RecoCoDe: Recover From Data Corruption in Deniable Storage

Vero Estrada-Galiñanes
EPFL
Lausanne, VD, Switzerland
veg@ieee.org

Andrej Milicevic
EPFL
Lausanne, VD, Switzerland
andrej.milicevic@epfl.ch

ABSTRACT

Some deniable storage tools use storage overcommitment as part of their methods to hide encrypted data. This technique may help to successfully conceal the existence of hidden data, however it introduces the risk of hidden data overwriting. This serious limitation of plausibly deniable tools is largely unexplored. Consider, for example, that an investigative journalist might have being exposed to serious life-threatening risks for gathering and safeguarding data with a system vulnerable to data corruption. For the journalist, ignoring data corruption is not an option.

Our ongoing research proposes RecoCoDe to store data redundantly in hidden volumes to cope with data corruption arising from arbitrary situations causing data overwriting. RecoCoDe can recover from data corruption caused from undesirable overwrites to hidden data by using some kind of error correcting codes. It achieves that with another layer of indirection that entangles content together in a way that creates a more resilient hidden volume without requiring any extra backup device and with insignificant impact on the writing performance.

KEYWORDS

deniable storage, coercion resistance, data corruption, redundancy, AE codes, device mapper target

1 INTRODUCTION

The need for concealing data is driven by various factors and has a complex interplay between privacy, security, and ethical concerns. Individuals have a right to privacy. Thus, concealment tools can help to protect sensitive information and safeguarding against breaches that could lead to personal harm. They can also contribute for compliance to privacy regulations, such as GDPR (Europe), CCPA (California, U.S.), etc. Governments and military organizations require these tools to protect information that if leaked would threaten national security. Businesses need to protect intellectual property, trade secrets, proprietary information from competitors, and data from their customers. Concealment tools can protect individuals located in places where certain aspects of their identity or beliefs may put them at risk of harm or discrimination. When freedom of speech is not guaranteed, tools to conceal data can protect dissidents, activists, and journalists from persecution. Furthermore, concealment tools allow for the safe communication and dissemination of information in oppressive regimes.

Free and Open Communications on the Internet 2024 (2), 54–63

Concealment tools include a wide range of solutions to hide information such as encryption, steganography, and various forms of data obfuscation. The primary purpose of these tools is to prevent unauthorized access. They work by making information unreadable or invisible without the proper keys or capabilities. Taking a step further, plausibly deniable (PD) storage offers the user the possibility to deny convincingly that certain information is stored on a storage device [5]. The first known PD tool was the steganographic file system proposed by Anderson in 1998 [1]. Since then, researchers have proposed hidden volumes with diverse features to conceal information with plausible deniability.

The rationale behind PD tools is that the adversary does not have enough reasons to justify the effort of searching for data that they cannot prove its existence. The adversary may persuade or coerce the user to provide information. But, if the user is able to resist that coercion and the tool successfully conceals the information, it is reasonable to think that at some point, the adversary will give up, letting the person go with the device. In the next subsections, we provide two examples to illustrate the importance of this topic.

1.1 Motivation 1: Cross-border Journalism

Plausibly deniable storage tools are potentially used by journalists and humanitarian workers in possession of sensitive information. A motivation factor for using these tools is to protect data that can imperil the identity of people under potentially life-threatening situations. Global journalism has increase the number of cases in which journalists need to transport material from one country to another for journalist purposes.

A routine border search at the airport occurs at the discretion of border agents when there is no evidence to support a reasonable suspicion to seize any equipment. According to the US Customs and Border Protection Directive 3340-049A [12], Section 5.1.2, "a border search will include an examination of only the information that is resident upon the device and accessible through the device's operating system or through other software, tools, or applications." The person authorized to conducted the search, from now on the officer, "should also take care to ensure, throughout the course of a border search, that they do not take actions that would make any changes to the contents of the device." If the authorities stop a user for a laptop's inspection and request the device's password, as long as the user complies with the request and the agents do not find any reason to suspect, it is possibly that the user will be allowed to leave with its laptop. We assume that other countries have similar procedures. Despite those procedures, it is reasonable to consider that in certain borders computer equipment may be handled improperly during routine border examinations.

We want to help address the following concern: *Will the data be intact after the agent inspection?*

1.2 Motivation 2: Coercive Control and Domestic Abuse

Plausibly deniable storage tools are potentially used by victims of domestic abuse to avoid the disclosure of sensitive information to their coercers.

Victims of domestic abuse may need to conceal information from their coercers. Digital environments provide an alternative for safeguarding information without the need of a concealed physical space. We assume that the victim possess computer devices such as a laptop or a mobile phone. While these are every day devices that may not raise the attention to the coercer, the possession of encrypted information may put the victim in trouble. In particular, the coercer may force the victim to disclose the password. The victim may choose to store documents remotely in the cloud. Although this alternative can help in some cases, the victim may need to keep critical information locally, such as their credentials to access government services. For example, one mitigation for coercion in the context of e-voting systems is to give voters fake voting credentials that they can surrender to a coercer as if they were the real credentials [11]. Fake credentials look identical to real ones, but cast votes that are silently omitted from the final tally.

Voters need to register in-person to obtain the credentials, which are meant to be used during a few years. Basically, the registration booth prints QR code credentials in paper format that later are read, stored or activated using a trusted computer device. This solution has been studied in the literature but is yet to be seen in the real-world. One question that remains unanswered is how a victim of domestic coercion can store the credentials safely. A simple solution is suggesting that the victim entrusts their credential to a trusted person. However, we are interested in providing a self-sovereign solution that allows the victim to conceal information without depending of third parties.

