Traceless Execution Support for Privacy Enhancing Technologies

(extended abstract of the MSc dissertation)

Daniela Gorjão Lopes
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Nuno Santos

***Abstract*—This work studies how existing privacy enhancing technologies (PETs) can be contained to withstand attacks where the adversary has full access to a device's persistent state. For this, we aim to create a steganographic file system that provides the stored data with plausible deniability, a property that allows users to deny the existence of hidden data in their device. This way, users living under repressive regimes can use PETs without fear of reprisals. We implemented a prototype of Calypso, a system that provides support for the installation and execution of PETs on a "shadow partition", 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, creating an opportunity for the usage of forensic deniable footprints to justify changes in the disk.**

I. Introduction

Nowadays, Internet users face an increasing deterioration of the privacy and freedom of speech. This situation is worsened by the rise of censorship and repressive regimes. 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.

Despite the wide availability of PETs, existing software still imposes several drawbacks on its users. Many tools tend to: 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.

Several existing approaches allow users to evade this kind of forensic analysis, e.g., steganographic file systems [2], [3]. 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. 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 [4], [3], but require additional hardware for file system isolation and security.

This paper presents 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. To achieve this, it explores 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. This produces 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.

II. Related Work

Numerous anti-forensic solutions have been proposed to mitigate the digital traces left by software execution. Some systems focus on enhancing the privacy of program execution by isolating processes and concealing their data from the rest of the system. Lacuna [5] 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. CleanOS [6] 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.

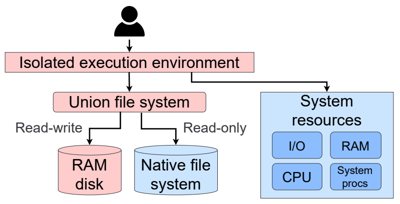

Figure 1: Residue-free computing architecture.

PrivExe [7] creates 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 by extending the Linux kernel.

Residue-free computing [8] aims to isolate all interactions made by a program to memory only using containers and a union file system that redirects write accesses to a RAM disk, as shown in Figure 1, preventing them from making persistent changes to the disk. This system has several shortcomings, such as not supporting persistence nor plausible deniability, since there is no attempt to hide the existence of this program and the other programs to be executed with its support. The isolation performed with containers depends on the characteristics of the application so that the right native resources can be allowed, causing a lack of support to some. Additionally, accessing the native resources causes changes at the file system level.

None of the previous systems provides forensic deniability, due to the lack of attempt to conceal the system itself. Deniable file systems, on the other hand, 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 [9] 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 if such a file is present in the system. StegFS [2] stores blocks of the hidden file system within the unallocated space of the native file system, using steganography. However, it 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.

A more advanced 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 is able to observe modifications that can indicate the existence of hidden data. The same authors from StegFS [10] proposed using oblivious RAM (ORAM) techniques to conceal data accesses but does not hide its presence. HIVE [11] and Datalair [12] propose a write-only oblivious RAM, but these techniques incur in significant performance penalties, besides random disk write-patterns and slower performance might indicate suspicious activity.

PM-DM [13] 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 of 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 [4] was proposed as a 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.

Artifice [14] aims to prevent multi-snapshot attacks, by disguising data accesses to disk with deniable operations such as redundant writes. 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, along with *checksums* and replication of the main data structure. However, 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. 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, and booting from another OS enlarges the disk footprint by forcing a reboot. Artifice offers no protection in case of swapping and computer hibernation which can cause the information to be written down to the disk. Artifice is described in more detail in a follow-up paper [3], 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. 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.

III. Threat Model

The goal of the adversary is to detect that Calypso is installed on the system, or that a PET was executed with Calypso's support. 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 on the machine or about anything executed or stored in Calypso's shadow partition, from inspecting the machine. 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 a journalist reporting humanitarian causes to a foreign country.

Calypso aims to provide **plausible deniability**, which 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. We aim to offer plausible deniability by providing the following two sub-properties: i) **isolation**, that guarantees no information leaks from the Calypso processes into the native file system; and ii) **non-observability**, which states that an adversary should not be able to detect hidden data on the native free blocks.

