Shufflecake: Plausible Deniability for Multiple Hidden Filesystems on Linux

Elia Anzuoni[1,2] and Tommaso Gagliardoni[2] ✉

[1] EPFL, Switzerland
`elianzuoni@gmaildotcom`
[2] Kudelski Security, Switzerland
`myfirstname@gagliardoni.net`

Abstract. We present Shufflecake, a new plausible deniability design to hide the existence of encrypted data on a storage medium making it very difficult for an adversary to prove the existence of such data. Shufflecake can be considered a "spiritual successor" of tools such as TrueCrypt and VeraCrypt, but vastly improved: it works natively on Linux, it supports any filesystem of choice, and can manage multiple volumes per device, so to make deniability of the existence of hidden partitions really plausible. Compared to ORAM-based solutions, Shufflecake is extremely fast and simpler but does not offer native protection against multi-snapshot adversaries. However, we discuss security extensions that are made possible by its architecture, and we show evidence why these extensions might be enough to thwart more powerful adversaries.

We implemented Shufflecake as an in-kernel tool for Linux, adding useful features, and we benchmarked its performance showing only a minor slowdown compared to a base encrypted system. We believe Shufflecake represents a useful tool for people whose freedom of expression is threatened by repressive authorities or dangerous criminal organizations, in particular: whistleblowers, investigative journalists, and activists for human rights in oppressive regimes.

Keywords: Shufflecake · TrueCrypt · VeraCrypt · plausible deniability · privacy · forensics · disk encryption · LUKS · dm-crypt · cryptsetup
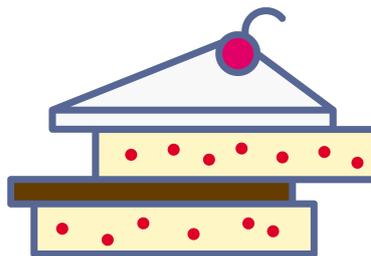
A 15-page abstract of this work appears (with the same title) in the proceedings of the *ACM Conference on Computer and Communications Security (CCS) 2023*. This is the authors' full version. This document supersedes any previous versions.

Original Version: 2023-10-06.   This Revision: 2023-10-06

Table of Contents

1 Introduction

Privacy of personal and sensitive data is, now more than ever, a topic of major public interest. In today's heavily interconnected world where data are, often by default, entrusted to an online third party, the last bastion of data confidentiality is local storage, as the physical disconnection greatly helps in reducing the room for abusive access. Even there, however, some protection measures need to be implemented, to guard against adversaries who might close that gap. The most trivial example of such an adversary is a thief stealing a user's personal hard disk and reading its raw contents; this is a very simple and well-studied threat model, for which many robust disk encryption solutions exist.

However, disk encryption alone is not enough to handle adversaries empowered by repressive laws or other, less legal methods (e.g., "rubber-hose"). Unlike the previous scenario, these adversaries gain more than simple "offline" access to the disk: they are in a position of power, which they can use to directly and aggressively confront the user about the contents of the protected storage, and by means of (physical, legal, psychological) coercion, they can obtain the encryption keys to any encrypted content identifiable on the user's device. The security goal in this scenario, then, becomes to still retain secrecy of some selected, "crucial" data on the disk, by making the presence of such data not even identifiable, thus allowing the user to make up a credible lie about the storage contents. This is exactly the aim of *plausible deniability (PD)*, a powerful security property, enabling users to hide the existence of sensitive information on a system under inspection by overreaching or coercive adversaries.

In the context of secure storage, PD refers to the ability of a user to plausibly deny the existence of certain data stored on a device even under interrogation or forensic inspection of the physical device. The underlying idea is that, if the adversary cannot conclude anything about the existence of hidden sensitive data, they have no motivation to further continue the coercion, thereby (hopefully) limiting the damage for the user. PD was first proposed in 1998 [1] and, since then, many different PD solutions have emerged, attempting to balance the security-efficiency trade-off. One of the most popular PD solutions was TrueCrypt [46], first released in 2004, discontinued in 2014, and replaced by its backward-compatible and technically similar successor VeraCrypt [49].

TrueCrypt and VeraCrypt remain the most well-known PD disk encryption tools available, probably because of the large open source community around them and the good performance, but they suffer from many drawbacks that have been left unaddressed for many years now, both in terms of security and operational model, such as: the possibility of only having *one* layer of extra secrecy, limited filesystem support, and limited functionalities on Linux. In this work we present Shufflecake, a novel disk PD tool that aims at solving these and other limitations by still achieving a good security-performance tradeoff.

1.1 Motivation

Albeit less extensively covered by the existing literature, coercive adversarial attacks unfortunately represent many diverse real-world situations. Most of the demonstrable facts, in this regard, concern national security provisions in countries like the USA, France, or the UK, that allow prosecutors to legally oblige citizens to disclose the passwords to their encrypted storage devices [21,25,38,40,48], under threats of harsh legal or economic penalties for non-compliance. These provisions are reported to have been often misused, sometimes to the point where people's rightful privacy has been arguably trampled on [2,4,8,50].

The relative abundance of such reports coming from Western countries, however, is likely due to the comparatively attentive and critical public oversight on the government's operations. In countries with a less-developed system of checks and balances, the precise extent to which the sensitive data of activists, journalists, dissidents, and oppressed minorities are violated, can only be hinted at by the few sporadic cases that occasionally make it to the international headlines [7,44]. These kinds of coercive attacks are not merely "tinfoil-hat paranoia", but a real-world concern.

1.2 Previous Work

Different approaches to PD on storage have been proposed, starting from the layer one chooses to intervene at. Digital storage, in fact, is composed of many stacked layers, from the topmost "logical" layer (the filesystem) down to the more physical one. Such a layered structure complicates the security analysis, as different PD solutions focus on different layers, each of them leading to different approaches with pros and cons. Certain solutions work at the filesystem layer [28, 35]; they have to implement a rich interface, made of complex file- and directory-oriented methods (`fileOpen`, `fileRead`, `mkDir`...). Other schemes choose instead to go at the lowest level and modify the FTL [23] (flash-translation layer, for SSDs), but this approach clearly leads to highly vendor-specific solutions. Security of solutions designed for a specific layer might be defeated by adversaries with access to lower layers.

A versatile approach for a robust PD solution is arguably to operate at the *block layer*, whereby the scheme exposes a *block device* interface (a common abstraction layer used by many operating systems to represent storage devices as arrays of fixed-size data *blocks*), providing just a `bRead` and a `bWrite` method. This is the approach used by solutions like TrueCrypt, and also by Shufflecake; in the remainder of this work, we will only focus on block-layer solutions. In this framework, a single underlying *disk* or *device* is formatted to host one or more *volumes* (logical storage units, usually represented as virtual block devices), each encrypted with a different (usually password-derived) symmetric key. When confronted with the adversary, the user surrenders the passwords to some of these volumes, called *decoy* volumes (because in some use cases they might contain deceptively innocent data): PD security guarantees that, even after these passwords have been given up, inspection of the disk by the adversary still yields

no clue whatsoever hinting at the presence of some further, *hidden* volumes. The intuition behind the formal definition of these guarantees, is that the adversary cannot distinguish between the case where all the passwords have been surrendered, and the case where there is still undisclosed secret information left.

Security models. Early PD literature has focused mostly on *single-snapshot adversaries*, who are assumed to only check the storage device once. This was considered a natural assumption: in the typical scenario, an activist or journalist is stopped at a checkpoint or arrested and interrogated one time, and her electronics confiscated and analyzed. Provided she manages to escape, she will be on high alert for future investigation, and in particular she will *refresh* the PD protection in use through some specific procedure (e.g., by reformatting the hard disk, or by buying a new device). So, in case of a second check, the adversary will de facto face a completely new *instance* of PD storage, therefore falling back to the single-snapshot case. This is the threat model addressed by solutions like TrueCrypt and VeraCrypt.

The safety of the single snapshot threat model, however, has been questioned in the literature over time, not only because it relies on good user security practice, but also because the technological evolution of storage brought a new issue on the table. In modern devices, especially solid-state disks (SSDs), overwriting a logical sector often results in the underlying physical sector being simply marked as "unused" rather than being really overwritten, thereby leaving "traces", or "snapshots" of the data content at previous points in time. This in turn can (in theory) allow the adversary to break plausible deniability even with a *single* inspection, because by analysing these traces that are left on the device one can see that content at certain locations has changed; since empty, unused space should not change over time, the presence of hidden information therein can be betrayed. This is the scenario considered in *multi-snapshot security* models.