We may rely on plausibly deniable tools to solve this problem. A caveat is that the coercer may request or force the victim to show what the contents of their computer devices. Although plausibly deniable tools may do their job of concealing information, it is possibly that hidden data may be corrupted during such examination. Again, the question that is relevant to answer is: *Will the data be intact after the coercer inspection?*

1.3 The Data Corruption Problem

Researchers have proposed diverse solutions to the problem of concealing information. Generally speaking, the ultimate goal is to protect data so only the authorized eyes can see the information while the adversary is unaware of the existence of such protected data. Under standard definitions for plausible deniability [5], the tool does not need to safeguard data integrity. In fact, a basic device inspection that includes write operations on the standard volume may cause *hidden data overwrites* without the adversary awareness. If an inspection corrupts data, no error messages can be shown otherwise it will break the purpose of the protection. Unfortunately, this is bad news for the user. It's likely that some information could be permanently lost when handling a volume that contains hidden encrypted data in its free space.

We observe that the possibility of irrecoverable data does not seem a cause of concern for researchers. From informal discussions with PD tool developers, we learned that data corruption, resulting from a device's security being compromised, is often seen as a beneficial side effect because it lowers the risk of information leaks. This approach is, however, not sufficient to protect information that may not have any other backup. In some scenarios, the data holder may be exposed to serious life-threatening risks while the tools cannot provide sufficient protection against data corruption.

We think that PD tools should provide redundancy to mitigate data corruption arising from hidden data overwrites.

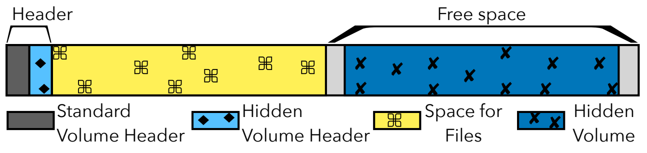

Figure 1: TrueCrypt volume layout with a hidden volume inside the free space.

1.4 Contributions

In this paper, we propose to rethink plausible deniable storage tools to consider the problem of data corruption. We propose *RecoCoDe* a device mapper target for the Linux kernel that uses entanglement codes to store data redundantly on hidden encrypted volumes. *RecoCoDe* may be integrated with PD tools like Shufflecake, a next generation TrueCrypt-like solution. We also provide an insightful discussion about adding redundancy to PD tools.

2 PLAUSIBLY DENIABLE STORAGE

Various PD schemes exist. While this paper focuses on device-oriented PD schemes relying on storage layers, trace-oriented PD schemes focus on the traces that read/write operations leave on the storage medium.

2.1 TrueCrypt and VeraCrypt

TrueCrypt [14] is a discontinued software that offers on-the-fly encryption (OTFE). It allows users to create a virtual encrypted disk within a file or encrypt a physical disk or partition, i.e., a hidden volume inside the free space of a TrueCrypt volume as seen in Figure 1. Full disk encryption such as Linux Unified Key Setup (dm-crypt/LUKS) [9] use plain disk headers, whereas in PD schemes the disk headers and content are indistinguishable from random.

Truecrypt is limited and does not offer data integrity protection. The system cannot protect the content that is hidden in the free space while an adversary operates or observes the operation of a device in which only the outer volume is mounted and the hidden volume remains hidden (motivation 1 and 2). Writing operations on the outer volume may request the allocation of free space containing hidden data without raising errors.

VeraCrypt is a fork of TrueCrypt. As such, it supports plausibly deniability and it is OFTE based. As its predecessor, it allows the creation of a single hidden volume within another volume. A relative recent security evaluation [10] conducted by the Fraunhofer Institute for Secure Information Technology (SIT) on behalf of the Federal Office for Information Security (BSI) determined that VeraCrypt seem to protect data effectively in case of theft or loss of encrypted devices since it protects data confidentiality as long as the hidden volume is not mounted. However is not recommended for any form of online attacks on a running system. Another limitation, relevant to our work, VeraCrypt does not protect data integrity.

2.2 Shufflecake

Shufflecake [3] is a tool used to create multiple hidden volumes on a storage device. It is designed as a block indirection layer on top of an encryption layer and implemented as a deviced mapper target for the Linux kernel. The current version has been tested with Linux kernel versions 6.1, 6.2, and 6.3. It is a successor to tools such as TrueCrypt and VeraCrypt, and provides two key properties that those tools do not: Shufflecake can create multiple hidden volumes, not just one, and it is independent of the filesystem of the underlying storage device. Shufflecake can handle up to a maximum of 15 hidden volumes on one storage device.

Shufflecake operates with a single underlying physical disk or device. The device is formatted to host one or multiple volumes (logical storage units, usually represented as virtual block devices), each encrypted with its own symmetric key. The user can yield to the examiner the password to some of these volumes (*decoy volumes*). The decoy volumes might only contain deceptively innocuous content. The crucial part of the design is that it should be impossible for an adversary to tell whether all passwords have been revealed, or if there is still undisclosed secret information remaining. Volumes are only opened when their corresponding password is provided. The mountpoint is hidden within the context of the volume itself. In Shufflecake, it is up to the user to mount the devices. But other implementations may mount devices automatically once the volume is opened.

Shufflecake addresses the storage space at the slice granularity instead of block granularity. The slice mechanism reserves a whole slice of $S_L = 256$ blocks every time a volume requests a block. The slice mechanism works in harmony with some file systems to avoid fragmentation in the long run. For instance, *ext4* tries to keep related files using the concept of block groups (32768 consecutive blocks).

3 DATA CORRUPTION PROBLEM

In this section we focus on data corruption arising from hidden data overwrites and the burden of responsibility, which is also relevant to the problem.

3.1 Space Overcommitment

Overcommitment is a common practice in virtualization systems. It refers to the technique of allocating more virtualized resources, such as CPU, memory, or disk space, than the physical hardware actually possesses. In its essence, Shufflecake is a disk virtualization system, employing space overcommitment to facilitate Plausibly Deniable (PD) storage. This is what allows the total size of the logical volumes to surpass the physical storage space limit, as long as the actual utilized space does not exceed the total available space.

