0.91	33	0.91	33	0.91	33
Data blocks			9.66	354	14.41	528	1.97	72
Calypso meta-data blocks			0.07	3	0.07	3	0.07	3

Table 5.1: Blocks distribution in native partition being used by Calypso before and after executing each PET's utilization script.

indexed by line 0, column 0 of the matrix.

The pink, yellow and cyan blocks represent the blocks that were allocated by Calypso to support executing Megasync. As we can see, these blocks overlap exclusively the native free blocks in purple, ensuring that only the free blocks of the native file system are used.

Calypso uses the free space of the native partition efficiently: Table 5.1 shows a summary of the initial state of the native partition used, discriminating the allocated and free blocks ratio and respective space in megabytes (MB). It also shows how the free blocks are affected by the utilization of each PET, and how these blocks are being used, i.e., i) associated with the Calypso file system, ii) for storing PET-generated data blocks, or iii) Calypso's meta-data blocks.

The Calypso file system blocks and meta-data blocks are constant across the three PETs. These numbers are constant because they depend on the size of the native partition and the number of maximum blocks assigned to Calypso, which is the same for the three cases, and bring a non-significant increase in the occupied space: i) only 33 megabytes (MB) for the file system in the shadow partition, representing a small 0.91% of the blocks in the partition; and ii) just 3 megabytes (MB) for Calypso meta-data, equivalent to as little as 0.07% of the total blocks in the partition. Signal was the one that occupied more space, with a total of 528 megabytes of application data, which constitutes 14.41% of all

the blocks in the native partition. If we sum the space occupied by the three types of Calypso blocks, we get 564 megabytes, equivalent to the 144258 blocks shown in Figure 5.2.

In summary, our experiments show that Calypso preserves the functionality of client PET applications. Calypso allows executing multiple PETs by taking advantage of the free blocks belonging to the native partition, which are going to be allocated on demand according to the requirements and particularities of the functionalities provided by each PET, without significantly increasing their overall size.

5.3 Isolation

By testing the isolation properties of Calypso, we aim to assess whether persistent observable traces are leaking from Calypso's execution environment to the native file system. Next, we present our approach for analysing these properties, and then we present our findings.

5.3.1 Metrics and Methodology

Metrics: When focusing on potential traces left onto the native file system, we consider the following main metrics to assess the isolation: the existence (and number) of accesses to the native file system, the number of different files accessed, the category of the files accessed, and whether the accessed file is in-memory only or is written to disk, even if temporarily. Ideally, if the isolation is perfect, a forensic analyst should not be able to detect any modifications to the native file system, including to the content or meta-data (e.g., access timestamps) of the files therein located.

Approach for assessing isolation: The ideal approach would be to take multiple images of the native file system, one before executing Calypso and another one after executing each tested PET. Then, detect all the differences between them, and mark the relevant ones, i.e., application-specific caches.

However, this methodology demanded more time and resources than what we had, so, for this evaluation, we adopted a simpler methodology.

First, we selected one of the PETs, in this case, we chose the Tor browser since it is the most familiar one to us. Then, we installed and executed it in two scenarios: i) from the native partition; and ii) from Calypso's shadow partition.

Tracing accesses to the native file system: To trace all the file system interactions, we used the Linux Auditing System (auditd)[8], and recorded three different logs: i) initial state, without executing the Tor browser; ii) executing the Tor browser in the native partition; and iii) executing the Tor browser in Calypso's shadow partition. Then, we generated summary reports on the accessed files using the "aureport" tool, obtaining all the accessed paths during a 10 minute time period with the respective number of accesses to each path. Then, we divided the accessed paths into the following categories:

a) **Temporary memory only:** includes paths starting with /dev, /proc, /run and /sys because they are mounted on memory-only file systems such as tmpfs, procfs and sysfs, which means their information is not written to disk and is erased every time the system is shut down.

[8] `https://capsule8.com/blog/auditd-what-is-the-linux-auditing-system/`

b) **Temporary in disk:** pertains to paths starting with /tmp and /var/run, which are mounted as regular ext4, writing temporary data to disk. This data is deleted with a certain regularity depending on the system configurations (generally at every system boot). However, this data is not necessarily fully wiped from the disk; it is just unlinked from the file system structures, meaning it might still be recoverable with file carving techniques.

c) **System config and OS resources:** takes into account the paths starting with /etc, along with / (the root directory for the whole system), where are located system-wide configuration files and the start of the file system hierarchy.

d) **Binaries, libs and apps:** considers paths that mainly include /bin, /lib, /opt, /sbin, /usr, /var/-cache and /var/crash which mostly contain binaries, shared libraries, linux commands, software packages, applications and respective data.

e) **User data:** takes into consideration paths that start with /home and /root, containing the data and configuration files for each different user.

f) **Calypso:** bears paths starting with /media/virtual_calypso, which includes all the files that are going to be hidden within the free blocks of the native file system, so these changes do not affect the native file system.

g) **Other:** includes the paths not reported as an absolute path by auditd and could not be tracked to a specific directory. We tracked some of these files due to their inode numbers, but for instance, the path '..' can correspond to several different inode numbers, and since the system has multiple file systems mounted, the same inode can correspond to several different files in different file systems.

5.3.2 Results

This subsection presents our main findings, consisting of surveys of the accesses audited during the experiments described above.

A summary of our results is depicted in Table 5.2. The lines indicate all the file categories pointed out above. The accesses count refers to the number of file system-related system calls that were audited during the associated period, to files of each established category. The columns of the different file count refer to the number of different files that were accessed, according to each file category.

To help the user navigate the displayed information, we provide an example: The line containing the results for user data files presents 0 accesses during the initial state, due to no user application being executed during this period. These would be the ideal results for isolation during the execution of a user application such as the Tor browser. The second period reports an increase from 0 to 2,553 accesses, meaning that we can expect the Tor browser executed natively to cause a similar increase.

The following period, where we executed the Tor browser from Calypso's shadow partition reveals an increase from 0 to 109 accesses, these targeting 63 different files, meaning that the Tor browser with Calypso is expected to produce about 109 accesses. Besides this, the results in this line allow

File category	Initial state		Tor native		Tor Calypso	
	Accesses count	Different file count	Accesses count	Different file count	Accesses count	Different file count
Temporary memory only	2138	67	4268	742	11377	5298
Temporary in disk	124	2	99	54	618	26
System config and OS resources	448	27	3015	161	2913	145
Binaries, libs and apps	239	62	3276	1389	1516	370
User data	0	0	2553	510	109	63
Calypso	0	0	0	0	318	75
Other	200	1	240	37	421	27

Table 5.2: Files accessed and respective classification during 10 minutes of auditing the system without Tor installed, while executing Tor in the native file system, and while executing Tor inside Calypso.

us to estimate how many accesses Calypso was able to isolate, by subtracting the results from the second period with the results from the third, where we can surmise that Calypso reduced the accesses performed by the Tor browser to user data related files in about 2444. At last, reporting 109 accesses and 63 different files, during the period where Tor was executed with the Calypso support, means that 109 file system-related system calls were issued to 63 different files, without specifying how many times each of these files were accessed.