One could argue that multi-snapshot attacks are likely to be very complex, to the point that 100% evidence of the presence of a hidden volume based only on past sector traces is unlikely to be reached, and an accusation in this sense might not stand in court. In fact, we are not aware of a *single* case in public literature of a conviction due to multi-snapshot attacks. On the contrary, there are many documented cases [36,37,47] where even a simple system such as TrueCrypt was enough to grant acquittal of a suspect. This is not to say that single- and multi-snapshot security are equivalent: the latter is stronger. However, one should question what price it is reasonable to pay (in terms of performance, etc) to achieve this stronger security.

ORAMs. Multi-snapshot attacks are a well-known issue in PD systems (True-Crypt and derivatives are also vulnerable in this sense) and designing countermeasures turns out to be challenging. As of today, only a few constructions [10] achieve multi-snapshot security, but at a hefty performance cost that makes them not practical for most use cases. Most of these solutions are based on *oblivious random access machines (ORAMs)*. ORAMs are cryptographic schemes that aim at hiding the access patterns (in addition to the data content itself) of a trusted

agent accessing an untrusted storage. The connection between ORAMs and PD has been investigated since 2014 with the HiVE construction [6]. In a nutshell, the idea is that if we use an ORAM to access a device, then nobody, not even a run-time backdoor in the device firmware, can know which (logical) location we access and how, thereby providing a solid method for implementing PD. However, ORAMs are extremely slow: it is known [26] that the bandwidth overhead of any secure ORAM of size n is $\Omega(\log(n))$. The HiVE paper circumvented this problem with the following observation: If we are not worried by run-time backdoors in the device firmware, but are only concerned about "traditional" multi-snapshot adversaries, i.e. post-arrest investigation of the device physical layer, then we do not need a fully-fledged ORAM, because `read` operations do not change the state of the device. So all we need is a *"write-only" ORAM (WoORAM)* that only obfuscates `write` requests. The advantage is that there is no known efficiency bound for WoORAMs, and in fact existing WoORAM constructions seem to be slightly better than full ORAMs. The WoORAM approach sparked a whole new line of research in multi-snapshot resistant PD solutions [9, 39], and it has been proven [10] that resistance to multi-snapshot security under certain assumptions is *equivalent* to the use of WoORAMs.

1.3 Limitations of Existing Solutions

So far, the landscape of available PD solutions presents many gaps, both in usability and in security, a fact also hinted at by the relatively scarce adoption of such solutions. By far the most widespread today is VeraCrypt, which comes with many limitations. WoORAM-based techniques have been studied in the last few years as promising alternatives to address TrueCrypt and VeraCrypt's security issues. However, it is important to stress that even the most performant WoORAM-based schemes are still very slow or wasteful. To put this in perspective: HiVE has a slowdown of roughly 200x I/O throughput and wastes 50% of the disk space, while some recent constructions such as DetWoOram [39] reach a slowdown of "only" 5x but at the cost of wasting 75% of the disk space. This leaves us with the dilemma of either choosing single-snapshot, efficient solutions with limited security, or WoORAM solutions with unacceptable performance loss and arguably stronger security.

Moreover, WoORAMs solutions themselves might not be bulletproof. In fact, we believe that the idea that `read` requests do not change the underlying state of the physical device is a somewhat strong assumption, and hard to justify with modern, complex SSDs that might, for example, cache `read` requests in some undocumented memory area of the firmware, or store read data on an ad-hoc buffer to improve performance, etc.

Another big problem of many plausible deniability solutions (including TrueCrypt) is that the OS itelf (or other applications installed therein) can unintentionally leak to an adversary the presence of hidden data when a hidden volume is unlocked. This can happen for example through the OS logging disk events, search agents indexing files within the hidden volume when this is unlocked, even applications such as image galleries or document readers caching previews

of opened documents. Customizing the OS' behavior in such a way to avoid these pitfalls is an almost hopeless task [11]. A proposed solution to this problem is to have the OS itself *inside* a hidden volume, which is the idea that led to the concept of "hidden OS" on TrueCrypt. However, as far as we know, TrueCrypt (and VeraCrypt) remain the only implementation of this idea, limited to the Windows OS. Overall, we can say that a versatile PD solution able to balance security and usability has been sorely missing for years, especially for Linux, where no really practical solution exists.

1.4 Our Contribution

In this work we present *Shufflecake*, a novel PD scheme that aims at striking a balance between the efficiency of TrueCrypt and the security of WoORAM-based solutions. Shufflecake operates at the block device layer, like TrueCrypt, but with important improvements:

1. It offers a virtually unlimited number of hidden volumes per-device, arranged hierarchically: for the user it is sufficient to unlock one volume, and all the "less secure" volumes will be unlocked automatically. This improves user experience and operational security, as we will see in Section 4.2.
2. Unlike TrueCrypt, Shufflecake is *filesystem-agnostic*, meaning that all its features are available regardless of the filesystem chosen by the user.
3. It works natively on Linux, and can be integrated with the kernel for use at boot time and for *hidden operating systems*.
4. Unlike WoORAM-based solutions, Shufflecake is extremely fast (achieving only a minor slowdown compared to a bare, non-PD system) and wastes less than 1% of the disk space.

Moreover, Shufflecake not only achieves (provable) single-snapshot security, but also implements features that could make possible in the future to achieve a form of "operational" (i.e., weak) multi-snapshot security. These features are Shufflecake's hierarchical design and atomic block-rerandomisation, which are not available in tools such as TrueCrypt. We discuss this in Section 6.

We implemented Shufflecake [45] in the C language, and we released it as a free software under the GNU General Public License v2+.

1.5 Acknowledgements

2 Preliminaries

In this section we give the required preliminaries that are going to be used in the rest of this work. In the following, we use "iff" as "if and only if". Array indices start from 1. By (efficient) *algorithm* or *procedure* we mean a uniform family of circuits of depth and width polynomial in the index of the family. We implicitly assume that all algorithms take the index of the family as a first input, so we will often omit this. In the case of cryptographic algorithms, we call such index a *security parameter*, and we denote it by λ. We will often label algorithms with names that reflect their role, e.g. "adversary", "distinguisher", etc. If an algorithm A is deterministic, we denote its output y on input x as $y := A(x)$, while if it is randomized we use $y \leftarrow A(x)$; when derandomising an algorithm we look at the deterministic algorithm obtained when considering explicitly the internal randomness r as an additional auxiliary input, and we write $y := A(x; r)$. We will call *negligible* (and denote by $\mathsf{negl}(x)$) a function that grows more slowly than any inverse polynomial in x, and *overwhelming* a function which is 1 minus a negligible function. Given an event E, we denote by $\bar{E}$ its negation. Finally, we will write $x \xleftarrow{\$} X$ if x is sampled uniformly at random from a set X.

2.1 Cryptographic Primitives

We assume familiarity with elementary cryptographic constructions and we refer the reader to, e.g., [24] for a more in-depth dive. Here we just recap informally the most relevant concepts.

Cryptographic security. We will often define the security of a cryptographic scheme Π in terms of a *game*, or *experiment*, that captures the 'difficulty in breaking the scheme', leading to so-called *game-based security*[3]. This usually entails comparing the probability of the adversary $\mathcal{A}$ (modeled as an efficient algorithm) in winning a Game, that is, breaking the scheme, versus the baseline probability of 'winning by pure chance', for example by guessing randomly. We call this difference of probabilities the *advantage* of the adversary winning the game for Π, and we define (computational) security by requiring that this advantage is negligible in λ for any (computationally bounded) adversary.

$$\mathbf{Adv}_{\Pi}^{\mathsf{Game}}(\mathcal{A}) := \left| \Pr\left[\mathcal{A}\left(\Pi, \mathsf{Game}\right) \rightarrow win\right] - \Pr\left[\mathsf{Guess}\left(\Pi, \mathsf{Game}\right) \rightarrow win\right] \right| \leq \mathsf{negl}.$$

Hash functions and KDFs. A *hash function* is an algorithm that maps strings of arbitrary length into strings of fixed length (e.g., 256 or 512 bits). The most relevant security property for hash functions in our case is *collision resistance*, meaning that it is computationally difficult to find two distinct input strings that map to the same hash value. In order to add resistance against *pre-computation*

---

[3] Other frameworks exist, such as simulation-based, but as a first approximation game-based security notions are very convenient for their intuitivity and simplicity.

attacks [32], most implementations of hash functions use an additional parameter, called *salt* (usually a non-secret, per-application string of 96-256 bits), to further randomize their mapping. Typical hash algorithms for cryptographic use are SHA256 [16], SHA-3 [12], and BLAKE2 [42].