IV. Design

Calypso aims to provide 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 can be allocated at any time.

Calypso's architecture is described in Figure 2, 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 that exports a shadow partition whose size is determined by the user when installing Calypso. This partition can be formatted and mounted on a given folder. The interaction with the user is done through a command-line application (CLI) that allows executing the bootstrap system that inserts or removes Calypso's module in the system.

The heart of the system is the Calypso driver. It is composed of two main parts, as shown in Figure 3: i) a block mapper and a block encoder, that together handle the requests to the shadow partition, hiding Calypso's data on the unused 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

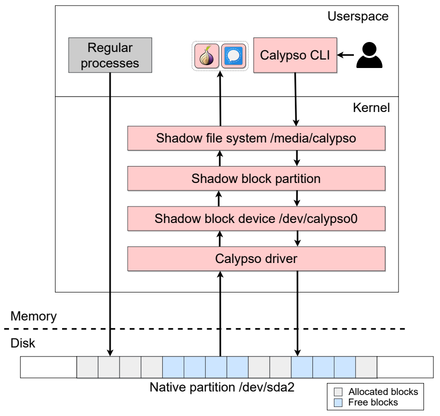

Figure 2: Calypso architecture.

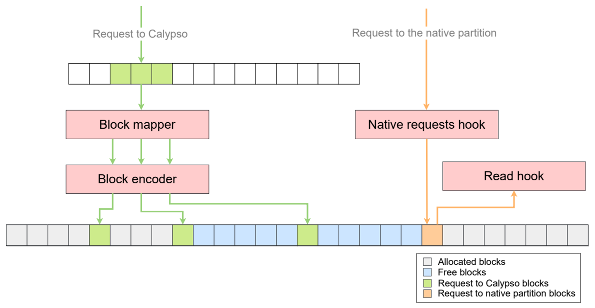

Figure 3: Calypso driver internals.

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.

A. *Traceless Bootstrap*

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. 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 [15] or splitting it among several blocks using Shamir Secret Sharing [16].

B. *Block Allocation*

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 does not data loss. 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.

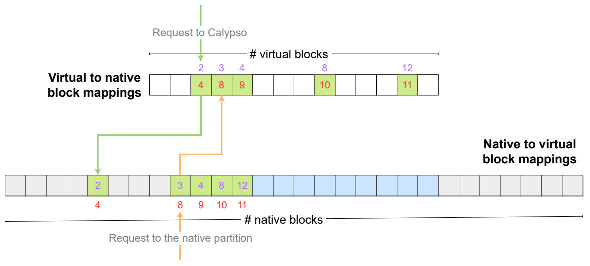
Request to Calypso
# virtual blocks
Virtual to native block mappings
2 3 4 8 12
4 8 9 10 11
Native to virtual block mappings
2 3 4 8 12
4 8 9 10 11
# native blocks
Request to the native partition

Figure 4: Calypso block mappings.

Block mappings: To keep track of the blocks that compose the shadow partition, 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. We use the first structure when we are remapping Calypso's requests to the native block device, and we use the structure that maps native to virtual blocks to have direct access times when we intercept requests from the native file system, as shown in Figure 4.

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 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 prevents assigning the same native block to multiple virtual blocks. The used blocks bitmap allows keeping track of which native blocks are allocated or free. It is a contiguous array containing 1 bit for each native block. If the bit equals 0, then that block is free, otherwise, it is allocated.

C. *Monitoring Native Changes to Blocks*

Firstly, Calypso needs to update the state of the used blocks bitmap as the native system allocates new blocks, which need to become ineligible for Calypso to allocate. Furthermore, using the free blocks of the native file system is unstable 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 irreparable damage to 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.

Copying blocks to avoid overwrites while Calypso is loaded: When Calypso intercepts a request to a block mapped by itself, this means a mapped block is about to be overwritten. To avoid losing data, Calypso should freeze 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 Figure 3, 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: Calypso persists the used blocks bitmap at the end of each session so that when Calypso is reloaded, it has the necessary data 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 is mapped by Calypso, then its contents may have changed, and we should resort to existing data recovery techniques based on redundant error-correcting codes.