Overcommitment is possible thanks to the lazy allocation technique that delays the allocation of resources, in this case space, until they are actually needed.

3.2 Hidden Data Overwrites

Hidden data overwrites may occur when not all volumes are opened. Shufflecake has no way of detecting the existence of a volume unless it receives the password for that volume. Given the overcommitment of the physical storage space, there is a risk that an open volume re-uses physical slices that were previously assigned to a hidden volume. In other words, an allocation error occur when a physical slice is incorrectly assigned to more than one volume. Since the volume is not opened, the error is produced in silence and eventually triggers a hidden data overwrite. We discuss in particular hidden data overwrites in Shufflecake.

In Shufflecake, the standard volume must contain a “decoy” volume to convince an examiner of the existence of a single disk volume. We assume that the victim has to surrender the password of the standard volume to the examiner. As a quick recap, in the context of our two motivation examples, the examiner represents the border agent or the domestic abuser. During an investigation, the examiner has access to the victim’s computer device to conduct any kind of arbitrary behavior. If the examiner writes content on the standard volume, it can corrupt data from the hidden volume.

Interestingly, developers admit this problem is “inevitable, and can only be mitigated by frequent backups of the disks” [2]. But victims of coercion may not have opportunity to create backups. In our examples, the users of PD tools may not have opportunity to create external backups. In addition, shifting the burden of responsibility onto users can be problematic.

We quantify the hidden data overwrite problem in the evaluation section. Overall, we notice that the inadvertent overwriting of hidden data is a significant problem. If the user does not open all volumes, as suggested by the Shufflecake’s authors, the risk of corrupting any important files stored in hidden volumes is real. The more storage space is used, the higher the chance of data corruption because the ratio between unused free space and free space containing hidden data diminishes.

3.3 Burden of Responsibility

Within the domain of operational security, using a device that has been compromised or potentially tampered with poses significant risks. Once a device is out of one’s possession, its security integrity is uncertain. It may have been merely inspected or, worse, infected with spyware. While security experts might be able to assess and mitigate these risks, a non-technical user likely lacks the skills to do so effectively. Therefore, the safest course of action is to discard the compromised device and restore data from a backup whenever possible.

Encouraging or enabling the continued use of a compromised device by adding features for post-compromise repair can be counterintuitive and hazardous. Such features might inadvertently shift the burden of security onto users, expecting them to discern and manage risks that are best handled by the developers of security tools.

Nonetheless, we urge to reconsider this stance as it is not universally applicable. In scenarios where backups are impractical or impossible, such as when data must pass through physical security checkpoints without duplication to avoid raising suspicion, the approach must be different. Here, implementing consistency checks and incorporating redundancies can enhance data resilience. This strategy, although it may introduce a higher degree of operational security risk, ensures that critical information remains intact and accessible when there are no alternatives.

3.4 Risk of Data Corruption

During coercion, users are forced to surrender their computer device and at least one password. The valuable data is concealed in a Shufflecake device, not opened during examination. If the examiner handles the computer device in a way that causes writes to the disk there is a risk that blocks that contain hidden data are assigned to the open device. The probability that a hidden data overwrite occurs in the next write is lineal with the space used by the hidden volumes. Considering the hidden volume used space is (H), and that the total free space in the decoy volume is (F), The probability P that a block written to the decoy volume corrupts data in the hidden volume can be expressed by the ratio $\frac{H}{F}$.

3.5 Solution Proposal

We propose to mitigate the risk of corruption when not opening all volumes by using some form of error correction on the unopened volumes. Once all the volumes are mounted again, the solution can try to recover the missing data using the redundancy available in the system. As with any solution that adds redundancy the price that we need to pay is a reduction on the effective storage capacity.

4 THE RECOCODE ENTANGLED VOLUME

In this paper, we study how to store data redundantly on a hidden volume without using any external device (as in local or off-site backups) or cloud-based backup. We propose *RecoCoDe*, a layer that sits between the user and the plausibly deniable tool to insert redundant information that can be used to recover data in case of hidden data overwrites.

4.1 RecoCoDe Layout

Redundant arrays of interdependent disks (RAID) [13] comprise different types of disk array architectures using striping, mirroring, or parities to increase performance and/or protecting data stored against disk failures. RAID-like techniques [16] are used at different storage layers and even in network file systems to protect data against different type of failures. The most simple well-known layout to create redundancy in storage devices is disk mirroring, also referred as RAID 1. Disk mirroring is a data storage configuration that copies (or "mirrors") data simultaneously onto two or more hard drives.

RAID 1 provides data duplication across two drives in an array. Notably, in our solution we use a single drive or device. This decision compromises one layer of domain failures since a failure of the physical drive is not repairable. But our aim is to mitigate the hidden data overwrite problem. In our scenario, the users have a particular priority, they have to avoid arousing suspicion by carrying or holding multiple devices. In case, this requirement does not hold, it is possible to adapt *RecoCoDe* for multiple devices.

RecoCoDe operates directly at the logical volume level. In this way the hidden volume will conceal data redundantly. Thus, if some

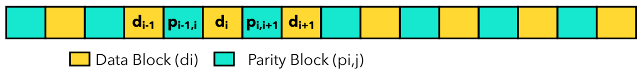

Figure 2: Example of an entanglement chain

of the blocks are rewritten during an examination, other blocks from the same logical volume may be used to recover data.

One of the main features of Shufflecake is the possibility to conceal multiple volumes. That raises the question of storing data redundantly across logical volumes. This strategy, however, does not enhance reliability. We have decided against this option because the volumes remain on the same physical drive, and using multiple volumes compromises the separation of concerns.