Hash functions are designed to be very fast and efficient in term of required computational resources. This might actually be an undesirable property when using the function to store images of user-chosen passwords, because it allows for faster adversarial brute-force. In these cases, a *key derivation function (KDF)*, should be used. KDFs are functionally similar to hash functions, but are designed in such a way to be *uniformly expensive to compute* on a broad range of computing devices, for example by requiring not only many CPU cycles but also large amount of memory and low latency. Typical KDFs for cryptographic use are Argon2id [5] and Scrypt [34].

Symmetric-key encryption and authentication. Regarding encryption, the most fundamental primitive is *symmetric-key encryption (SKE)*, also called *secret-key encryption*. An SKE scheme is a pair of algorithms (one for encryption and one for decryption) which define a bijection between a domain (*plaintext space*) and a co-domain (a subset of the *ciphertext space*) of strings. An additional input, the *secret key*, fixes the bijection across the set of all possible ones, and correctness of the SKE ensures that, if the same secret key is used, then the bijection offered by the decryption algorithm is the inverse of that offered by encryption (with overwhelming probability in the case of non-deterministic decryption). The size of a typical secret key in many real-world applications is 128 or 256 bit. This allows to index at most 2^{128} or 2^{256} unique bijections, which is generally much smaller than the number of possible bijections as the domain space gets larger. For this reason, it is generally impossible to ask the coverage of all possible bijections as a security property of SKEs. Instead, security is usually given in terms of *indistinguishability games*, with strength of the resulting notions depending on the additional power granted to the adversary. One of the most common security notions for SKEs is *indistinguishability under chosen plaintext attack* (or IND-CPA in short). In such game, the adversary's goal is to distinguish between the encryption of two messages of her choice, given additional access to an encryption oracle (for the same, unknown secret key used in the game). The scheme is called IND-CPA secure iff no efficient adversary $\mathcal{A}$ can successfully win with probability more than negligibly better than guessing at random. Notice that this, in particular, requires that the SKE must be *randomised*, i.e., encrypting twice the same plaintext with the same key will generally yield two different ciphertexts. We will denote encryption (resp., decryption) of a plaintext p (resp., ciphertext c) with a key k (and, optionally, a randomness r) as $\texttt{Encrypt}(p, k; r)$ (resp., $\texttt{Decrypt}(c, k; r)$). The randomness r is generally not needed for decryption, so we will omit it in that case.

If, in addition to privacy, *authenticity* of a message is also required, then SKEs are not enough, and *authenticated encryption (AE) schemes* must be used instead. An AE scheme works in a similar way to a SKE, but decryption of a

given ciphertext *fails* if the secret key used for decryption is not the same one used to encrypt the original plaintext. When this happens we write that the decryption procedure returns $\perp$. This allows to check that a ciphertext has not been altered or replaced by a malicious adversary, thereby granting authenticity and integrity of the message. A typical way to implement AE is to append a *message authentication code (MAC)* to a ciphertext. A MAC is a random-looking bitstring, for example computed through the *encrypt-then-MAC* procedure with a hash function on a combination of ciphertext and secret key. MACs are useful, among other properties, to check whether a provided key is the correct one to decrypt a ciphertext (without having to actually decrypt the ciphertext first).

Block ciphers. As a building block for SKEs, *block ciphers* are widely used. These are algorithms that typically offer two different interfaces, one for encryption and one for decryption. In encryption mode, they take as input a block of plaintext of fixed size B (the *block size*) and an encryption key of K bits (the *key size*) and return a block of ciphertext, also of size B. This mapping is undone in decryption mode, provided the same key is used. In other words, block ciphers implement a subset of size at most 2^K of the space of all possible $(2^B)!$ permutations (and their inverses) over B-bit strings. One of the most widely used block ciphers are those from the AES family [20], identified by AES-K (with a block size of 128 bits, and a keysize K of 128, 192, or 256 bits).

In order to turn a block cipher into a generic SKE, a *mode of operation* is required. This is a deterministic procedure that describes how to split input plaintexts or ciphertexts of arbitrary length into fixed size blocks, and iteratively applying the block cipher on these blocks. Typical modes of operation are ECB, CBC, and CTR [43]. Among these modes, CTR is usually preferred for its better characteristics. In order to achieve randomisation as a protection against known-plaintext attacks, many modes of operations also include an additional input called *inizialization vector (IV)*, typically a string of a fixed size, like 64, 96 or 128 bits, not necessarily secret but unpredictable and variable according to the message. Block ciphers can also be used to build AE, but with different modes of operations than the ones used for encryption only, such as GCM [29]. For a given block cipher $\mathcal{B}$ with keysize K and a given mode of operation $\mathcal{M}$, the resulting SKE is usually denoted as $\mathcal{B}$-$\mathcal{M}$-K, for example AES-CTR-192 or AES-GCM-256.

2.2 Full Disk Encryption

Full disk encryption (FDE) is a security technique that protects the content of a digital storage device (such as a hard drive or SSD) by using encryption 'on the fly'. This can include applications, user files, and even the OS itself. The primary purpose of FDE is to prevent unauthorized access to sensitive information in the event of device theft, loss, or unauthorized physical access.

FDE works by employing a cipher (usually a block cipher) to encrypt data at rest on the storage device. A user (or the device manufacturer) must first *initialise* the storage device, by providing an encryption key or passphrase to create

and write on the device a special metadata structure that represents an (initially empty) state encrypted with the provided key. In order to protect against space analysis attacks, a very first step before initialisation consists of completely over-writing the device with random noise. Then, every time the system is powered on or the device accessed, the user must provide the valid key or passphrase to decrypt and read/write data on the device. This key is not stored on the device itself and must be entered each time the device is prepared for use (*opening*), usually cached in a volatile and protected area of memory, and thereby erased when the device stops being used (*closing*) or the system is shut down. Except for the one-time initialisation phase (which can be quite slow depending on the device size), the encryption process is typically invisible to the user, as the OS handles the encryption and decryption of data as it is read from or written to the disk. Once the correct key or passphrase is provided, the user can interact with the device normally, without having to manually encrypt or decrypt files, as the OS only exposes a *virtualized* device that looks unencrypted to the user.

FDE can be implemented using hardware-based or software-based solutions. Hardware-based FDE is typically performed by a dedicated encryption chip, while software-based FDE is achieved using encryption software that runs at the operating system level. Implementation is usually done using standard block ciphers contructions and modes of operations like AES-CTR-256. Some examples of software-based FDE solutions include BitLocker for Windows, FileVault 2 for macOS, and LUKS for Linux [15, 30]. All these implementations have typically only negligible impact on performance compared to a non-FDE system, also thanks to the widespread presence of dedicated CPU instructions to speed up AES computation on personal devices such as smartphones and laptops.

Notice that if the whole OS is protected this way with a software-based solution, then there is a *bootstrapping problem*, because the OS itself cannot natively run while encrypted. This is addressed by either having a small, unencrypted bootloader which launches a minimal FDE application before the rest of the OS can start, or with a lower-level solution usually provided through hardware support such as a *Trusted Platform Module (TPM)*.

Cipher modes for FDE. For block ciphers used in disk encryption, the XTS mode of operation [22] is the most widely adopted because of its performance and security. It avoids the need of explicitly writing IVs for every block on disk by deriving these IVs *pseudodeterministically* from a global IV and sector-dependent metadata. Using CTR mode in a similar way would be a serious security mistake, unless care is taken in refreshing (and storing) IVs at every data write, which usually has an impact on performance and space usage. The latter approach, however, has a potential advantage: it gives the possibility of *re-randomising* blocks, i.e., changing the ciphertext without changing the underlying plaintext.

Caveats. It's important to note that FDE primarily protects data at rest, meaning it is most effective when the device is powered off or in a locked state. It does not provide protection against unauthorized access or data breaches while

the system is running and the encryption key or passphrase has been entered. In particular, FDE is arguably less effective on devices such as smartphones, which stay most of the time in an "on" state, and offers no protection against malware such as *keyloggers*, which might intercept the password entered by the user, or even access the unencrypted content directly. For comprehensive security, FDE should be combined with other security measures, such as strong authentication, secure boot processes, and proper access control policies.