D. *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. To prevent 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.

E. *Deniably Encoding Data Blocks using Entropy*

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 an established entropy threshold to store the encrypted data of Calypso.

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 [17], [18] for each block, described in Equation 1.

$$H(X) = -\sum_{i=1}^{n} P(x_i) log P(x_i), 0 \leq H(X) \leq 8 \tag{1}$$

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 2. These values are normalized to be between 0 and 1.

$$\triangle H(X) = \frac{|H(x_1) - H(x_0)|}{max(H(X))}, 0 \leq \triangle H(X) \leq 1 \tag{2}$$

Using entropy as a deciding factor to change the contents of free blocks allows Calypso to encode data in the free blocks while maintaining their characteristics.

Optimal entropy threshold: The *entropy threshold* is a value that allows the user to customize the selection of which blocks Calypso uses to hide its data. The optimal entropy threshold value depends 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. 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, who only needs to use a whistleblower submission system to share its reports, then the highest entropy threshold is advisable.

Multi-Snapshot Attacks and Plausible Deniable Footprints: Resistance to multi-snapshot attacks entails that the adversary cannot observe changes between multiple snapshots of the system, 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 that no changes were made to those blocks, but we can claim these changes were originated from the system's regular utilization, such as temporary 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*.

V. Implementation

Calypso's prototype was built as a kernel module, targeted for Linux platforms running the kernel version 5.4.

A. 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 currently in use by the native file system. This can be technically challenging since the Linux Virtual File System (VFS) does not have functions to retrieve the free blocks, so we need to use code specific to the Ext4 file system. We start by obtaining 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.

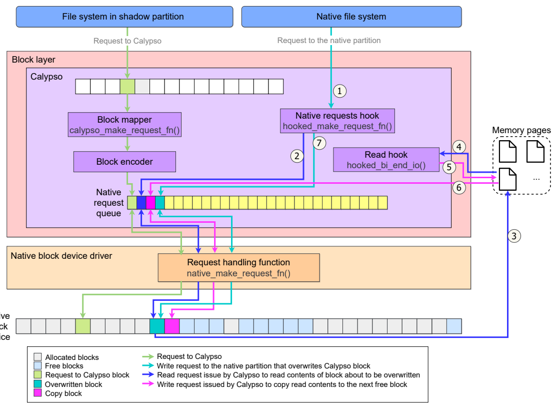
File system in shadow partition
Native file system
Request to Calypso
Request to the native partition
Block layer
Calypso
Block mapper
calypso_make_request_fn()
Block encoder
Native requests hook
hooked_make_request_fn()
Read hook
hooked_bi_end_io()
Memory pages
Native request queue
Native block device driver
Request handling function
native_make_request_fn()
Native block device
1
2
3
4
5
6
7
Allocated blocks
Free blocks
Request to Calypso block
Overwritten block
Copy block
Request to Calypso
Write request to the native partition that overwrites Calypso block
Read request issue by Calypso to read contents of block about to be overwritten
Write request issued by Calypso to copy read contents to the next free block

Figure 5: Calypso request redirection and interception to handle overwrites. The green arrow shows how a request to the shadow partition is handled by *redirecting* it to the native request queue. Once a request is received by Calypso, the virtual block associated with that request is mapped to the corresponding native block by the block mapper, using the Calypso mapping structures. Then, the request data is encoded and the request is placed in the native request queue. Finally, the request is fetched by the native request handling function and gets redirected to the native block in the block device. The remaining arrows show how Calypso performs *interception* and handles 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).

B. Redirecting and Intercepting Block Requests

Calypso needs to issue the requests received to the native block device. For this, it redirects the requests, as detailed in Figure 5 by the green arrow. Calypso should act as the front end, 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()` so that requests are passed directly to Calypso's driver instead of going to a request queue. To redirect Calypso's requests to the native device, we need to open it without exclusive mode, obtaining an instance of `struct block_device`. Then, for every request, we need to change the the target device sector and the target device fields, and finally send the request to the native device's request queue using the `generic_make_request()` function.