Calypso notably reduces accesses to files in the native disk partition: As presented in Table 5.2, most of the file system accesses performed by the Tor browser are diverted from the native file system to the Calypso partition. If we sum all the accesses counts for the Tor browser with Calypso, we can observe a higher number of accesses due to the bootstrap and cleanup processes. However, most of those accesses (11,377) happen in memory only, so they are of no concern. The number of accesses to temporary files that go to disk increased from 99 to 618, but the different file count was reduced from 161 to 145. Still, we can improve on these results, but the other accesses to the native file system were heavily reduced compared to executing the Tor browser natively, such as the accesses to binaries, libs and apps, with a reduction from 3,276 to 1,516 accesses and from 1,389 different files to only 370.

Overall, the accesses to files in disk reduced significantly when the Tor browser was executed from within Calypso, which shows Calypso produced an encouraging improvement in the domain of isolation.

Calypso still makes some accesses to the native file system that can be mitigated: In Table 5.3 we show the top 10 accessed files in the scenario where the Tor browser was executed with Calypso by each category. Here, we discuss whether these are relevant accesses that can prove the Tor browser was executed in the system and what we can do to mitigate these traces. We exclude the temporary memory only and the Calypso files since they do not translate to traces in the native file system.

a) Temporary files: The most accessed folder was /var/run/utmp. It stores information on user logins, logouts, system events, system status, system boot time, among others. These events are not directly linked to the execution of the Tor browser[9,10]. We could not identify what /tmp/namespace-dev-9shiTV

[9] https://www.thegeekdiary.com/what-is-the-purpose-of-utmp-wtmp-and-btmp-files-in-linux/
[10] https://www.sandflysecurity.com/blog/using-linux-utmpdump-for-forensics-and-detecting-log-file-

File types	Top 10 accessed files	Number of accesses
Temporary in disk	/var/run/utmp	573
	/tmp/namespace-dev-9shiTV/dev/	11
	/tmp/namespace-dev-9shiTV/dev/char/	7
	/tmp/	2
	/tmp/namespace-dev-9shiTV	2
	/tmp/namespace-dev-9shiTV/	2
	/tmp/namespace-dev-9shiTV/dev	2
	/tmp/namespace-dev-9shiTV/dev/pts	1
	/tmp/namespace-dev-9shiTV/dev/char	1
	/tmp/namespace-dev-9shiTV/dev/char/5:2	1
System config and OS resources	/	565
	/etc/fstab	144
	/etc/ld.so.cache	109
	/etc/group	51
	/etc/passwd	43
	/etc/localtime	37
	/etc/login.defs	35
	/etc/fonts/conf.avail	18
	/etc/fonts/conf.avail/10-antialias.conf	18
	/etc/fonts/conf.avail/10-autohint.conf	18
Binaries, libs and apps	/usr/share/dbus-1/system-services	97
	/lib/x86_64-linux-gnu/libc.so.6	32
	/lib/x86_64-linux-gnu/libpthread.so.0	27
	/lib/x86_64-linux-gnu/libdl.so.2	27
	/lib64/ld-linux-x86-64.so.2	26
	/usr/share/locale/locale.alias	23
	/lib/x86_64-linux-gnu/libselinux.so.1	18
	/lib/x86_64-linux-gnu/libpcre2-8.so.0	18
	/usr/lib/locale/locale-archive	18
	/lib/x86_64-linux-gnu/liblzma.so.5	16
User data	/home/daniela/.cache/tracker/	8
	/home/daniela/.cache/tracker/meta.db	5
	/home/daniela/.local/share/gnome-shell/	4
	/home/daniela/vboxshare	4
	/home/daniela/vboxshare/logging/cat-4611189__480.jpeg	4
	/home/daniela/.local/share/tracker/data/	4
	/home/daniela/.cache/tracker/meta.db-wal	4
	/home/daniela/.local/share/gnome-shell/application_state	3
	/home/daniela/.cache/mesa_shader_cache/c2/	3
	/home/daniela/.cache/mesa_shader_cache/ec/	3

Table 5.3: Top 10 files in disk accessed by Calypso when executing the Tor browser, by file type.

relates to, but it seems to be the temporary directory and files written by a character device. Accesses to /tmp/, as long as they remain unrelated to Calypso or the executed PET, are not of concern. One possible mitigation for the high number of accesses to temporary files is to execute containers on top of Calypso so that the accesses are limited to the writable file system in the container image, and the container image is stored on a Calypso shadow partition.

b) System configuration files: Files from this category in /etc such as fstab, group, passwd, localtime and login.defs are regular accesses that do disclose the usage of particular applications. The root directory / is also a directory that is not associated with the usage of particular applications. On the

tampering/

other hand, ld.so.cache determines the libraries required for a program to run[11], meaning that it can be linked to specific applications execution, hence we should repackage the Tor browser to use specific local libraries to Calypso without using the native's dynamic library configuration and libraries. Similarly, the font configuration files in /etc/fonts can be used for web browser fingerprinting [83], so, to improve, we should repackage the Tor browser to include all the font associated files in its directory.

c) Binaries, libs and apps: As far the files in this category go, /usr/share/locale/locale.alias and /usr/lib/locale/locale-archives are not relevant for the detection of specific apps' execution, they simply contain important locale specific data for the system to work. The remaining files are shared libraries that should be packaged with Calypso's shadow partition to avoid dependencies in the native file system and analysis of the accesses patterns or tracking which processes accessed them. Combining Calypso with containers can also solve this issue, depending on the containers' configurations.

d) User data: Lastly, files in /home/daniela/vboxshare were manually accessed during the utilization of the Tor browser to upload the picture named cat-4611189480.jpeg. The files under /home/-daniela/.cache/tracker and /home/daniela/.local/share/tracker are associated with the GNOME Tracker searching tool [84][12] and have purposes such as index content from the home directory so applications can provide instant search results, which can compromise the deniability of Tor's execution. Likewise, application_state contains information on the execution of desktop applications. The directories in /home-/daniela/.cache/mesa_shader_cache are associated with a graphics library that is not directly associated with the Tor browser. Much like the above, containers can help prevent this trail.

In summary, our results show that Calypso significantly reduces the on-disk accesses to the native file system. However, this isolation is not perfect given that some file accesses generated by the PET application may target certain paths that are mounted on persistent locations of the native file system. As a countermeasure, the remaining relevant accesses can be mitigated by repackaging the targeted PET application or by containerizing the applications executed on top of Calypso.

5.4 Security

The security of Calypso relies on its ability to remain non-observable during a physical inspection of the device. So, we need to prove that the initial entropy of the partition blocks is preserved after introducing changes by Calypso, namely after the user executes a private session running their preferable PET application. In this section, we start by clarifying our metrics and methodology for accessing the security of our system, and then we present our findings.