2.3   Plausible Deniability

In this section, we present a formal game-based definition of PD security. It is worth noting that almost every paper in the field has given *its own* security definition, always slightly different from the others; valid attempts have been made to unify them into a single framework [10], but here we will follow the arguably more intuitive one given in [6], which is well suited for the block-layer scenario we work in. In this setting, a user employs a PD scheme to multiplex a single storage device into ℓ independent volumes $V_1, V_2, \ldots V_\ell$, each V_i being associated to a different password P_i. The PD scheme supports up to max volumes per device (so $1 \leq \ell \leq$ max); the value of max is publicly known. Both the volumes and the underlying device are block-addressable, meaning that the `read` and `write` operations they support have the granularity of a block. The semantics of a scheme Π is given as follows.

Definition 1 (PD Scheme). *Let $\ell \leq$ max, and $P_1, \ldots P_\ell$ user-provided passwords. A PD scheme Π is a tuple of algorithms:*

- *$\Pi.\mathtt{setup}(P_1, \ldots, P_\ell) \rightarrow \Sigma$: Initialises the disk to host ℓ volumes $V_1, \ldots V_\ell$, encrypted with passwords $P_1, \ldots P_\ell$; returns a device instance description Σ which encapsulates everything.*
- *$\Pi.\mathtt{read}(\Sigma, i, B) \rightarrow d$: Reads data d in block address B from volume V_i (we assume `read`s to not modify the instance)[4].*
- *$\Pi.\mathtt{write}(\Sigma, i, B, d) \rightarrow \Sigma'$: Writes data d into block address B of volume V_i, and updates the instance.*

The following correctness requirement applies: for any fixed block B and volume V_i, if $\Pi.\mathtt{write}(\Sigma, i, B, d)$ is the most recent write query which precedes a query $\Pi.\mathtt{read}(\Sigma', i, B)$, then $\Pi.\mathtt{read}(\Sigma', i, B) \rightarrow d$ (for simplicity, we consider operations atomic).

Access patterns. Let us define an *access* as the tuple $o = (\mathtt{op}, i, B, d)$, with $\mathtt{op} \in \{ \mathtt{read}, \mathtt{write} \}$ (if $\mathtt{op} = \mathtt{read}$, then d is the return value), and i being the index of the volume targeted by the access. Let us also define an *access pattern* as a (chronologically) ordered sequence of accesses $O = < o_1, \ldots, o_n >$. An empty access $o = \bot$ is also defined, which is simply ignored by the instance Σ.

---

[4] Note this is also the case for WoORAM-based constructions, but not necessarily for ORAM-based or other, arbitrary, PD schemes constructions.

PD security The security game for PD inherits some high-level concepts from the IND-CPA game (*ciphertext indistinguishability under chosen-plaintext attack*) for secret-key encryption. The adversary is a distinguisher, and is challenged with deducing whether she is interacting with a Σ encapsulating ℓ or $\ell - 1$ volumes. Also, she is allowed to choose the `read` or `write` operations to be executed, to capture the idea that indistinguishability must hold no matter the accesses performed on the volumes, hence including adversarial ones. A secret bit b within the game determines whether Σ_0 (containing ℓ volumes) or Σ_1 (containing $\ell - 1$ volumes) is first instantiated in the game and made to interact with the adversary. In both cases, we allow the adversary to choose the first $\ell - 1$ passwords[5], and her goal is to guess b.

Experiment 2 (PD game, generic) *For a PD scheme Π and an adversary $\mathcal{A}$ the* plausible deniability experiment $\mathsf{PD}(\Pi, \mathcal{A})$ *is defined as follows:*

1. *$\mathcal{A}$ chooses $\ell \leq \mathsf{max}$ and chooses $\ell - 1$ passwords $P_1, \ldots, P_{\ell-1}$.*
2. *A secret random bit $b \xleftarrow{\$} \{0, 1\}$ is drawn. If $b = 0$, then an additional secret high entropy password is sampled $P_\ell \xleftarrow{\$} \{0, 1\}^\lambda$, where λ is the security parameter[6].*
3. *The game creates $\ell - b$ volumes: $\Pi.\mathtt{setup}(P_1, \ldots, P_{\ell-b}) \to \Sigma_b$*
4. *$\mathcal{A}$ performs interactive rounds of queries. Every query works as follows:*
   - (a) *$\mathcal{A}$ chooses access patterns O_0 and O_1, where O_0 is, in the adversary's intentions, aimed at Σ_0 (thus potentially containing some operations on V_ℓ) and O_1 is aimed at Σ_1 (and so only contains operations on $V_1, \ldots, V_{\ell-1}$). She also chooses a bit v, signalling whether she wishes a snapshot of the disk at the end of this round.*
     These adversarial choices are subject to constraints, which we discuss in the next paragraph.
   - (b) *The game only executes O_b (on Σ_b, the only instance that was created in step 3). If requested, it sends the resulting disk snapshot D to $\mathcal{A}$.*
5. *At the end of all rounds, the adversary outputs a bit b'.*
6. *The game outputs 1 iff $b = b'$, 0 otherwise.*

Here we have omitted the constraints that the adversary is subject to in step 4a, when choosing the access patterns and when choosing the snapshot bit v; we will present them now. Without any such constraints, we will see that security would be impossible to achieve; also, the exact set of constraints will modulate the induced threat model.

---

[5] This represents the most unfavourable situation for the user, as we consider these passwords compromised anyway.

[6] We abuse notation by representing a password as a binary string, but w.l.o.g. it is equivalent to the case of a user-chosen password with λ bits of entropy. We assume that the user will choose a high entropy password at least for the hidden volume.

Constraints on the bit v. This constraint governs the snapshotting capabilities of the adversary, thus the adversary power. We only define two extreme cases:

1. Arbitrary - No constraint: the adversary is allowed to set $v = 1$ in all of the interactive rounds. This is the strongest form of multi-snapshot security, as the adversary can obtain a snapshot any time she desires.
2. One-Time. The adversary is single-snapshot, i.e. can only set $v = 1$ for one of the interactive rounds.

Constraints on the access patterns. These constraints define the adversary goal, by specifying which two exact situations she has to distinguish between: if a PD scheme is secure (i.e., the adversary cannot distinguish) under the game enforcing such a constraint, the implication is that a user, having performed some access pattern O_0 including some operations on V_ℓ, can plausibly claim to instead have executed a corresponding O_1, which only accesses the volumes $V_1, \ldots, V_{\ell-1}$ (whose passwords have already been surrendered).

Discussion on the constraints. Let us first clarify that some constraint is necessary in order to have any hope in the PD game against the adversary. Otherwise, the adversary could submit O_0 and O_1 containing completely different (logical) `write` accesses to the decoy volumes $V_1, \ldots, V_{\ell-1}$, and there would be no way of making the two outcomes indistinguishable, since the adversary holds the passwords to those volumes, so it could trivially verify which of the two patterns was executed. This suggests the need for a minimal rule, stating that, whether O_0 or O_1 is executed, the resulting logical contents of the decoy volumes $V_1, \ldots, V_{\ell-1}$ must be the same. From the user's perspective, this basic requirement means that we do not try to disguise the `write`s to decoy volumes as something else, both because we do not need to, and because there would be no way of doing it even if we wanted to.

Furthermore, we notice that many PD solutions (including those based on WoORAMs) treat `write` requests in a completely different way than `read` requests, i.e. a `write` request could *trigger allocation or reshuffling* of a certain volume sector. This might happen without breaking the minimal rule described above: if an adversary first `read`s at a previously unallocated position B, obtaining data d, and then `write`s the same data d at the same location B, this might cause a change of state in the instance without changing its logical content (as mandated by the minimal rule above). This, in turn, would enable an attack where the adversary merely checks whether this state change happened or not. However, we must consider this attack trivial: detecting this kind of state change on a decoy volume would actually not compromise security. Therefore, in order to capture this concept, we also demand that the set of all blocks that are touched by `write` requests be the same for both O_0 and O_1, for all volumes $V_1, \ldots V_{\ell-1}$. We stress that this extra constraint is also very minimal and does not weaken the security guarantees offered in practice; in fact, most schemes demand even stricter constraints, for example by demanding that O_0 and O_1 are of the same length, which we do not require here.

Single-snapshot security In the single-snapshot case, the security game can be simplified. More formally, if the "One-Time" constraint is enforced on bit v, then Experiment 2 is equivalent to the same game where the number of interactive rounds is set to 1 instead of being chosen by the adversary. This is because the access patterns submitted *before* obtaining the (only) disk snapshot might as well be all concatenated into one, as they cannot be chosen adaptively; moreover, all the access patterns submitted *after* obtaining the snapshot can be disregarded altogether, as their effect will not be detectable by the adversary.