4.2 The Entanglement Algorithm

Simple data entanglements are a technique that outperforms the reliability of duplicating data (RAID 1 level) [8]. The additional reliability increases the chances to recover missing data without increasing the storage overhead of RAID 1 or compromising the decoding complexity as in other RAID-like solutions. Any single failure affecting, for example, a logic block of data is repaired with only two blocks. Finally, simple entanglements are a straightforward algorithm that eases the *RecoCoDe* implementation.

Here, we explain briefly the main concepts of entanglements and refer the reader to the Appendix A and the original paper [8] for more details. The main component is the entanglement chain, which is comprised of data and parity elements connected together in an alternating manner, as seen in Figure 2. The actual physical location of data and parity elements does not need to be contiguous. The parities are computed using the `XOR` operation between the previous two elements in the entanglement chain, as in:

$$p_{i,j} = p_{i-1,i} \oplus d_i \tag{1}$$

The idea behind Equation 1 is that each parity propagates redundant information from the previous data element in the chain. For example, $p_{4,5}$ propagates information from the previous data element in the chain, d_4, and together with d_5 will create $p_{5,6}$. Redundancy propagation increases the way data can be recovered. As opposed to data duplication that can only tolerates the loss of one copy only, simple entanglements can tolerate the loss of the data and its parity and other failure patterns. The analysis of irrecoverable failure patterns can be found in the Appendix A.4.

Simple entanglements are designed for log-structured append-only storage systems, where data can only be added and not modified or deleted once written. One could create more complex entanglements designed particularly for read-write systems, allowing users to modify files and directories as needed. If the user has storage space to spare or the reliability is of utmost importance alpha entanglement codes [7] may be considered. But, we leave these alternatives for further research.

5 RECOCODE IMPLEMENTATION

This section gives insights of RecoCoDe main components and our implementation. Appendix B proposes how to use it.

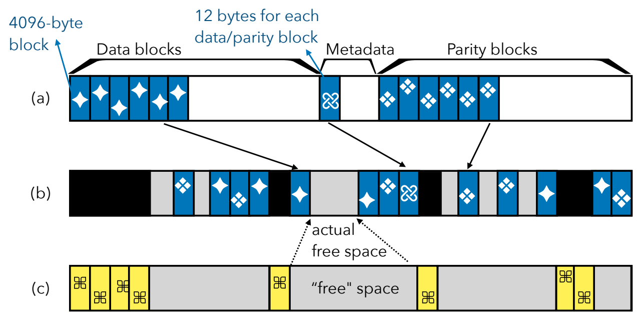

Figure 3: RecoCoDe Overview: (a) The logical entanglement layer contains data, parity, and metadata blocks. The 4096-byte blocks at the entanglement layer are stored in the underlying physical device. (b) When using a Shufflecake hidden volume, the previously entangled blocks and metadata are encrypted and allocated in the free space of the decoy volume. (c) When mounting a decoy volume without mounting the hidden volume, the visible free space may contain concealed data from the hidden volume.

5.1 Overview

RecoCoDe operates at the block layer, implemented in the Linux kernel. Entanglements can be implemented at the application, file system or block device levels. We opt for the block level since it is easier to create a block device mapper than creating a new file system and expect user adoption. Moreover, implementing entanglements at the application layer is impractical for our use case, as it would necessitate modifying any application that the user expects to use in conjunction with the PD tool. Finally, our implementation is done beneath the ext4 system, though it shall work with other file systems. This is relevant since Shufflecake is file system agnostic.

RecoCoDe intercepts bio requests to create a logical entanglement layer that is then stored in the underlying device. Figure 3 gives an overview of the solution. For simplicity the underlying device shows blocks instead of the slices.

- The logical entanglement layer contains data, parity, and metadata 4KB blocks, stored on the underlying physical device.
- The Shufflecake hidden volume includes a header (not shown). The entangled blocks and metadata are encrypted and allocated randomly within the free space of the decoy volume. While all volumes are mounted there is no risk of hidden data overwrite.
- When a hidden volume is not mounted, e.g., during an examination, the a decoy volume shows an ostensibly free space that may actually conceal data from the unmounted hidden volume.

5.2 The Device Mapper Framework

The Linux Device Mapper (DM) framework acts as a hardware abstraction layer that sits between the file systems and the actual block devices (like HDDs, SSDs, or RAID arrays). This abstraction provides flexibility by facilitating the operation with logical devices instead of operating direcly with the underlying hardware. Device mapper targets rest above one or more physical devices and appear as regular block devices to the upper layers. The DM framework supports stackable block devices. The kernel's call stack is used efficiently thanks to the recursion avoidance feature. Bio requests are queued and submitted to the next layer only when the parent bio submission completes.

5.3 Blocks I/O (bio) Requests

A block is a fixed amount of bytes used in the communication between a block device and its associated physical hardware. At the user space level, a block usually represents how much data is read from/ written to files in a single system call. At the logical level it represents the small unit in bytes that is addressable in that device. Our device mapper target operates with 4 KB data blocks, which is common for modern file systems.

5.4 RecoCoDe Device Mapper

RecoCoDe intercepts bio requests between the file system and the underlying device, essentially, it would be a hidden device created by Shufflecake or a similar tool. It should be noticed that our evaluation is done independently of the Shufflecake service, which is an independent project under ongoing development. Thus, in our case, the underlying device is directly the physical device. For each 4KB data block intercepted, we compute the parity block, and store both blocks on the device. The way we differentiate between data and parity blocks is by their address, i.e., the sector number.

The underlying device is organized in two halves, the first half containing data blocks and the second half containing parities. At intercepting the bio write request, *RecoCoDe* takes care of the following steps: 1) calculate the sector number for the parity block, 2) create a new request for that parity, 3) submit both the data and parity requests to the layer below, where the blocks will be persisted to disk.