To monitor and prevent changes to blocks mapped by Calypso, our module needs to add functionality to the native request handling function by hooking it. We do this by overriding the pointer to this function, `make_request_fn`, a field belonging to the instance of `struct request_queue`. This instance is obtained from the native block device instance previously acquired. When doing this, we need to be careful since any disruption that prevents native requests from being correctly processed may invalidate the entire system. So, we need to store the pointer to the original `make_request_fn` function so that we can call and restore it when Calypso is unloaded. The substitute function should ensure that the original function gets called at some point.

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. Calypso intercepts all requests to the native block device, so it detects when its data is about to be overwritten. This is the case when a write request issued by the native file system targets a block mapped by Calypso. So we need to freeze that request while Calypso's data is being copied as shown in Figure 5 by the blue, pink and cyan arrows.

C. 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 when Calypso is loaded, without having the mappings to the meta-data blocks in memory. To enable these cryptographic primitives, the user needs to insert a password of its choosing each time the Calypso's kernel module 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.

Hiding data blocks: Calypso needs to hide its data, present in the native partition, from potential adversaries using encryption. This is done whenever a request is remapped to the native device: i) in case of a write request, it replaces the data to be written with its encrypted version; and ii) in case of a read request, it replaces the read encrypted data with its decrypted contents. This way, the native device will only have access to the encrypted data. This requires generating always the same key at run time using the *HMAC-based extract-and-expand key derivation function*, contributing for Calypso changes to remain non-observable.

Storing and retrieving meta-data: Calypso stores meta-data on its own mapped blocks. When Calypso gets unloaded, those memory structures disappear. We need to provide a way for Calypso to determine the first block of meta-data every time it is reloaded, without the support of in-memory structures. This way, it can reconstruct them, allowing the mapping of blocks. Passing a seed to a pseudo-random number generator allows obtaining always the same sequence of pseudo-random numbers. We exploit this concept to generate the numbers for the first block from which we will attempt to store and retrieve meta-data.

VI. Evaluation

To evaluate our system, we focus on the following goals: i) test whether Calypso can sustain the execution of PET applications without file system errors (Section VI-A); ii) show if Calypso reduces the traces left by PETs in the native file system (Section VI-B); iii) assess the capacity of Calypso to make only deniable changes to the native file system (Section VI-C); and iv) test the performance degradation caused by using Calypso, as well as the performance of file system accesses to the shadow partition (Section VI-D).

A. Functionality

We monitored the execution of PET applications representative of multiple disk workloads: i) Tor browser; ii) Signal desktop; and iii) Megasync desktop. Then, we assessed whether Calypso can provide the storage resources to accommodate their installation and execution.

We used a native partition 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. We consider a block as being equivalent to 4096 bytes, which is also the default block size in the Ext4 file system. Then, Calypso is initialized with 300,000 blocks, approximately equivalent to 1.1 gigabytes.

Respectively, each of the following plots shown in Figures 6 and 7 reflect how the block utilization evolves with multiple different actions being executed with the Tor browser and Signal. Each number of blocks represented in the y axis should be interpreted as 1k= 1×10^3 blocks. Annotation labels highlight interesting steps of the PET utilization. Each one contains the action performed in that step and the cumulative number of allocated blocks.

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: i) Calypso's shadow partition was formatted with the ext4 file system; ii) the installation of each tool and associated files on the shadow partition; and iii) each tool, when first executed, stores initial meta-data for usability purposes, such as user accounts or preferences.