5.4.1 Metrics and Methodology

Metrics: To evaluate the security of Calypso, we use the *entropy differential*. This metric translates to how much the entropy of the blocks located on the disk has changed between inserting and removing

[11] https://developer.ibm.com/tutorials/l-lpic1-102-3/
[12] https://sansorg.egnyte.com/dl/cbAeuaGNey

Calypso on the system. The entropy differential is computed by taking two entropy values for each block in two different moments in time: i) one before using Calypso; and ii) one after using Calypso.

The disk changes made by Calypso can be controlled with a parameter named *entropy threshold*. This value decides which of the free blocks are going to be used to encode Calypso's encrypted data: only the blocks with entropy equal to or above a given entropy threshold will be used to store Calypso data. Intuitively, we expect that increasing the entropy threshold value causes a reduction in the number of blocks that will change, thus increasing the security levels. We also expect to observe a trade-off between security and storage capacity: if we set higher entropy threshold values, fewer blocks will be considered safe for usage, which will decrease the storage capacity of Calypso shadow partitions.

Experimental setup: We created four virtual disks of 1 GB each, with all blocks filled with different common file types to simulate slack space in a user's disk: i) plaintext files were taken from the 'Gutenberg ebooks' data set[13], a collection of free English ebooks, into the virtual disk; ii) JPEG image files were taken from the COCO (Common Objects in Context) [85] 2017 Test images data set[14]; iii) a compressed zip file of the images data set; and iv) an encrypted file, generated by encrypting a test file[15] with OpenSSL's DES3 implementation[16], a block cipher in CBC mode.

Measurement of disk block entropy: To measure the entropy of the blocks, we created a simple tool to iterate through all the unused blocks in the native file system's partition and classify their entropy. Then, we encrypted all blocks with entropy equal or above the threshold and executed the tool again. Ideally, we would ask for volunteers with Linux systems to test our entropy classifying tool in their disks and to use publicly available disk data sets, which could reveal important data about the entropy patterns of regular users' disks. However, this approach would require asking special authorizations to use the disk data sets and might bring privacy concerns due to inspecting the disk of the volunteers, so we decided not to perform such tests for now.

5.4.2 Results

This subsection presents our findings. Our discussion next will be based on the measurements of entropy of the blocks located on the four virtual disks.

Most blocks that constitute highly used media files have an entropy close to the entropy of encrypted blocks: In Figure 5.5, we can observe that plaintext files have an average entropy of around 4.5, whereas images, compressed and encrypted files have a high entropy average of almost 8. However, the variability in both images and compressed files is much higher, which means there is a significant number of blocks that is not well characterized by the average entropy. These are very positive results because these files are very common in most personal computers and may even have a certain turnover, which suggests that many usable blocks may be available on personal computers with higher usage.

Table 5.4 presents some additional statistics. These values confirm that image and compressed files

[13] `https://web.eecs.umich.edu/~lahiri/gutenberg_dataset.html`
[14] `http://images.cocodataset.org/zips/test2017.zip`
[15] `http://engineerhammad.com/2015/04/Download-Test-Files.html`
[16] `https://www.openssl.org/docs/man1.0.2/man1/enc.html`

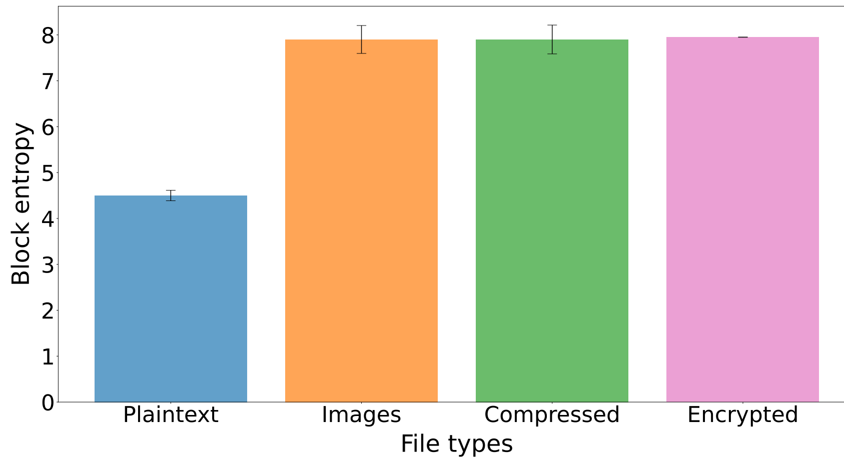

Figure 5.5: Average block entropy and standard deviation in the blocks of 1GB virtual disks simulating unused space, for multiple file types.

File type	Average ± Error	Min	1Q	Med	3Q	Max
Plaintext	4.4987 ± 0.1136	0.2613	4.4293	4.4876	4.5527	5.4699
Image	7.8976 ± 0.3040	0.0000	7.9212	7.9303	7.9370	7.9591
Compressed	7.8964 ± 0.3144	0.0000	7.9213	7.9303	7.9369	7.9612
Encrypted	7.9546 ± 0.0040	7.9334	7.9520	7.9547	7.9574	7.9702

Table 5.4: Statistics on the entropy variation in the blocks of 1 GB virtual disks simulating unused space, for multiple media file types.

have higher variability in entropy than the other file types. However, their average entropy is still on par with the encrypted file's entropy, contributing to a potentially bigger storage capacity. Supported by the high quartiles, these blocks represent a big storage opportunity, with most blocks being usable even for higher entropy threshold values. Expectantly, plaintext files have significantly lower entropy, meaning they may not be appropriate to store Calypso's encrypted data if we want to maximize security. As we expected, encrypted files show a very consistent level of entropy, with a minimum of around 7.93, meaning all encrypted free blocks are usable by Calypso without compromising its non-observability.

Choosing entropy threshold values: To analyse appropriate entropy threshold levels, we established 8 different levels, going from the lowest of 0 to the highest, with a value of 7. Then, we measure the initial entropy of all the blocks in the virtual disks and proceed to encrypt all blocks with entropy equal or above each threshold so that we can measure the resulting entropy of all the blocks, simulating Calypso's behaviour. With this methodology, we produced the following results in Table 5.5 and Figure 5.6.

Each line in Table 5.5 shows important data obtained for each entropy threshold level. The usable blocks count refers to the number of blocks, out of a total of 1,048,576 equivalent to 4 GB, that are usable by Calypso with the respective entropy threshold. The usable blocks count is then shown as a percentage out of the available blocks, shown with the equivalent storage capacity in gigabytes (GB). Then, the average differential shows us the overall difference in entropy after the blocks above a certain

Threshold	Usable blocks count	Usable blocks (%)	Storage capacity (GB)	Average differential	Minimum differential	Maximum differential
0	1048576	100.00%	4.00	0.11225	0.00000	1.00000
1	1048291	99.97%	4.00	0.11198	0.00000	0.87262
2	1048141	99.96%	4.00	0.11187	0.00000	0.74745
3	1047967	99.94%	4.00	0.11175	0.00000	0.62262
4	1047261	99.87%	3.99	0.11139	0.00000	0.49616
5	785484	74.91%	3.00	0.00306	0.00000	0.37172
6	784098	74.78%	2.99	0.00262	0.00000	0.24590
7	782983	74.67%	2.99	0.00244	0.00000	0.12039

Table 5.5: Differential entropy statistics.

threshold have been replaced with encrypted data. The minimum and maximum differential are also very important metrics to show the best and worst-case scenarios, respectively.