The most recent block added to the chain is kept in cache memory for efficiency as it is used by the entanglement algorithm in the next write. The cache block is updated with every new element added to chain. Keeping the last element in memory eliminates the need of an additional read operation for every write.

RecoCoDe also intercepts read requests, which are directly resubmitted to the underlying device.

5.5 Block metadata

The metadata contains information related to the encoding and checksums to increase integrity. Ideally, we would persist metadata in a different device but *RecoCoDe* uses a single device. Hence these metadata blocks are also stored in the same drive.

When the user creates a file, blocks are allocated randomly. Although the entanglement algorithm is deterministic, we need to save the information of which blocks are entangled together. This is done by saving the block (logical) sector in the order in which they appear in the entanglement chain. Otherwise, it would be impossible to repair files.

Each block requires 8 bytes of memory to keep the sector. We also calculate a CRC32 checksum for each block that requires 4 additional bytes per block. In summary, we use 12 bytes per 4096-byte block (by block here, we mean both data and parity blocks) to store all the information we need to implement this mechanism. This is a less than 0.3% storage overhead.

6 RECOVERING FROM HIDDEN DATA OVERWRITES

Hidden data overwrites can occur as a consequence of mounting some of the volumes but not all of them. Because the PD tools over-commit storage space, the file system does not know that within the "free" space of a decoy volume there are actually used blocks containing the most valuable data for the user.

The allocation errors that cause the hidden data overwrites are silent. Thus, we need to first check if the volume has corrupted blocks. This is a straightforward process that goes through all blocks and compare their checksum with the actual content on disk. Corrupted blocks are flagged for later repair. We keep a bitmap which says if a block has been corrupted or not. Once corrupted blocks are properly identified we proceed to try repairing them. In this section we explain how repairs are done.

6.1 Repairs

The repair algorithm inspects the entanglement list to check whether or not the blocks are corrupted. For each corrupted block, it checks if the required blocks for the repair are not damaged. For a data block, it requires the two adjacent parity blocks, i.e. the previous element and the next element in the entanglement chain. For a parity block, it requires either the two previous elements (so one data, one parity block), or the next two elements in the entanglement chain. The repair proceeds if the required blocks are intact. If not, the algorithm recursively repairs the required blocks until it can repair the one that initiated the recursive call.

In these recursive calls, we need a way to know if the blocks are irrecoverable. As in any repair mechanism based on redundancy, the algorithm succeeds only if the system has enough redundancy to recover the missing data. As presented in Appendix A.4, entanglement chains have three irrecoverable failure patterns that will prevent block repairs inside those pattern. Our algorithm recognizes the patterns and marked the blocks as irrecoverable. We can do that in the following way. Firstly, we only start repairing data blocks. More specifically, in the recursion tree, we only want data blocks as roots. That way, if we run into a corrupted data block in the recursion while repairing another data block, that means we ran into either type B or type C failure (as indicated in Appendix A.4). In this case, we mark all blocks in the recursion as irrecoverable. The procedure stops when all the elements of the chain are examined and recovered when possible.

Algorithm 1 and Functions repairBlock, repairBlockRec represent the repair process and the `irrecoverable_blocks_bitmap` represents the bitmap which tells which blocks are irrecoverable. The bitmap size is equivalent to the total number of device' sectors. `corrupted_blocks` also represents a bitmap, this one just telling us which blocks have been corrupted. `entanglement` is a doubly-linked list of blocks representing the entanglement. The resulting bitmap is not stored anywhere. It serves as a tool for checking for irrecoverable blocks during the execution of the algorithm.

Algorithm 1: Finding irrecoverable blocks

```
Result: Repaired blocks, and a bitmap with information on
        which blocks are irrecoverable
irrecoverable_blocks_bitmap := $[0]^N$;
corrupted_blocks;
entanglement := list of blocks;
for block ∈ entanglement do
    if block is data block ∧ block is corrupted ∧ block is
      recoverable then
        repairBlock(block, irrecoverable_blocks_bitmap);
    end
end
```

Function repairBlock(block, irrecoverable_blocks_bitmap)

```
leftBlock;
rightBlock;
leftBlockState := REPAIRED;
rightBlockState := REPAIRED;
if leftBlock is corrupted ∧ leftBlock is recoverable then
    leftBlockState = repairBlockRec(leftBlock,
      irrecoverable_blocks_bitmap, LEFT);
if rightBlock is corrupted ∧ rightBlock is recoverable then
    rightBlockState = repairBlockRec(rightBlock,
      irrecoverable_blocks_bitmap, RIGHT);
if leftBlockState == IRRECOVERABLE ∨ rightBlockState ==
  IRRECOVERABLE then
    irrecoverable_blocks_bitmap[blockSector] = 1;
    return;
repairedBlock := leftBlock ⊕ rightBlock;
persistToStorage(repairedBlock);
```

Function repairBlockRec(block, irrecoverable_blocks_bitmap, direction)

```
nextDataBlock;
nextParityBlock;
resultState;
if nextDataBlock is corrupted then
    irrecoverable_blocks_bitmap[blockSector] = 1;
    return IRRECOVERABLE;
resultState = repairBlockRec(nextParityBlock,
  irrecoverable_blocks_bitmap, direction);
if resultState == IRRECOVERABLE then
    irrecoverable_blocks_bitmap[blockSector] = 1;
    return IRRECOVERABLE;
repairedBlock := nextDataBlock ⊕ nextParityBlock;
persistToStorage(repairedBlock);
return REPAIRED;
```

7 RECOCODE AND SHUFFLECAKE

Our solution decouples the data encoding performed by *RecoCoDe* from the data encryption performed by Shufflecake. In this section we present the design rationale and how *RecoCoDe* can be possibly integrated with Shufflecake or similar PD tools.