Security in this one-round game can then be rephrased as follows:

Definition 3 (Single-Snapshot Security) *A PD scheme Π is* single-snapshot *(SS)-secure iff for any PPT adversary $\mathcal{A}$ which chooses $\ell \leq$ max, passwords $P_1, \ldots, P_{\ell-1}$, and access patterns O_0 and O_1 subject to the constraints outlined in Section 2.3, it holds:*

$$\left| \Pr\left[\mathcal{A}\left(D_0\right) \to 1\right] - \Pr\left[\mathcal{A}\left(D_1\right) \to 1\right] \right| \leq \mathsf{negl}, \tag{1}$$

where:

- D_0 *is the disk snapshot resulting from the application of O_0 to Σ_0, where $\Sigma_0 \leftarrow \Pi.\mathtt{setup}(P_1, \ldots, P_\ell)$ and $P_\ell \xleftarrow{\$} \{0,1\}^\lambda$.*
- D_1 *is the disk snapshot resulting from the application of O_1 to Σ_1, where $\Sigma_1 \leftarrow \Pi.\mathtt{setup}(P_1, \ldots, P_{\ell-1})$.*

3 TrueCrypt and VeraCrypt

Given its relevance in the context of comparison to Shufflecake, we want to discuss here TrueCrypt [46], which was the first disk encryption software (now discontinued) to offer PD capabilities. It was developed around the early 2000s, before BitLocker [30] and LUKS [15] became the default standards for disk encryption on Windows and Linux, respectively. Its development has come to a sudden halt in 2014, but a backward-compatible successor exists (VeraCrypt [49]) that has kept most of the design principles, and improved on some minor aspects (like a stronger key derivation). For our purpose in this work, we will focus on TrueCrypt only, as all our considerations similarly apply to VeraCrypt.

TrueCrypt can operate in two main modes: with "standard" (sometimes called "outer") encrypted volumes, or with "hidden" volumes. In the former case, it is functionally similar to other FDE solutions like LUKS (but with random-looking encrypted headers). In the latter case, a hidden volume is embedded in the unused empty space left by the content of the decoy standard volume. Plausible deniability is given by the fact that disk headers and content are indistinguishable from random, which makes it hard to distinguish between the two cases without the correct passwords.

3.1 Design

TrueCrypt (like Shufflecake and many other existing PD schemes) works as a *stacking driver*, that is, a device driver operating on top of another device driver. It exposes a *logical* (virtual) storage space to the upper layer, which directs *logical* `TcRead` and `TcWrite` requests to it; the stacking driver then executes its algorithm to map these requests to *physical* block `bRead` and `bWrite` requests to the underlying device driver, which manages the *physical* storage space. Here the distinction between *logical* and *physical* is the distinction between before and after the translation operated by the stacking driver, regardless of whether the *physical* storage space is also a virtual device.

The first initialisation operation performed by TrueCrypt when creating new volumes within a device is to fill the disk with random bytes, which is also the case for regular disk encryption tools including LUKS, as we already discussed. The first part of the disk contains the fixed-size encrypted header of the standard volume, and an equal-size empty slot filled with random bytes (remaining from the initialisation procedure). Then comes the actual encrypted data section of the standard volume, which includes some empty space, also filled with random bytes (coming from the initialisation procedure).
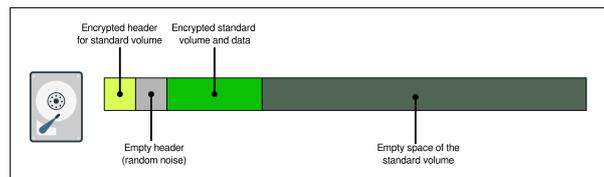
Fig. 1: TrueCrypt's disk layout (standard volume).

TrueCrypt optionally allows to "embed" a hidden volume in the (contiguous) empty space left by the standard volume: this is the mechanism providing plausible deniability. Its encrypted header then fits in the empty slot left after the header of the standard volume. The standard volume and the hidden volume are encrypted with two different passwords.

Fig. 2: TrueCrypt's disk layout (standard/decoy and hidden volumes).

One big limitation of TrueCrypt is that it can only support a hidden volume if the outer volume is formatted as a FAT filesystem. This is because we need the empty s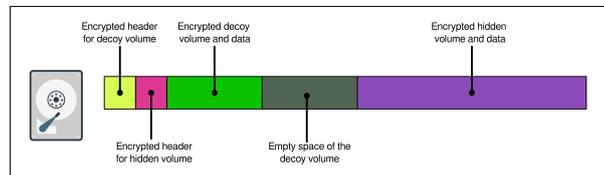pace left by the decoy volume to be contiguous. Most modern filesystems (like ext4 and NTFS) "jump back and forth" on the disk as they are written with data, leaving lots of empty blocks, or "holes", in the middle. Instead, the FAT filesystem is special, in that it grows incrementally and, up to the physiological holes created by deleted files, occupies all the space up until its last utilised block. This way, one can have the hidden volume start at a certain offset, after the end of the decoy volume, and then follow data allocation linearly. TrueCrypt automatically computes a convenient starting position for the hidden volume (leaving some leeway buffer after the end of the standard volume), and places it among the metadata of the hidden volume's header. Whereas the hidden volume is assigned a logical size that follows the physical space actually allocated to it, the standard volume is *not* resized, and keeps logically mapping onto the whole disk. This is crucial in order to not defy deniability: if we resized the standard volume, this information (which leaks the existence of a hidden volume) would be written in the metadata of its file system, which is inspected by the adversary.

Another big limitation of this approach is that, from the moment that a hidden volume is embedded into a standard volume, the latter will be very limited in the possibility of content growth: as the hidden volume lives in the empty space of the standard volume, this seemingly empty space can never shrink (except for the leeway buffer), or the hidden volume will be corrupted.

3.2   Operational Model

Using TrueCrypt comes with restrictions on what the user can do in order to preserve data integrity and plausible deniability.

As already mentioned, once the hidden volume is created, its starting position is final, and it "freezes" the end of the decoy standard volume, limiting the maximum size that it will ever be able to attain. Since the standard volume cannot be resized to accommodate for the hidden volume, and instead keeps mapping onto the whole disk, it is up to the user to not let it grow too much and overwrite the hidden volume. This is achieved both by frequent de-fragmentation of the FAT filesystem within the standard volume, and by actually not writing too much data into it. If one wants to be absolutely safe about data corruption within the hidden volume, the recommendation for the user is to never unlock only the standard volume for daily use (except if under coercion), but to always unlock either 1) the hidden volume only, or 2) both hidden and standard volumes, but keeping the latter in read-only mode.

The standard volume must contain "decoy" data, that will reasonably convince the adversary that it is the only volume existing on the disk. Clearly, the user should only surrender the password of the standard volume to the adversary when under coercion. This, in turn, opens the door for corruption of the hidden volume. Eliminating completely such corruption risk is unavoidable, in TrueCrypt this can can only be mitigated by frequent backups.

3.3 Security

Here, we analyse TrueCrypt's security under different threat models.

Pseudorandomness. Unlike other FDE solutions such as LUKS and Bitlocker, TrueCrypt-formatted devices do not contain any cleartext header. This means that a TrueCrypt-formatted device is indistinguishable, when at rest, from a device completely filled with random noise. This feature is desirable in certain scenarios, for example it is more straightforward and less risk-prone if one wants to embed a TrueCrypt container file within another medium using steganography, but represents a tradeoff against ease of integration with other parts of the system, which is the approach preferred by all-purpose FDE solutions such as LUKS and Bitlocker. Anyway, it must be stressed that this feature is not relevant per se for PD, because in the PD scenario the adversary is always provided with at least *one* decryption key.

Single-snapshot security. In TrueCrypt, once the user surrenders the password of the decoy volume and lets the adversary decrypt it, the only part of the disk contents that remains to be "interpreted" by the adversary is the non-decrypted space after the end of the decoy FAT file system. However, whether this space is actually empty or whether it contains a hidden volume, it will be filled with random bytes that are not readable with the decoy password alone. Therefore, even if a hidden volume is present, the user can *plausibly claim* that the remaining space is empty and filled with random bytes: the adversary has no way to disprove this claim, *or even to question its likelihood*, based on the observed disk content.

Multi-snapshot security. It is easy to see why TrueCrypt is insecure in the multi-snapshot threat model: what happens if the adversary obtains two snapshots of the disk at two different points in time, and the user has made changes to the hidden volume in the meantime? By comparing the two snapshots, the adversary clearly sees that some of the allegedly "empty" blocks have changed, which immediately reveals that a second volume exists, because TrueCrypt never re-randomises the actually-free space.