For a threshold of 0, 100% of the blocks are usable, whereas, for a threshold of 7, that percentage comes down to 74.67%, which is still very reasonable, with about 2.99 usable GB out of a total of 4GB free blocks. Since the files that compose our virtual disks already had a high entropy, even for a threshold of 0 the differential does not surpass 0.11225, very far from the maximum of 1. This means that in a partition with file turnover, we would be able to replace most blocks without inducing significant changes in the partition's overall entropy. For the maximum threshold, the average differential comes down to almost 0, meaning the disk would maintain approximately the same entropy in all the blocks, with a maximum of about 0.12 differential, which is still low, so no block had significant changes in entropy.

Moreover, there is a distinct turning point with threshold 5. Here, we observe a larger reduction in the usable blocks and the average differential since blocks with plaintext have about 4.5 entropy, as we have seen in Figure 5.5. So thresholds 4 and 5 present the most balanced aspects, where a threshold of 4 maximizes the storage capacity without overlooking security, and a threshold of 5 brings a significant increase in the security, but also significantly reduces the storage capacity. It is hard to say which threshold values are better since it depends on the characteristics of the available blocks. The ideal would be to perform an initial analysis on the partition to determine these parameters.

Most free blocks can be used by Calypso: To get a more detailed view on the differential entropy of our experiments, Figure 5.6 shows the distribution of the entropy differential of the blocks. The lightest squares correspond to values of entropy differential not allowed by the respective threshold. The remaining squares show the value of the probability density function (PDF) for each combination of entropy threshold and differential, which can go from light orange for a PDF close to 0 to maroon for a PDF very close to 1. The PDF translates the probability of a block having a certain entropy differential value with a certain entropy threshold.

By looking at each small square, we can know the probability of finding a block with the respective entropy value. For instance, with an entropy threshold of 0, we have a probability of around 0.025% of

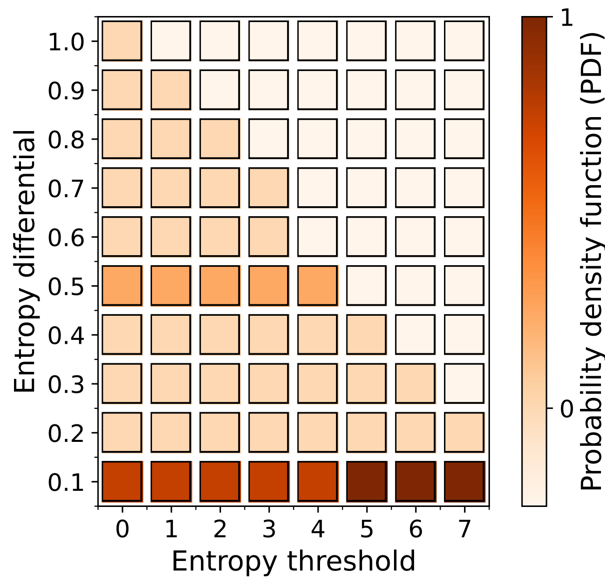

Figure 5.6: Entropy differential PDF for multiple entropy threshold values.

finding a block with the maximum possible entropy differential, implying around 26214 blocks with initial entropy of 0 were encrypted, by observing Table 5.5. For that same entropy threshold of 0, there is a probability of finding a block with entropy differential around 0.5 of 24.443%, which is equivalent to saying 256,303 blocks were encrypted with a resulting noticeable difference in the entropy.

Still, for the lowest entropy threshold, a majority of 74.453% of the blocks did not suffer a noticeable change in entropy. Furthermore, for an entropy threshold of 7, 99.860% of the blocks did not suffer a significant change in the measured entropy. However, blocks that would have generated an entropy differential higher than 2 were not encrypted with an entropy threshold of 7, thus are not usable by Calypso, reducing our storage capacity. Nevertheless, since our data set included mostly high entropy blocks, that still results in 74.67% of the free blocks being usable by Calypso, by observation of Table 5.5.

Here, the distinction from thresholds below and above 5 is also very sharp, showing that with a threshold of 7, there would be no blocks with entropy differential above 0.2, whereas an entropy threshold of 4 would not have blocks with entropy differential above 0.5, but there would be a significant percentage of the blocks with almost 0.5 differential. An entropy differential of 0 would not be appropriate since it does not ensure the user there are not going to be very visible changes to the partition blocks.

To sum up, the higher the entropy level, the fewer blocks we have available to store our data, but the higher the security will be because it will not change the entropy of the blocks as much as a lower entropy level would. However, a lower entropy level will offer higher storage capacity. This will also be a decision the user can make, depending on the level of security they are looking for.

5.5 Performance

To evaluate the performance of Calypso, we run pre-established file system benchmarking tools and test suites on i) the standalone native file system; ii) the native file system with Calypso loaded; and iii) a file system installed on Calypso's shadow partition. We used bonnie++[17], a file system test suite and bench marking tool. After we clarify our metrics and experimental setup, we present our findings.

5.5.1 Metrics and Methodology

Metrics: Latency and throughput are the metrics that most matter to us to measure the overhead introduced by Calypso in the native file system and when directly accessing Calypso's file system. These tests tend to produce highly variable results, so we executed each test 11 times (excluding the first execution results) and obtained their average, along with the standard deviation as the possible error.

Tested accesses: We focused on file system operations consisting of reading and writing the data set in multiple sequential blocks, random seeks, both sequential and random file creation, read and deletion.

Experimental setup: Producing the desired results entails comparing the latency and throughput in different conditions: i) regular usage of the native file system without Calypso being loaded, and compare the obtained performance measurements with performing the same tasks in the native file system with Calypso loaded, where we expect to observe substantial overhead since we are intercepting IO requests to the disk; ii) regular usage of the native file system without Calypso being loaded, and compare the results to the results obtained in equivalent file system operations in Calypso's shadow partition, where we anticipate a performance decrease as well, since we are not only intercepting but also redirecting IO requests to the disk; and iii) comparing the results in Calypso's shadow partition and a ramfs disk to explain why the results were not the expected for the ii) scenario. These results allow us to observe the degradation of performance caused by Calypso.

Besides this, the setup required limiting the virtual machine to have 1 gigabyte RAM for the native file system and the Calypso shadow partition and 3 gigabytes for the ramfs since we executed tests with a 2-gigabyte data set and in-memory file systems require storing all the data in RAM. We also specified 128 files for the file creation tests (both sequential and random file creation).