7.1 Design Rationale

Our initial approach for was more integrated with Shufflecake and based on the ext4 filesystem. Those decisions simplified the rapid prototyping of our ideas but reduced the flexibility of the solution.

We first duplicated data on the second half of a Shufflecake volume. Paradoxically, our first attempts corrupted data at the filesystem level causing unpredictable crashes. The cause was that our redundancy system overwrote filesystem metadata, i.e., superblocks. After some effort done to solve this problem, we realized that tailoring our solution specifically for ext4 compromises Shufflecake's filesystem agnostic design. In the end, we discarded this approach.

Instead, we opt for a device-mapper target, which creates a virtual layer and defines how data is mapped from the virtual device to underlying physical devices. This can involve complex operations like striping data across multiple devices (as in RAID configurations), or encrypting data on-the-fly. This novel kernel module would take care of the data encoding with simple entanglement codes, and all the necessary details needed to make it work. Such a module can be loaded into the kernel, and Shufflecake could use it as a sort of black box, without knowing any implementation details.

7.2 RecoCoDe Integration with Shufflecake

RecoCoDe is simply put at front of the deniable storage solution (Figure 3). The implementation aspects of this work were time-constraint to two master's semester project iterations of 14 weeks, each with a steep learning curve. The first iteration did not consider the current *RecoCoDe* solution but other alternatives. When we started with this work, Shufflecake was under development and many aspects were unstable. Being an independent project many aspects were outside of our control. Today, Shufflecake is incorporating some features and API that will help to integrate our solution in a more seamless manner. Otherwise, to load *RecoCoDe* into Shufflecake, we would need to modify some parts of the Shufflecake codebase. When a hidden data overwrite occurs, it is due to a double allocation problem that needs to be fixed in Shufflecake [6]. Finally, in this paper we only consider a single hidden volume. When using *RecoCoDe*with Shufflecake, one should create a virtual entangled layer for each hidden volume that aims to protect with redundancy.

8 EVALUATION

We evaluate our *RecoCoDe* prototype implemented mostly in C and compare the write request performance when writing directly onto the block device. For all the testing and debugging purposes, and during the research for this project in general, we use a USB stick of 1GB in size as the underlying block device. More precisely, the USB has a 32GB size and it was partitioned into a small device of 1GB to make it easier to test and debug.

8.1 Write Requests Performance

To evaluate our implementation, we measure the speed of write requests since *RecoCoDe* introduces another layer between the filesystem and the block device. We expect that the user is not considerable affected by the addition of our solution.

For the speed of the write requests, we use the Linux `dd` and `time` commands. First, we create files filled with random bytes, and wrote them to both the block device directly, and to the block device through my device-mapper target. This gives us a clear picture of whether or not the process of creating new parities, calculating checksums, etc. affects the time it takes to write an arbitrary file, if it affects it at all.

We tested the implementation on various workloads, repeating the process for 30 iterations in order to observe the average and the standard deviation of the requests. The results of these tests can be seen in the two tables below.

Results from the tests on the block device directly						
	1MB	5MB	10MB	50MB	100MB	200MB
Mean	10	35	57	243	480	940
SD	4	13	11	10	10	12

Table 1: Time in msec (mean and standard deviation) for writing files of a particular size directly onto a block device.

Results from the tests using the device-mapper target						
	1MB	5MB	10MB	50MB	100MB	200MB
Mean	10	38	58	241	482	927
SD	4	15	13	10	15	20

Table 2: Time in msec (mean and standard deviation) for writing files through the entanglement device-mapper target

From these results, we observe that our new entanglement device-mapper target does not impact the performance of write requests from the user. The solution also scales well. This is generally what was desired, to simply have the redundancy scheme working in the background, without the user having to sacrifice any time.

The bash scripts for these particular tests, as well as the results, can be found in the `speed_tests` directory of this project [1].

9 DISCUSSION

RecoCoDe is a prototype not fully functional or ready for production. Adding redundancy to a coercion resistant tool such as Shufflecake proved to be more challenging than anticipated. While the implementation described here could be used by a savvy user, the current usage is limited as shown in Appendix B.

9.1 The `mdadm` and RAID approach

The Linux tool `mdadm` stands for "multiple disk administration" and is a utility in Linux used to manage software RAID devices. It allows users to create, manage, and monitor RAID arrays within the Linux environment using the disks available on the system. We tried this option as an alternative approach to add redundancy for data in the hidden volume. We report some of the encountered problems here. When compared to plain Shufflecake, the mdadm approach has a performance overhead of 1-3x for reads and 2-5x for writes. Our measurements may have caused some overhead but the additional

[1]URL removed for peer-review anonymity.

RAID layer could turn out to be the culprit. We did not investigate further since we also found that this method requires an excessive use of Shufflecake slices during repairs because the RAID manager cannot see Shufflecake slices. That was a problem, because at least at the moment of doing this operations Shufflecake did not garbage collected slices so slices cannot be reclaimed once assigned, which renders this option impractical.

9.2 Append-only File System

Our prototype doesn't restrict or handle block deletions or modifications. This should not be an issue for our use case, a user wants to protect/preserve documents. If the user deletes content in the hidden volume, *RecoCoDe* does not have a way to know that the content was deleted. Two cases can occur: (A) While a block is marked free but not rewritten, our decoder may use the block for repairing other blocks in the future. (B) If a block is rewritten after deletion, *RecoCoDe*' scan will detect it as data corruption. The failure detection does not distinguish between content changes due to deletions or content changes due to hidden data overwrites. Some possible solutions: If the filesystem is ext4, the user could add the append-only attribute to the file or filesystem. Alternatively, Security-Enhanced Linux (SELinux) can be used for creating system wide policies in consistency with an append-only logical volume. RecoCoDe may interact with Shufflecake to know the mapping of blocks, slices, and volumes to improve the failure detection.