TrueCrypt's hidden OS. Latest versions of TrueCrypt offer a solution to the OS' tendency to leak the existence of hidden partitions through a "hidden OS" feature. A decoy OS is installed within a standard volume, while a separate OS is installed within the hidden volume of another partition. In order to decide which OS is booted according to the provided password, the computer's bootloader is replaced by the ad-hoc *TrueCrypt bootloader*, which will first try to boot the decoy OS with the user-provided password and then, if unsuccessful, will try to boot the hidden one. Since the decoy OS itself never sees the hidden partition, there is no possibility for it to even be aware of the existence of hidden data. Notice, however, that this feature is only available for Windows.

3.4 Other Limitations

Here we discuss other problematic aspects of TrueCrypt. It is to be noted that some of such considerations apply today, but were less relevant in the early 2000s, when TrueCrypt was first conceived.

First of all, as discussed, the standard volume must be formatted as a FAT filesystem if a hidden volume is desired. However, FAT is now outdated: it used to be very widespread, but today there is little plausible reason to use it anymore. Therefore, the mere fact that we are using FAT raises a red flag to the adversary.

Another problem arises from the fact that the user must avoid or limit the use of the decoy partition in order to not corrupt the hidden one. Yet, decoy volume(s) must "look legitimate": it must be plausible by looking at their content that they are the only ones. In particular, they must be reasonably up to date: if we only ever work on the hidden volume, and completely forget about the decoy, an adversary unlocking the decoy would become very suspicious seeing that the most recent updates are months if not years old.

In general, it is not within the scope of PD schemes to hide *themselves*, i.e., hide the very fact that that scheme is being used. We must assume that this fact is known to the adversary, who might, e.g., by searching the user's laptop, discover a TrueCrypt installation. Since a locked TrueCrypt volume is indistinguishable from random data, when asked for the first password, we could in theory even claim that the disk is not formatted with TrueCrypt at all, but is instead the result of a secure wiping procedure, or even that it is a pool of random data coming from other sources. However, a real-world adversary will arguably be unconvinced given the knowledge that a system like TrueCrypt is in use. So, a recommended course of action is to separate the TrueCrypt-supporting system (e.g., a laptop) and the encrypted media (e.g., a USB stick). This might be cumbersome for most use cases.

Another big limitation is the fact that only *one* hidden volume within each standard volume is supported. This is a problem, because the adversary might reasonably suspect that TrueCrypt is in use *exactly* to hide something through its PD feature: if we only meant to encrypt a volume, we would reasonably use a more widespread and supported solution like LUKS or BitLocker. A user could claim that they prefer using independent, niche open-source software to secure their data, and it would be relatively credible, but the safest course of action when designing a PD scheme is assuming that the adversary might not believe this claim, and ask for a second password.

The short answer to this problem is: a robust TrueCrypt-like PD solution should allow *simultaneous access* to *more* than two layers of volumes. That way, we can create a series of volumes with increasingly "private" contents (that could well be all decoys), so as to reveal more than one password to the adversary and convince them, based on the resulting decrypted contents, that we have effectively given up what we were hiding, while in fact we are still holding the password to one more top-secret volume whose existence they have no more reason to suspect.

4 The Shufflecake Scheme

In this section we present our Shufflecake scheme, we explain its way of operation, and we provide a security analysis.

4.1 Design

By *device*, we mean the underlying disk, which exposes a *physical* storage space. Instead, *volumes* are the *logical* storage units that map onto a device. The name 'Shufflecake' stems from the analogy of mixing up *slices* of a cake (the device) in order to provide many stacking *layers* of privacy (the volumes). Conceptually, Shufflecake's operation consists of four functionalities:

1. *Initialize* a device: this is done only once, when a new device is first prepared for use in Shufflecake. It consists in overwriting the device with random data, asking the user to provide the number ℓ of desired volumes and related passwords, and creating an encrypted header with metadata using this information. In our implementation, this functionality is provided by the `init` command.
2. *Instantiate* a device: this is the preliminary stage of preparing a Shufflecake-initialized device for use. It consists of reading the user's provided password, trying to decrypt the device's header metadata with the derived key, and, if successful, recover information on the available volumes provided. In our implementation this functionality is invisible to the user (it gets executed together with `open`), so we don't provide an associated command.
3. *Open* a volume: using the correctly derived volume key, volume-specific metadata is read from the relevant header section. This metadata is used to create a logical device which is presented to the user, and the user's OS can issue `SflcRead` and `SflcWrite` requests to this logical volume. In our implementation, this functionality is provided by the `open` command.
4. *Close* a previously instantiated device: ephemeral state changes, if present, are written (encrypted) to disk, and then *all* the open volumes provided by that device are removed from the user's view. In our implementation, this functionality is provided by the `close` command.

At its core, Shufflecake is a block indirection layer on top of an encryption layer. Our indirection layer realizes a mechanism which is already a strong improvement over TrueCrypt, since it fixes two of its crucial limitations: it allows multiple volumes, and it is filesystem-independent. Data decryption keys for every volume are derived by a password and other header-derived randomness. Furthermore, the decrypted payload of the header of volume $\ell > 1$ also contains a copy of the header decryption key for volume $\ell - 1$. This allows to recursively open all volumes present in a device by using a single password, which in turn improves security and user experience, as we will see.

Disk Layout. The device's physical storage space is statically divided into a *header section* and a *data section*. The header section is found at the beginning of the disk, and it is composed of a fixed-size *device master block (DMB)*, and max equal-sized *volume headers* (each of them comprised of a *volume master block (VMB)* and a *position map*), irrespective of how many volumes there are effectively. This mild waste of space is necessary in order to to prevent the adversary from trivially deducing the number of volumes by the size of the device header (which might be possible by analysing the data allocation pattern even when not all volumes are opened). Let us analyse all these sections, starting first with the data.
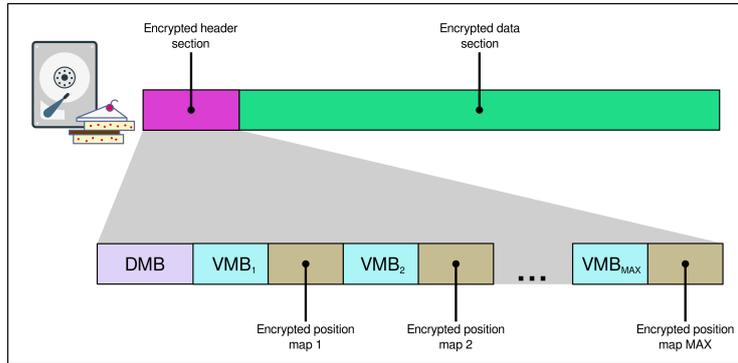

Fig. 3: Shufflecake's disk layout overview.

Data section. Instead of mandating that the volumes be physically adjacent on-disk, like in TrueCrypt, we randomly interleave them as encrypted (but not authenticated) fixed-size *slices*, where every slice belongs to one volume and contains a certain number of blocks. Metadata in the i-th volume header allows to reconstruct the logical content of V_i by mapping the corresponding slices as a (virtual) contiguous space.

We distinguish between *logical slices*, of size S_L blocks, and *physical slices*, of size $S_P = S_L + \Delta_S$, where Δ_S blocks are used to store the encryption IVs for that slice. Logical slices store the data of their respective volumes, while physical slices are used to reserve space on the disk to allocate the encrypted logical slice. A physical slice can either be unallocated (i.e., unclaimed by any volume), or mapped to a single logical slice belonging to some volume V_i. In the latter case, the mapping from the *logical slice index (LSI)* σ of volume V_i to the device's *physical slice index (PSI)* Ψ is given by a function $\texttt{GetSliceMap} : (i, \sigma) \mapsto \Psi$ which is computed by simply looking up a *position map* (basically, a per-volume array). I/O operations to a *logical block address B* of volume V_i are performed through the two interfaces $\texttt{SflcRead}$ and $\texttt{SflcWrite}$. We will describe later these two interfaces, as well as the structure of the position map.
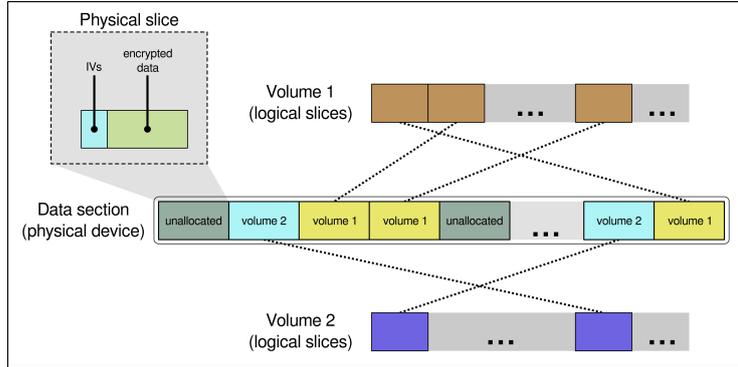
Fig. 4: Shufflecake's data section layout and slicing.