However, we were not able to display the results for the throughput of the file creation read times since most tests were executed too fast to be considered as reliable results by bonnie++. The same happened for the throughput of random seeks in ramfs.

5.5.2 Results

In this section, we show our main findings after executing the performance tests in the file systems in the native partition, in Calypso's shadow partition and a ramfs drive, and compare these results to conclude whether Calypso is usable in terms of performance.

[17] https://www.textuality.com/bonnie/advice.html

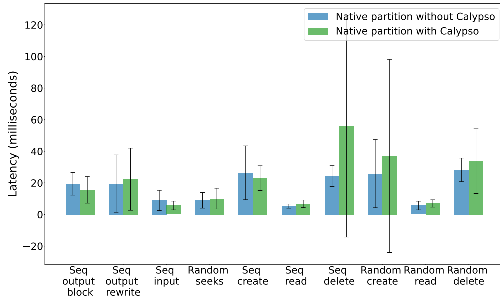

Figure 5.7: Comparison of the latency of the native file system without and with Calypso loaded.

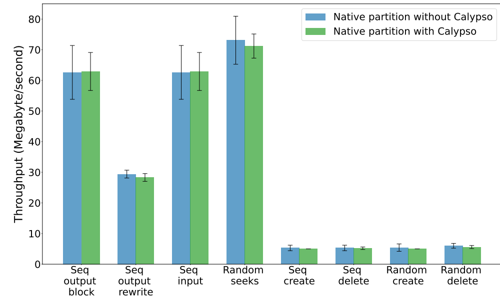

Figure 5.8: Comparison of the throughput of the native file system without and with Calypso loaded.

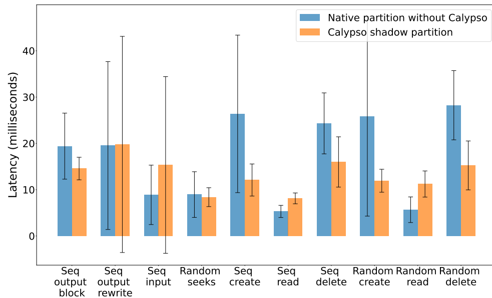

Figure 5.9: Comparison of the latency of the native file system without Calypso loaded and the file system in Calypso's shadow partition.

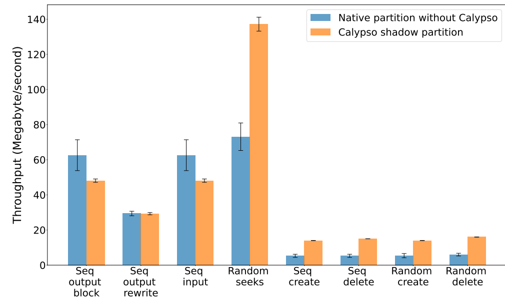

Figure 5.10: Comparison of the throughput of the native file system without Calypso loaded and the file system in Calypso's shadow partition.

The native latency and throughput values are similar with and without Calypso: For the first comparison scenario, where we weight up the latency of the native partition without Calypso against the same one with Calypso, the latency results are very irregular as shown in Figure 5.7, ranging from an error of less than 5 milliseconds in results from operations such as sequential input in the native partition with Calypso, to an error of around 70 ms for the sequential delete in the native partition with Calypso. Overall, the latency values in this scenario do not surpass the 60 milliseconds average. The latency in both cases is very similar, with the native partition with Calypso loaded presenting slightly higher latency, which translates in higher overhead, for the last three types of accesses. This plot seems to have smaller bars only because the scale is smaller.

The throughput for the first scenario represented in Figure 5.8 shows very similar values for both the native partition without Calypso and with Calypso.

Calypso presents similar or better latency and throughput values than the native file system without Calypso: The latency for the second scenario, where we set the latency values of the native partition without Calypso (equivalent to any in-disk ext4) against the values of Calypso's shadow partition, can be seen in Figure 5.9, where the results were also very variable, in particular for the sequential output rewrite in the Calypso shadow partition, presenting an error of over 20 milliseconds. Calypso's shadow partition presents a slightly lower latency overall, which corresponds to a faster performance over the native partition without Calypso.

As far as the throughput for the second scenario goes, illustrated in Figure 5.10, Calypso's shadow partition shows a slight improvement in the throughput when faced against the native partition without

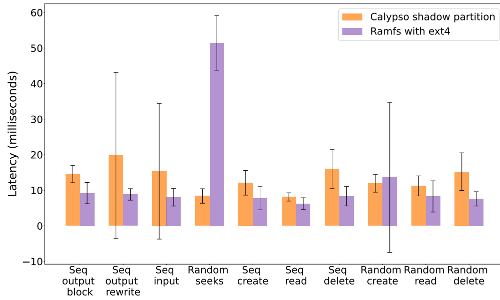

Figure 5.11: Comparison of the latency of the file system in Calypso's shadow partition and a ramfs without Calypso.

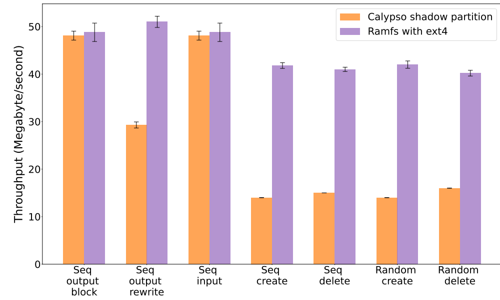

Figure 5.12: Comparison of the throughput of the file system in Calypso's shadow partition and a ramfs without Calypso.

Calypso being loaded, meaning more megabytes can be processed per second.

Calypso reveals to be comparable to a ramfs in terms of latency and throughput values: For the last scenario, presented in Figure 5.11 comparing the latency of Calypso's shadow partition with the latency of a ramfs, we can also observe error values that are quite significant, but Calypso's shadow partition has a higher latency overall, translating into a comparable but slightly lower performance.

Lastly, when comparing the throughput of Calypso's shadow partition against the throughput of the ramfs, shown in Figure 5.12, we can observe that the ramfs has a higher throughput overall, representing slightly better performance.

Calypso does not cause significant overhead on the native file system: Despite this, in Figure 5.7 and Figure 5.8 we can observe that Calypso does not impact the native file system performance significantly, even though a very small overhead can be observed, with a maximum latency difference of at most 40 milliseconds, but this has a high error value. This is due to the interception of the requests to the native partition performed by Calypso, which entails checking updates to the allocated blocks and blocking requests that override Calypso's blocks, while these are moved to other free blocks. The file creation tests show a higher overhead, but this may be misleading due to the higher error values.

Calypso's data allocation and how it is actually faster than the native file system: Contrarily to what we believed initially, Calypso presents better overall performance than the native file system, as shown in Figure 5.9 and Figure 5.10. This is due to the allocation being performed in memory, much like in-memory file systems such as ramfs, hence the comparison between the file system in Calypso's shadow partition and a plain ramfs performed in Figure 5.11 and Figure 5.12. This is not the case, however, for the latency of sequential reads and random reads, probably because of the time to decrypt the data in each block. Calypso's shadow partition presents slightly higher latency and lower throughput than ramfs, the latter in particular for file creation results. For example, Calypso's shadow partition can create around 15 megabytes per second of sequential files, whereas a ramfs creates around 40 megabytes per second. This is to be expected since Calypso requires the extra overhead of redirecting requests to the native partition and encrypting data, writing that encrypted data and reading from disk the encrypted data and transmitting the decrypted data when reads happen.