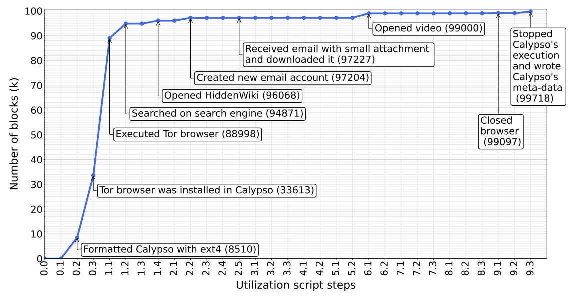

Figure 6: Block utilization evolution with time for Tor browser executed with Calypso

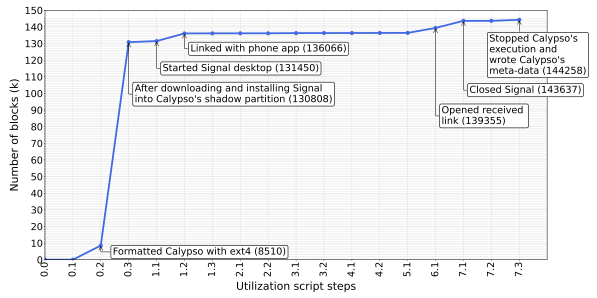

Figure 7: Block utilization evolution with time for Signal desktop executed with Calypso

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 when Calypso is closed since it persists the meta-data.

In more detail, Figure 6 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 big amount of files occupied by the Tor browser folder. When first executed, the browser needs to get server descriptors with information on Tor relays, 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 **Signal**'s block evolution in Figure 7, 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 as contacts, user configurations and previous conversations, heading towards a smaller steady increase in most successive

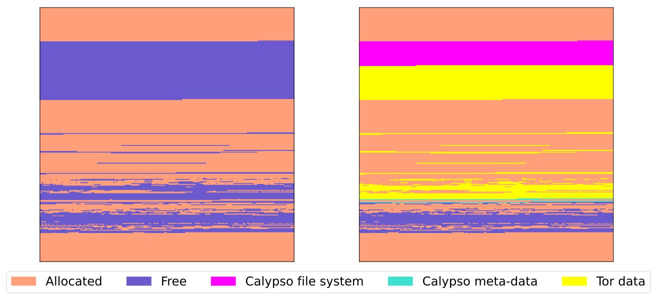

Figure 8: Comparison of a part of the native partition state, before and after executing Megasync with Calypso, from block 200 to block 90000, as a matrix of 300x300

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.

Calypso allocates blocks using only the native free blocks: Figure 8 pictures the initial segment of the native partition state before executing Calypso on the left, and those same blocks after the utilization of Megasync with Calypso, 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 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 and they overlap exclusively the native free blocks in purple, ensuring that only the free blocks of the native file system are used.

B. Isolation

We aim to assess whether persistent observable traces are leaking from Calypso's execution environment to the native file system. For this, we consider the following metrics: 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.

Tracing accesses to the native file system: To trace all the file system interactions, we used the Linux Auditing System (auditd), 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

File category	Initial state		Tor native		Tor Calypso	
	Access count	Diff file count	Access count	Diff file count	Access count	Diff 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 I: 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

summary reports on the accessed files, obtaining all the accessed paths during a 10 minute time period with the respective number of accesses to each path.

A summary of our results is depicted in Table I. Each line refers to a different file category. The accesses count refer 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 refers to the amount 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 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 I, most of the file system accesses performed by the Tor browser are diverted

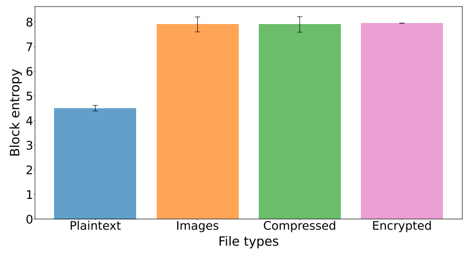

Figure 9: Average block entropy and standard deviation in the blocks of 1GB virtual disks simulating unused space, for multiple file types

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 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.

C. *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. For this, 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 Calypso on the system. The entropy differential is computed by taking two entropy values for each block in two different moments in time, before and 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 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.