In addition, RecoCoDe will need to handle the decrease of redundancy. This requires additional research that falls outside the scope of this paper. Entanglements work by propagating redundancy across a system. If some blocks of the entanglement chain are deleted, the decoder may stop earlier due to lack of those blocks.

9.3 Metadata Protection

Entanglements protect data blocks, but the metadata blocks (containing checksums and other coding related information) are not further protected. Since *RecoCoDe* works regardless of what the underlying device actually is, it just writes the metadata onto the underlying device. In case of the device being a Shufflecake volume, that same metadata will be written to slices as per the Shufflecake device mapper target. This means that the slices containing the entanglement metadata are susceptible to corruption, much like other data on said slices. Worthy to mention, *RecoCoDe* does not require too much metadata. The ratio between user information (data/parity blocks) and metadata is 341:1. Thus, there are lower chances that a hidden data corruption event will affect the metadata block. One can always replicate metadata blocks to increase the chances to survive an allocation error.

9.4 Wear Leveling

Filesystems manage storage devices via the device's firmware and drivers rather than direct interactions with memory cells. Wear leveling, crucial for flash memory management, redistributes data uniformly across the device to extend its lifespan and improve redundancy across different failure domains. At the same time, wear leveling mechanisms may compromise plausibly deniability by leaking information that can be used to detect the existence of a hidden volume. During an investigation, wear leveling logic can expose critical information about past traces or even the traces themselves [5]. Consequently, Shufflecake advises against using wear leveling.

10 RELATED WORK

Data integrity is widely missing in the PD literature. In particular, the data corruption problem is ignored.

TrueCrypt provides *protection of hidden volume damage* by making the hidden volume read-only. This protection prevents that content in the hidden volume gets overwritten when writing to the outer volume. To activate this protection the user needs to check the option before mounting the outer volume. It requires that the user introduces the password for the hidden volume. TrueCrypt is a discontinued software.

Artifice [4] combines the challenges of data concealment and hidden data overwriting in one move using an information dispersal algorithm. It leverages Shamir's Secret Sharing (SSS) or systematic erasure codes, along with an all-or-nothing transformation, to create carrier blocks. Through these techniques, Artifice establishes a (n, k) configuration, requiring a minimum of k carrier blocks from a total of n to restore the initial data. If less than k blocks are present, no data can be retrieved. SSS is closely related to Reed Solomon Codes. Simple entanglement codes and its more general version alpha entanglement codes depart from traditional erasure codes bringing interesting properties for plausibly deniability.

While tool detection falls outside the scope of this paper, it is worth to discuss it here. The Linux kernel 2.6 and later versions use dm-crypt as its disk encryption subsystem. It encrypts entire block devices allowing various encryption models. Invisiline [15] is a recent multi-snapshot solution relies on dm-crypt, and stores hidden data in dm-crypt's initialization vectors (IV), ensuring that changes to hidden data do not reveal the use of a PD system.. It introduced the concept of plausible invisibility to describe a situation in which adversaries cannot even detect the use of a PD system. Unfortunately, Invisiline does not eliminate the risk that the concealed data might be inadvertently deleted. If the disk is unmounted and the solution is uninstalled, the user can still read public data with the vanilla dm-crypt. However, any write operations may overwrite the hidden data permanently, i.e., the deleted data cannot be recovered even if the tool is re-installed. This problem indicates that there is a larger space of solutions that can benefit from solutions like ReCoCoDe.

11 CONCLUSION

This paper highlights the importance of the hidden data overwrite problem in plausibly deniable tools. To our knowledge, *RecoCoDe* is the first attempt to mitigate this problem with a redundancy layer that can be combined with Shufflecake or other similar tools.

ACKNOWLEDGMENTS

We are grateful to Tommaso Gagliardoni and Elia Anzuoni for their support with Shufflecake and insightful discussions. We thank Kilian d'Eternod for his early involvement in this project and Simone Colombo for his feedback on preliminary work. We are grateful to Cornell Tech and the IC3/Ethereum Grant who partially funded this reasearch.

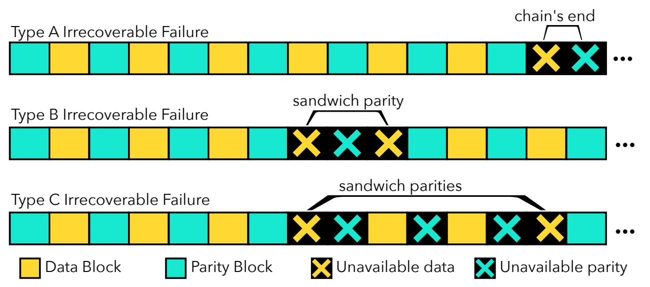

Figure 4: The three (3) types of failures where some blocks are irrecoverably lost.

REFERENCES

[1] Ross Anderson, Roger Needham, and Adi Shamir. 1998. The steganographic file system. In *International Workshop on Information Hiding*. Springer, 73–82.

[2] Elia Anzuoni. 2022. *Hidden Filesystem Design and Improvement.* Technical Report.

[3] Elia Anzuoni and Tommaso Gagliardoni. 2023. Shufflecake: Plausible Deniability for Multiple Hidden Filesystems on Linux. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security.* 3033–3047.

[4] Austen Barker, Staunton Sample, Yash Gupta, Anastasia McTaggart, Ethan L Miller, and Darrell DE Long. 2019. Artifice: A deniable steganographic file system. In *9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19).*

[5] Chen Chen, Xiao Liang, Bogdan Carbunar, and Radu Sion. 2022. SoK: Plausibly Deniable Storage. *Proceedings on Privacy Enhancing Technologies* 2 (2022), 132–151.