Concluding the performance evaluation, we can state Calypso presents a very promising perfor-

mance and will not significantly impact the execution of programs, either the ones executed natively or the ones executed from the Calypso shadow partition.

Summary

This chapter introduced the experimental evaluation of Calypso, detailing the experiments conducted to assess whether Calypso fulfills the proposed goals: i) functionality; ii) isolation; iii) security; and iv) performance. Calypso maintains the functionality of a variety of PET applications, providing the necessary storage resources to install and execute these applications while using only the free space of the native file system and leaving the allocated part intact. Calypso significantly reduces the accesses to the native file system caused by the execution of PETs, in particular to binaries, libraries and user-specific application data that could otherwise be linked to the execution of Calypso or PETs. However, it still has a large margin of improvement in isolation, which can be accomplished by offering repackaged PET versions and by combining Calypso with container technologies. Furthermore, we found that blocks with several different media files have comparable entropy levels to the ones of encrypted blocks, providing Calypso with a larger opportunity to use the native free blocks deniably. In combination with the execution of plausible deniable footprints, users can claim usages of the changed blocks unrelated to Calypso. Additionally, we discussed the implications of different entropy threshold values in the resulting storage capacity and entropy differential. Finally, Calypso offers low latency and high throughput, comparable to the ones of in-memory file systems. The next chapter concludes this document by outlining the main findings of the conducted thesis and introducing some directions for future work.

Chapter 6

Conclusions

The recent increase in censorship techniques and degradation of users' privacy has led to the creation of many privacy enhancing technologies (PETs). However, their utilization presents several drawbacks, such as increasing the digital footprint on the users' machines. Thus, users with fear of reprisals refrain from using them, conditioning their freedom online. Furthermore, we presented a study case specifically to investigate the extent of the traces left by one of the most widely used PETs, the Tor browser, in a Linux system. This study confirmed that this tool leaves traces of its execution across the whole system and requires storing several persistent files for security and usability. Additionally, there are several techniques to retrieve artifacts left by the execution of programs from the persistent system of a machine, which can be employed by a forensics investigator, like when a user is crossing border controls.

To tackle this issue, some systems have emerged to enhance the privacy of program execution within the local machine itself, but all fail at concealing the existence of hidden data. Other systems offering plausibly deniable storage have been proposed over the last few years, but most rely on leaving implausible traces such as random writes to blocks and degrade the performance of the system. Alternatively, some systems depend on external storage components or observable adaptations to the system, being a single point of failure to the system's deniability.

In this thesis, we described the design and implementation of Calypso, a deniable steganographic storage system, which leverages using the free blocks of the native system to compose a shadow partition, where data and programs can be stored and executed deniably by performing selective changes to the blocks based on their original entropy, generating opportunity to execute plausible deniable footprints to justify the changes to the disk. This way, we aim to achieve resistance against multi-snapshot attacks. The main technical challenge faced consisted of maintaining the regular system operation while managing its requests to the block device, due to the complex kernel environment.

The experimental evaluation performed on Calypso's prototype proved that it can be used to support the execution of multiple PETs, while significantly reducing the extent of their persistent traces, and without disrupting the system or compromising performance. However, there's still room for improvement by combining Calypso with containerization technologies to completely eliminate traces of programs executed within Calypso. Furthermore, we found that blocks with media files have similar entropy char-

acteristics to the encrypted blocks, giving Calypso a larger opportunity to create deniable changes in the disk, resulting in a larger storage capacity.

An important goal for future work is to combine Calypso with containerization technologies for it to fully provide isolation to PETs execution. Then, we should explore other algorithms to encode data on lower entropy blocks and study the storage capacity provided with multiple entropy threshold values in real user machines. From this analysis, we could also further infer user utilization patterns and resulting entropy levels in the underlying blocks, while studying the implications of using plausible deniable footprints on real user disks. It would also be interesting to extend our evaluation of Calypso's performance by measuring the time Calypso takes to be loaded and inserted for different amounts of virtual and native blocks. Ultimately, it would be useful to solve some implementation limitations, such as finding a way to hook the request handling function in more recent kernel versions.

Bibliography

[1] G. Gebhart and T. Kohno. Internet Censorship in Thailand: User Practices and Potential Threats. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, 2017.

[2] D. Robinson and M. Tannenberg. Self-censorship of regime support in authoritarian states: Evidence from list experiments in china. *Research Politics*, 2019.

[3] R. M. Gabet. A comparative forensic analysis of privacy enhanced web browsers and private browsing modes of common web browsers. 2018.

[4] M. R. Arshad, M. Hussain, H. Tahir, S. Qadir, F. I. Ahmed Memon, and Y. Javed. Forensic analysis of tor browser on windows 10 and android 10 operating systems. *IEEE Access*, 2021.

[5] G. King, J. Pan, and M. E. Roberts. Reverse-engineering censorship in China: Randomized experimentation and participant observation. *Science*, 2014.

[6] R. Ramesh, R. Raman, M. Bernhard, V. Ongkowijaya, L. Evdokimov, A. Edmundson, S. Sprecher, M. Ikram, and R. Ensafi. Decentralized control: a case study of Russia. In *Proceedings, 2020 Network and Distributed System Security Symposium*, 2020.

[7] T. K. Yadav, A. Sinha, D. Gosain, P. K. Sharma, and S. Chakravarty. Where The Light Gets In: Analyzing Web Censorship Mechanisms in India. In *Proceedings of the Internet Measurement Conference 2018*, 2018.

[8] H. Pang, K. . Tan, and X. Zhou. StegFS: a steganographic file system. In *Proceedings 19th International Conference on Data Engineering*, 2003.

[9] A. Barker, Y. Gupta, S. Au, E. Chou, E. L. Miller, and D. D. E. Long. Artifice: Data in Disguise. In *Proceeding of the Conference on Mass Storage Systems and Technologies (MSST '20)*, 2020.

[10] A. Zuck, U. Shriki, D. E. Porter, and D. Tsafrir. Preserving Hidden Data with an Ever-Changing Disk. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017.

[11] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu. Toward Robust Hidden Volumes Using Write-Only Oblivious RAM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[12] D. J. Solove. A Taxonomy of Privacy. In *University of Pennsylvania Law Review*, 2006.

[13] ENISA - European Union Agency For Network and Information Security. ENISA's PETs Maturity Assessment Repository - Populating the Platform. 2018.

[14] ENISA - European Union Agency For Network and Information Security. Online privacy tools for the general public - Towards a methodology for the evaluation of PETs for internet mobile users. 2015.