Device master block (DMB) The DMB encapsulates all password-related data. It begins with one single KDF salt, shared for all volumes: this salt, combined with a volume V_i's password through a KDF, yields the volume's *key-encryption-key (*KEK_i*)*. Notice that we derive every KEK_i by using just a single global salt for all max volumes, otherwise we would incur in up to max expensive different key derivations every time we instantiate a device[7]. This does not hamper security, as password hashes are never stored on disk, but only used to generate the KEK which is in turn used as a decryption key.

Then come max DMB *cells*, each being an authenticated ciphertext (together with the corresponding IV), encrypted with the respective volume's KEK: the plaintext is itself another cryptographic key, encrypting the volume's VMB (the *volume master key* VMK_i). This key decoupling allows us to possibly change a volume's password without having to re-encrypt all its content with a different key. For granularity and consistency, the overall size of the DMB is fixed to be exactly one block. The rest of the DMB, and the unused DMB cells, contain random noise.
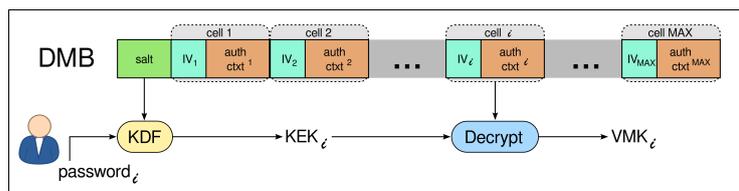

Fig. 5: Shufflecake's Device Master Block (DMB).

---

[7] This limitation can actually be avoided by using some key derivation tricks, like *'re-salting'*, i.e. using the output of the KDF in combination with a (fast) hash using a per-header salt. We leave this approach for future consideration.

Volume master blocks (VMBs). The i-th volume header is composed of a Volume Master Block (VMB), followed by the encrypted (but not authenticated) position map of volume V_i. We discuss the position map in the next paragraph. The VMB is a single block containing a (non-authenticated) ciphertext, encrypted with the volume master block's key VMK_i, and the associated IV. The underlying plaintext is composed of the following fields:

– The volume's *volume-encryption key (*VEK_i*)*, used for encrypting the actual data section and the position map.
– The previous volume's VMB key VMK_{i-1} (or a random value if $i = 1$).
– The number of slices $\mathtt{NumSlices}$ contained in the device.
– The remaining space up to filling the block size is left random, but can be optionally used to embed additional volume-related metadata if needed.

The device-specific value $\mathtt{NumSlices}$ defines and fixes the size of the position maps, even in the case that the device is resized; it is replicated across all volume headers in order to be decryptable with any provided password. The presence of VMK_{i-1} is what allows us to impose a hierarchy on the otherwise-independent volumes (they are all treated equally in the data section). This way, once we open volume V_i, we can iteratively walk the backwards-linked list induced by this field, and also open volumes V_{i-1} through V_1. While this approach compromises deniability for a volume V_i "in the middle", once the password to V_j, $j > i$ has been provided, it does not harm deniability as defined by the security game: the volume we want to hide is the last one, V_ℓ, not the middle ones. The usefulness of this approach is discussed in Section 4.2.
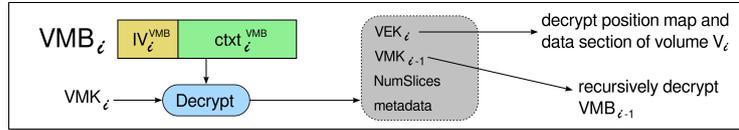

Fig. 6: Shufflecake's Volume Master Block (VMB).

Slice maps. Slice maps are arrays $\mathtt{SliceMap}_i$ of $\mathtt{NumSlices}$ elements, where every element is a PSI: the index of each element is the LSI mapping to that PSI. Each volume's slice map is decrypted and loaded in memory when the volume is *instantiated*; these maps are kept entirely in RAM while the volumes are "live", and persisted on-disk (encrypted, together with their fresh IVs) when the volumes are closed. Slice maps are stored after the VMBs as equal-size ciphertexts, large enough to address all the possible physical slices. The RAM and disk space requirements are modest: if the underlying device is N blocks large, the size of a slice map is just $O\left(\frac{N}{S_P} \log \frac{N}{S_P}\right)$, because there are at most $\frac{N}{S_P}$ physical slices, each requiring $O\left(\log \frac{N}{S_P}\right)$ bits to be indexed. This is in turn due to the choice of addressing the storage space at the slice granularity, instead of the block granularity, which would have entailed a position map of $O(N \log N)$ bits.

Operation. We now describe more in detail how Shufflecake operates and explain the rationale behind certain design choices. We start by explaining how we use encryption to secure data on-disk. Then we look at the indirection layer between physical data allocation on disk and data on the logical volumes, i.e., the mechanism which creates the correspondence between physical and logical slices. Finally, we explain how this correspondence is updated when new logical slices are needed as data is written on the logical volumes.

Cryptographic layer. As in many disk encryption solutions, we encrypt with the block granularity, meaning that blocks are the unit of both I/O requests and encryption/decryption; in other words, one IV encrypts one block. Many disk encryption schemes generate these IVs pseudo-deterministically from some public context information and the volume's secret key (e.g., this is what happens in the XTS mode of operation) in order to save the space and the I/O overhead needed to store and retrieve them. Having such a deterministic procedure for generating IVs on the fly is enough for the threat model covered by FDE and single-snapshot secure PD solutions like TrueCrypt.

In our case, however, we stick to explicitly random IVs because we want to keep the future possibility to extend Shufflecake with some degree of multi-snapshot security, requiring us to re-encrypt some blocks with a different IV while leaving their content unchanged. For this reason, we use CTR mode instead, and the IV of a block is refreshed at each `write` for that block, to avoid IV-reuse attacks. This means that all these IVs are stored on-disk.

This strategy introduces a potential performance issue: a naive implementation would translate each logical `SflcWrite` to a physical `bWrite` of the corresponding physical data block, plus an additional `bRead`-update-`bWrite` of the *whole* corresponding IV block. This would be very wasteful in terms of I/O overhead, because we only need to update one IV (e.g., 16 bytes for AES-CTR), but we are forced to load and store whole blocks (typically 4096 bytes).

We avoid this problem by caching IV blocks in RAM, in an LRU cache of predefined depth (e.g., 1024 entries). For the performance reasons just discussed, this cache is not write-through. This way, we coalesce possibly many updates of the same IV block (triggered by many logical `SflcWrite`s to the same data block, or by logical `SflcWrite`s to many blocks within the same slice) into just one physical `bWrite`, thereby lowering the I/O overhead.

For each physical slice, we pack the IVs for the blocks contained therein into Δ_S physical blocks at the beginning of the slice. There is a simple static correspondence between a physical block and the on-disk location of its IV: the $(m - \Delta_S)$-th block within a physical slice is encrypted by the m-th IV within the initial Δ_S blocks. Hence, we assume the existence of functions `LoadIV` and `SampleAndStoreIV` (with self-explanatory behaviour) which take as input a physical address B_{phys} and return the corresponding IV. Analogously, we consider the functions $\mathtt{Encrypt}(\mathsf{ptxt}, \mathsf{key}; \mathsf{IV})$ and $\mathtt{Decrypt}(\mathsf{ctxt}, \mathsf{key}; \mathsf{IV})$ as acting on blocks according to the implied mode of operation.

Indirection layer. Consider a read or write operation to a logical block address B for volume V_i. There are three possible cases:

1. The requested operation (read or write) for volume i happens on a block whose logical address B was previously allocated. In this case we need to map efficiently B to the corresponding physical address B_{phys}.
2. It is a `SflcRead` operation for a B which falls within a slice that was not allocated before. We need to specify the behavior in this case.
3. It is a `SflcWrite` operation for a B which falls within a slice that was not allocated before. We need to define how to allocate a new physical slice.