[6] Shufflecake developers. 2023. Repair Corrupted Physical Slice Indexes (PSIs). https://codeberg.org/shufflecake/shufflecake-c/issues/67 [Accessed: (April 20th, 2024)].

[7] Vero Estrada-Galiñanes, Ethan Miller, Pascal Felber, and Jehan-François Pâris. 2018. Alpha entanglement codes: practical erasure codes to archive data in unreliable environments. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 183–194.

[8] Vero Estrada-Galiñanes, Jehan-François Pâris, and Pascal Felber. 2016. Simple data entanglement layouts with high reliability. In *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC).* IEEE, 1–8.

[9] Clemens Fruhwirth. 2017. LUKS On-Disk Format Specification Version 1.2. http://cdn.kernel.org/pub/linux/utils/cryptsetup/LUKS_docs/on-disk-format.pdf [Accessed: (April 1st, 2024)].

[10] Evkan Hülya, Norman Lahr, Ruben Niederhagen, Richard Petri, Andreas Poller, Philipp Roskosch, and Michael Tröger. 2020. *Security Evaluation of VeraCrypt.* Technical Report. Fraunhofer Institute for Secure Information Technology (SIT) on behalf of the Federal Office for Information Security (BSI).

[11] Louis-Henri Merino, Alaleh Azhir, Haoqian Zhang, Simone Colombo, Bernhard Tellenbach, Vero Estrada-Galiñanes, and Bryan Ford. 2024. E-Vote Your Conscience: Perceptions of Coercion and Vote Buying, and the Usability of Fake Credentials in Online Voting. In *2024 IEEE Symposium on Security and Privacy (SP).*

[12] Acting Commisioner of the U.S. Customs and Border Protection. 2018. U.S. CBP Directive for Border Search of Electronic Media. https://www.cbp.gov/sites/default/files/assets/documents/2018-Jan/CBP-Directive-3340-049A-Border-Search-of-Electronic-Media-Compliant.pdf [Accessed: (Feb 28th, 2024)].

[13] David A Patterson, Garth Gibson, and Randy H Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data.* 109–116.

[14] André Pilarczyk. 2015. TrueCrypt Documentation. https://www.truecrypt71a.com [Accessed: (Feb 25th, 2024)].

[15] Sandeep Kiran Pinjala, Bogdan Carbunar, Anrin Chakraborti, and Radu Sion. 2023. INVISILINE: Invisible Plausibly-Deniable Storage. In *2024 IEEE Symposium on Security and Privacy (SP).* IEEE Computer Society, 18–18.

[16] James S Plank. 1997. A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems. *Software: Practice and Experience* 27, 9 (1997), 995–1012.

A SIMPLE ENTANGLEMENT CODES 101

A.1 Opened vs Closed Chain

ReCoCoDe uses open entanglements. An entanglement chain can be, however, opened or closed. They both work in the same way: for each data element, the encoder creates a parity element. Though, the data element at the end of an opened chain, as shown in Figure 2, is less protected than previous elements because its information is not yet propagated. Thus, the benefit of a closed chain is to provide better protection to the elements located at the end. A closed chain creates a connection between the last and first elements of the chain, by recomputing the second element of the chain. A closed chain can be re-opened to continue incorporating more elements to the chain. On the other hand the cost of closing the chain and re-opening in the next write may be considerable. We opted for open chains to optimize writing performance.

A.2 Space Efficiency

In terms of space efficiency, simple entanglements is a better approach than mirroring. In other words, simple entanglements can recover more data using the same space overhead as duplication.

A.3 Repairs

Due to the properties of the XOR operation, recalculating (i.e. repairing) elements is a simple step. For example, if d_2 is corrupted, we can recover it by computing $p_{1,2} \oplus p_{2,3}$ according to Equation 1. For corrupted parity elements, we have two potential equations. Following with our example, assuming that $p_{2,3}$ is not available, we can recover it using any of these two choices: either $p_{1,2} \oplus d_2$, or $d_3 \oplus p_{3,4}$. After successfully recovering the parity, we can proceed to recover d_2.

A.4 Irrecoverable Failures

Simple entanglements have three *irreducible fatal failure patterns* as shown in Figure 4. An irreducible fatal failure patterns is defined with the minimum number of failed elements that causes data loss. A recover process cannot recover data if those elements are missing.

- *Type A* refers to a failure pattern in which both the last data and parity elements are missing or corrupted.
- *Type B* refers to a pattern in which two consecutive data elements and the parity between them is unavailable. [2]
- *Type C* is an extension of type B and describes the pattern where two data blocks and all parity blocks between them fail.

It is worth noticing that type A pattern can only appear at the chain's end while type B and C can appear at any place of the chain. If any of these patterns appear, the repair algorithm cannot recover the missing elements. They are gone forever.

B USAGE

Our RecoCoDe implementation does not have a user interface. The module should be inserted into the kernel, with the insmod command. The program can be run through the terminal by running

[2]We can refer to this pattern as as the sandwich parity.

the executable file, called *entanglement app*, with the following commands as parameters:

- `init:` initializes the device [3], and opens it for the user to use freely. Before actually writing files to this new device, the user first has to create a filesystem and mount it. The basic example would be to use the mkfs.ext4 [8] and mount [9] commands.
- `open:` opens the device, loading all the necessary things such as the entanglement into memory. At this point, the user can just start writing files, because the device was already set up beforehand.
- `close:` closes the device, removing it from the kernel. The command also ensures that everything which is necessary was persisted to storage before removing the device.

As additional parameters to these commands, the user should specify the path to the underlying block device, and in the case of the open command, whether or not the user wants to run a repair on any potentially corrupted blocks according to the procedure presented in Section 6.

[3]In this context, we refer to the new logical device created by the device-mapper target, not the actual block device underneath