[15] ENISA - European Union Agency For Network and Information Security. PETs controls matrix - A systematic approach for assessing online and mobile privacy tools. 2016.

[16] D. Barradas, N. Santos, L. Rodrigues, and V. Nunes. Poking a Hole in the Wall: Efficient Censorship-Resistant Internet Communications by Parasitizing on WebRTC. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[17] A. H. e. a. Kwon. Riffle: An Efficient Communication System With Strong Anonymity. In *Proceedings on Privacy Enhancing Technologies*, 2016.

[18] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[19] K. Bock, G. Hughey, L.-H. Merino, T. Arya, D. Liscinsky, R. Pogosian, and D. Levin. Come as You Are: Helping Unmodified Clients Bypass Censorship with Server-Side Evasion. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2020.

[20] M. Muir, P. Leimich, and W. J. Buchanan. A Forensic Audit of the Tor Browser Bundle. In *Digital Investigation 29*, 2019.

[21] A. K. Jadoon, W. Iqbal, M. F. Amjad, H. Afzal, and Y. A. Bangash. Forensic Analysis of Tor Browser: A Case Study for Privacy and Anonymity on the Web. In *Forensic Science International*, 2019.

[22] R. A. Sandvik. Forensic Analysis of the Tor Browser Bundle on OS X, Linux, and Windows. 2013.

[23] L. Zeng, Y. Xiao, and H. Chen. Linux auditing: Overhead and adaptation. In *2015 IEEE International Conference on Communications (ICC)*, 2015.

[24] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, 2004.

[25] Z. Peterson and R. Burns. Ext3cow: A Time-Shifting File System for Regulatory Compliance. In *ACM Transactions on Storage*, 2005.

[26] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *ACM SIGOPS Oper. Syst. Rev.*, 2003.

[27] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and V. Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, 2007.

[28] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable Virtual Machines. 2010.

[29] B. Li, J. Li, T. Wo, C. Hu, and L. Zhong. A VMM-Based System Call Interposition Framework for Program Monitoring. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, 2010.

[30] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[31] H. Xiong, Z. Liu, W. Xu, and S. Jiao. Libvmi: A Library for Bridging the Semantic Gap between Guest OS and VMM. In *2012 IEEE 12th International Conference on Computer and Information Technology*, 2012.

[32] C. Wang, C. Wang, and S. Shieh. ProbeBuilder: Uncovering Opaque Kernel Data Structures for Automatic Probe Construction. In *IEEE Transactions on Dependable and Secure Computing*, 2016.

[33] Q. Feng, A. Prakash, H. Yin, and Z. Lin. MACE: High-Coverage and Robust Memory Analysis for Commodity Operating Systems. In *ACM, Proceedings of the 30th Annual Computer Security Applications Conference*. Association for Computing Machinery, 2014.

[34] F. Pagani and D. Balzarotti. Back to the Whiteboard: a Principled Approach for the Assessment and Design of Memory Forensic Techniques. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[35] S. L. Garfinkel. Automating Disk Forensic Processing with SleuthKit, XML and Python. In *2009 Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering*, 2009.

[36] N. A. Aziz, F. Mokhti, and M. N. M. Nozri. Mobile Device Forensics: Extracting and Analysing Data from an Android-Based Smartphone. In *2015 Fourth International Conference on Cyber Security, Cyber Warfare, and Digital Forensic (CyberSec)*, 2015.

[37] J.-N. Hilgert, M. Lambertz, and D. Plohmann. Extending The Sleuth Kit and Its Underlying Model for Pooled Storage File System Forensic Analysis. *Digit. Investig.*, 2017.

[38] S. L. Garfinkel. Digital Media Triage with Bulk Data Analysis and Bulk_extractor. *Comput. Secur.*, 2013.

[39] B. Nishida. Ubuntu Artifacts Generated by the Gnome Desktop Environment. *Technical report, SANS Institute*, 2020.

[40] C. K. J. P. Konstantinos Solomos, John Kristoff. Tales of Favicons and Caches: Persistent Tracking in Modern Browsers. *NDSS*, 2021.

[41] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[42] T. Bui, S. P. Rao, M. Antikainen, V. M. Bojan, and T. Aura. Man-in-the-Machine: Exploiting Ill-Secured Communication Inside the Computer. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[43] Y. Tang, P. Ames, S. Bhamidipati, N. Sarda, and R. Geambasu. CleanOS: Increasing Mobile Data Control with Cloud-based Eviction. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[44] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda. Privexec: Private execution as an operating system service. In *2013 IEEE Symposium on Security and Privacy*, 2013.

[45] J. B. Djoko, B. Jennings, and A. J. Lee. Tprivexec: Private execution in virtual memory. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016.

[46] L. Arkema and M. Sherr. Residue-free computing. *Proceedings on Privacy Enhancing Technologies*, 2021.

[47] A. Ross, N. Roger, and S. Adi. The Steganographic File System. In *Aucsmith D. (eds) Information Hiding*, 1998.

[48] X. Zhou, H. Pang, and K.-L. Tan. Hiding Data Accesses in Steganographic File System. In *Proceedings of the 20th International Conference on Data Engineering*, 2004.

[49] A. Chakraborti, C. Chen, and R. Sion. DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries, 2017.

[50] C. Chen, A. Chakraborti, and R. Sion. Pd-dm: An efficient locality-preserving block device mapper with plausible deniability. *Proceedings on Privacy Enhancing Technologies*, 2019.

[51] A. Barker, S. Sample, Y. Gupta, A. McTaggart, E. L. Miller, and D. D. E. Long. Artifice: A Deniable Steganographic File System. In *9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19)*, 2019.

[52] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[53] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. Lightweight Virtualization: A Performance Comparison. In *2015 IEEE International Conference on Cloud Engineering*, 2015.

[54] C. C. Spoiala, A. Calinciuc, C. O. Turcu, and C. Filote. Performance comparison of a WebRTC server on Docker versus virtual machine. In *2016 International Conference on Development and Application Systems (DAS)*, 2016.

[55] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014.

[56] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs Containerization to Support PaaS. In *2014 IEEE International Conference on Cloud Engineering*, 2014.

[57] NCCGroup. Understanding and Hardening Linux Containers. 2016.

[58] M. Souppaya, J. Morello, and K. Scarfon. Application Container Security Guide. In *NIST Special Publication 800-190*, 2017.

[59] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran. Understanding Security Implications of Using Containers in the Cloud. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[60] T. Bui. Analysis of Docker Security. *CoRR*, 2015.

[61] A. A. Mohallel, J. M. Bass, and A. Dehghantaha. Experimenting with docker: Linux container and base OS attack surfaces. In *2016 International Conference on Information Society (i-Society)*, 2016.

[62] Z. Jian and L. Chen. A Defense Method against Docker Escape Attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, 2017.

[63] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: general security support for the Linux kernel. In *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, 2003.

[64] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. *NAILabs Report 1*, 2001.

[65] M. Bauer. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux Journal*, 2006.