Most blocks that constitute highly used media files have

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 II: Differential entropy statistics

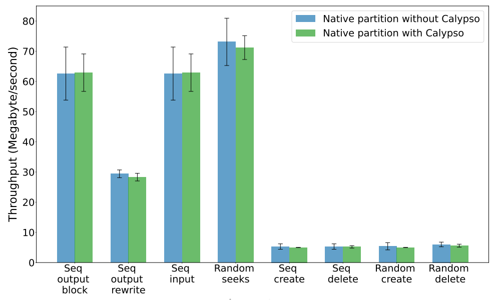

Figure 10: Comparison of the throughput of the native file system without and with Calypso loaded

an entropy close to the entropy of encrypted blocks: In Figure 9, 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 are 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 computers with higher usage.

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. Each line in Table II shows 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). The average differential shows us the overall difference in entropy after the blocks above a certain threshold have been replaced with encrypted data. The minimum and maximum differential 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 4 GB of free space. 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 9. 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 reduces the storage capacity.

D. Performance

We want to attest that: 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. This can be achieved by comparing the latency and throughput of file system operations 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 mounted 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 mounted 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.

We used bonnie++, a file system bench-marking tool, 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. We specified 128 files for the file creation tests. Each test was executed 11 times (excluding the first execution results) and we obtained their average, along with the standard deviation as the possible error.

The native throughput values are similar with and without Calypso: The throughput for the first scenario

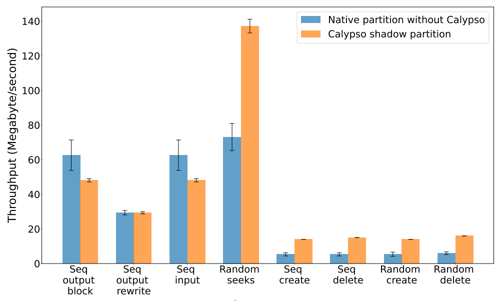

Figure 11: Comparison of the throughput of the native file system without Calypso loaded and the file system in Calypso's shadow partition

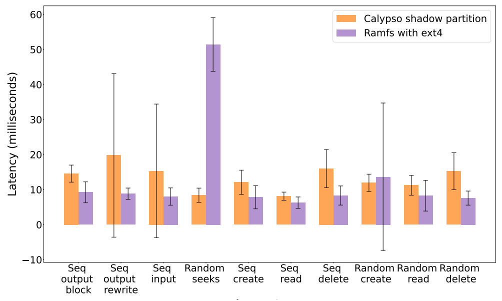

Figure 12: Comparison of the latency of the file system in Calypso's shadow partition and a ramfs without Calypso

represented in Figure 10 shows very similar values for both the native partition without Calypso and with Calypso.

Calypso presents similar or better throughput values than the native file system without Calypso: Figure 11 demonstrates that Calypso's shadow partition shows a slight improvement in the throughput when faced against the native partition without Calypso being loaded, meaning more megabytes can be processed per second.

Calypso reveals to be comparable to a ramfs: Figure 12 shows that Calypso has a higher latency overall, translating into a comparable but slightly lower performance. In Figure 13 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: Figure 10 demonstrates 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.

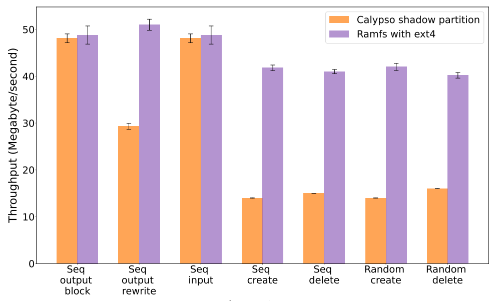

Figure 13: Comparison of the throughput of the file system in Calypso's shadow partition and a ramfs without Calypso

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 11. 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 12 and Figure 13. 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.

VII. Conclusions

This work describes 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. The experimental evaluation performed on the 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 is still room for improvement in the isolation of executed programs.

Acknowledgments

This work benefited from the helpful discussions with Diogo Barradas regarding its design and evaluation.

References

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] A. Shamir, “How to Share a Secret,” *Commun. ACM*, 1979.

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

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