[66] M. Bélair, S. Laniepce, and J.-M. Menaud. Leveraging Kernel Security Mechanisms to Improve Container Security: A Survey. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, New York, NY, USA, 2019. Association for Computing Machinery.

[67] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger. Security Namespace: Making Linux Security Frameworks Available to Containers. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[68] J. Johansen. Making Linux Security Modules available to Containers: Stacking and Namespacing the LSM. In *Free and Open Source software Developers' European Meeting (FOSDEM '18)*, 2018.

[69] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[70] Z. Shen, Z. Sun, G. E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2019.

[71] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin. BASTION: A Security Enforcement Network Stack for Container Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[72] X. Liu, Q. Zhang, C. Tang, J. Zhao, and J. Liu. A Steganographic Algorithm for Hiding Data in PDF Files Based on Equivalent Transformation. In *2008 International Symposiums on Information Processing*, 2008.

[73] A. Shamir. How to Share a Secret. *Commun. ACM*, 1979.

[74] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 1948.

[75] P. Penrose, R. Macfarlane, and W. Buchanan. Approaches to the classification of high entropy file fragments. *Digital Investigation*, 2013.

[76] M. Halcrow. ecryptfs: An enterprise-class encrypted filesystem for linux. 2005.

[77] H. Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. 2010.

[78] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 1998.

[79] J. Chen, S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer. Machine learning for load balancing in the linux kernel. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2020.

[80] E. Oberstar. Fixed-point representation fractional math. 2007.

[81] J. Le Maire, N. Brunie, F. De Dinechin, and J.-M. Muller. Computing floating-point logarithms with fixed-point operations. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, 2016.

[82] P. Gill, M. Crete-Nishihata, J. Dalek, S. Goldberg, A. Senft, and G. Wiseman. Characterizing web censorship worldwide: Another look at the opennet initiative data. 2015.

[83] D. Fifield and S. Egelman. Fingerprinting web users through font metrics. 2015.

[84] Y. Teing, A. Dehghantanha, and K.-K. R. Choo. Cloudme forensics: A case of big data forensic investigation. *Concurrency and Computation: Practice and Experience*, 2017.

[85] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár. Microsoft coco: Common objects in context, 2015.

Appendix A

PET Utilization Scripts

0. Before executing Calypso and inserting the kernel module.
0.1 After executing Calypso and inserting the kernel module, without retrieving previous metadata.
0.2 After installing ext4 in Calypso's shadow partition.
0.3 After downloading Tor into Calypso's shadow partition.
1.1 Execute the Tor browser for the first time with the option "Always connect automatically" checked and "Use a bridge" checked, paired with "Select a built-in bridge" with "obfs4" selected, under "Tor Network Settings", for circumvention in censored countries, and finally connect.
1.2 Go to https://duckduckgo.com/ search engine and input "top onion websites" on the search bar.
1.3 Select the https://www.wizcase.com/blog/safely-find-the-best-dark-web-sites/ web page.
1.4 Open the HiddenWiki URL on a new tab.
2.1 On the HiddenWiki's tab, scroll down to the "Darknet versions of popular sites" and open "Mail2Tor" on a third tab.
2.2 Create a new account on the Mail2Tor's tab.
2.3 Login into Mail2Tor with the created account.
2.4 Send an email with a small attachment.
2.5 Refresh and open a received email with a small attachment and download it into Calypso's shadow partition. Close Mail2Tor's tab.
3.1 Return to HiddenWiki's tab and scroll down to the "Social Networks" section. Open Facebook's onion URL on another tab and login with an existing account.
3.2 Scroll through the feed and go to a friends profile and open the first profile picture.
3.3 Send a message to a friend and receive the response. Close Facebook's tab.
4.1 Return to HiddenWiki's tab and scroll down to the "Whistleblowing" section. Open ProPublica's onion URL on another tab.
4.2 Scroll through the main page and open an article. Close ProPublica's tab.
5.1 Return to HiddenWiki's tab and scroll down to the "Audio - Music / Streams" section. Open DeepWeb Radio's onion URL on another tab.
5.2 Click "Play" in the first stream and stay in the page for a while. Close DeepWeb Radio's tab and return to the HiddenWiki's tab.
6.1 Open a new tab and go to `https://www.youtube.com/watch?v=tpiyEe_CqB4&ab_channel=Rufus`
6.2 Click play and stay in page for a few seconds. Close the video's tab
7.1 Return to HiddenWiki's tab and scroll down to the "File Uploaders" section. Open Image Hosting's onion URL on a new tab
7.2 Upload image and wait for it to load
7.3 Then, download it with the onion URL provided, in a new tab, to Calypso's shadow partition. Close both tabs
8.1 Open a new tab and go to `https://www.mingpao.com/`, a censored website in China.
8.2 Scroll the web page until the end
8.3 Click on the last article available. Close both tabs
9.1 In the end, close the Tor browser
9.2 Unmount Calypso's shadow partition
9.3 Stop and remove Calypso's module, storing the metadata

Tor browser utilization script

0. Before executing Calypso and inserting the kernel module.
0.1 After executing Calypso and inserting the kernel module, without retrieving previous metadata.
0.2 After installing ext4 in Calypso's shadow partition.
0.3 After downloading and installing Signal into Calypso's shadow partition.
1.1 Start Signal desktop.
1.2 Link your phone to Signal Desktop via the QR code and your phone's app. Then, click "Finish linking phone".
1.3 Click the compose button and create a group message with messages disappearing after 1 hour.
2.1 Send a text message to the group.
2.2 Receive the text response
3.1 Send and audio
3.2 Receive an audio and listen to it.
4.1 Send JPEG picture as attachment.
4.2 Receive JPEG picture as attachment and save it to Calypso's shadow partition.
5.1 Make a video call and share the screen
6.1 Receive link and open it
7.1 In the end, close the Signal desktop app
7.2 Unmount Calypso's shadow partition
7.3 Stop and remove Calypso's module, storing the metadata

Signal desktop utilization script

0. Before executing Calypso and inserting the kernel module.
0.1 After executing Calypso and inserting the kernel module, without retrieving previous metadata.
0.2 After installing ext4 in Calypso's shadow partition.
0.3 After downloading Megasync into Calypso's shadow partition.
1.1 Start Megasync desktop and link empty account with device, which will cause syncing.
1.2 List files in the synced Megasync folder
2.1 Create new directory in the Megasync folder
2.2 Add a file to the new directory
3.1 Add a new directory with some files to the Megasync account from another machine
3.2 List files in the synced Megasync folder
4.1 In the Dropbox desktop app, go to "Preferences", then switch to the "Sync" tab, click "Selective syncing", and exclude the second folder from automatic syncing
4.2 List files in the Dropbox folder
4.3 From another machine, add another file to the second folder
4.4 List files in the Dropbox folder
5.1 In the end, stop the Dropbox service
5.2 Unmount Calypso's shadow partition
5.3 Stop and remove Calypso's module, storing the metadata

Megasync desktop utilization script