Notice that the offset of a block *within* a slice is left unchanged by the position map. So, we can easily find the LSI σ to which B belongs as simply $\sigma := \lfloor B/S_L \rfloor$. Then we need to check whether `GetSliceMap`(i, σ) is defined or not. Luckily, as we have seen above, position maps are not too large. So, when decrypting the position maps during device instantiation, we can store in memory a full view of these position maps for each volume as arrays of PSIs indexed by the related LSI, and define a special return symbol $\bot$ for those LSIs which have not yet been assigned.

Then let's analyse the three cases above one by one. In the first case, we just need to consider the PSI Ψ just obtained and add a correct offset. This will give us the physical address B_{phys} where we will find the data to decrypt. From what we have just discussed, this can be done as $B_{\mathsf{phys}} := \Psi \cdot S_P + \Delta_S + (B \mod S_L)$.

In the second case, instead, `SflcRead` should return a default, non-error value, e.g., 0. Not throwing an error in this case is necessary for the semantic of a volume: although it has never been written before, the block logically exists, it is within the logical boundary of the volume, so it would be incorrect to return an error. Notice how we never allocate new slices on `SflcRead` requests: as we will see in Section 4.3, this is necessary for security, and prevents logical read operations to leave a trace on the disk.

In the third and last case we need a way to allocate a new slice for volume V_i at a position consistent with B. We do this through a function `CreateSliceMap` which returns a PSI uniformly at random among those not yet mapped. We will see in the next paragraph how to implement this function.

Slice allocation. Slice mappings are created *lazily*, only when the first request for a block belonging to a new, yet-unmapped slice arrives. At this point, we create the mapping for this slice by sampling a physical slice uniformly at random *among the free ones*: this guarantees that no conflicts arise between volumes, and their slices end up randomly interleaved on the disk. To make this lazy sampling possible, we need to implement efficiently a function `NewSlice`: given as input a volume's identifier V_i and an LSI σ, it returns a corresponding PSI Ψ for the new slice. There are different ways to implement this, here we give a reference description using a permutation of the array representing the slices.

Concretely, we keep an in-memory, per-device array `prmslices` of PSIs of size $\lfloor \frac{N}{S_P} \rfloor$ (the maximum possible number of PSIs), and a bit-array of the same size

Algorithm 1 SflcRead(V_i, B)

1: $\sigma := \lfloor B/S_L \rfloor$
2: $\Psi \leftarrow$ GetPosMap(V_i, σ)
3: **if** $\Psi = \perp$
4:     **return** 0
5: $B_{\text{phys}} := \Psi \cdot S_P + \Delta_S + (B \mod S_L)$
6: IV $\leftarrow$ LoadIV(B_{phys})
7: $d \leftarrow$ bRead(B_{phys})
8: **return** Decrypt($d, V_i.K$; IV)

Algorithm 2 SflcWrite(V_i, B, d)

1: $\sigma := \lfloor B/S_L \rfloor$
2: $\Psi \leftarrow$ GetPosMap(V_i, σ)
3: **if** $\Psi = \perp$
4:     $\Psi \leftarrow$ NewSlice(V_i, σ)
5:     **if** $\Psi = \perp$ **return** Error
6: $B_{\text{phys}} := \Psi \cdot S_P + \Delta_S + (B \mod S_L)$
7: IV $\leftarrow$ SampleAndStoreIV(B_{phys})
8: $c :=$ Encrypt($d, V_i.K$; IV)
9: **return** bWrite(B_{phys}, c)

bfld, an *"occupation bitfield"* telling us which physical slices are occupied (the PSIs are the indexes of the array). Initially, at device instantiation, prmslices is initialized as prmslices[i] := i and then permuted using the efficient Fisher-Yates algorithm [13]. The bitfield bfld is initialized with all free elements. When a volume is opened, and we discover the physical slices it maps to, we mark them as occupied in bfld. The slice allocation algorithm then simply works by repeatedly taking the next element from the pre-shuffled array of PSIs until the bitfield tells us it is free (at which point we take it and mark it as occupied). A stateful occupation counter octr is kept to facilitate this task, initially set to 1, and then increased up to the first element of prmslices marked as free (i.e., the first octr elements of prmslices are guaranteed to be occupied in bfld). This way, when a new slice is required, it is immediate to advance to the next available one. We also need to update V_i's position map PosMap$_i$ before returning (this is the only change that will eventually persist, encrypted, on disk).

The permutation of indexes changes at every device instantiation, so that the mapping is not static. The size of these in-memory supporting data structures (array and bitfield) is $O\left(\frac{N}{S_P} \log \frac{N}{S_P}\right)$, as usual. The lazy allocation technique is also what allows us to *overcommit* the total physical storage space: we can have the sum of the sizes of the logical volumes exceed the total physical storage space, as long as the sum of "actually used" spaces does not. However, it might be useful to intentionally limit this overcommitment (for the opened volumes) to decrease the risk of I/O errors and improve user experience. This can be done using the metadata field in the VMB ciphertexts, as explained in Section 6.5.

Algorithm 3 NewSlice(V_i, σ)

1: **while** bfld[prmslices[octr]] = occupied **do:** octr := octr + 1
2: $\Psi_{new} \leftarrow$ prmslices[octr]
3: bfld[prmslices[octr]] := occupied
4: PosMap$_i$[σ] := Ψ_{new}
5: **return** Ψ_{new}

Volume operations. In principle, nothing in the scheme inherently prevents us from creating, opening, and closing volumes freely and independently, at any time. However, for real-world operations, we force volumes to be opened in a hierarchical way, by only providing *one* password (for the most secret volume).

To create a new volume it is needed: the index i of the new volume V_i, the chosen password, and the VMB key of volume V_{i-1} (if $i > 1$). This way, one can format the header by generating the relevant keys, filling the VMK_{i-1} field, and initialising the slice map as empty. No operation is needed on the data section.

To open a volume, only its password is needed in order to decrypt the header, which then allows to load its slice map and to decrypt its slices. Finding the right header for a provided password is done simply by trying every one of them, until the authenticated ciphertext in the related DMB cell decrypts correctly.

Closing a volume mainly modifies the state of the Shufflecake instance in RAM, by removing the relevant volume information (and securely erasing its key). The only required disk operations are the ones needed to persist some possibly-unsynchronised data.

No specific operation is needed to destroy a volume, i.e. to remove it from the disk. It is enough to just forget the password, or to overwrite the header with random bytes: by the PD guarantees, there is no way to then even prove that there was a volume in that slot, let alone to decrypt it.

4.2 Operational Model

In this section, we define the operational model of Shufflecake, to provide a safe *mode of use* allowing the user to retain both plausible deniability and data integrity. Besides some general constraints, we specify what the user has to do in ordinary working conditions, and how instead they must behave when confronted with the adversary.

Risk of data corruption. A simple observation shows how a legitimate-looking usage mode of Shufflecake actually entails a high risk of data corruption. If we do not open all ℓ existing volumes, and instead only open the ones we plan to use, we do not load all the slice maps in RAM, which leads to an incorrect reconstruction of the complete device's bitfield `bfld` of free physical slices. The physical slices belonging to the still-closed volumes will be counted as free, and will therefore possibly be allocated to the open volumes during data write, which would then overwrite their content. This can only be avoided with certainty by always opening *every* volume, regardless of which ones we are going to use: if the password to a volume is not provided, Shufflecake has no way of detecting its existence. It then follows from the overcommitment of the physical storage space that we risk re-using its physical slices for some other volume.

However, mitigation does need to be perfect. It could be possible to reduce the risk of corruption when not opening all volumes by using some form of *error-correction* on the unopened volumes (hence sacrificing some space), and then trying to recover the volume if corruption happens. We discuss this in Section 4.2.

General constraints. The first thing to do when initialising a device with Shufflecake is to fill it completely with random bytes. Though long and tedious, this operation is crucial even for single-snapshot security, as we will see in Section 4.3, just like it is for TrueCrypt. The most sensitive data should be placed in a volume of sufficiently high order. We cannot, of course, give precise indications of the form "use at least 3 volumes", or "6 volumes should be safe enough", because, by Kerchoff's principle, we assume that the adversary knows about Shufflecake, and in particular reads this document. The volumes of lower order, that will be disclosed to the adversary, should be filled with "mildly incriminating" data, so as to convince the attacker that one had a plausible reason to hide them. We do not specify more precisely what kind of content would be suited to this end, partly for the same reasons as before (the adversary would immediately flag it as decoy content, and ask for more passwords), partly because it heavily depends on the context and on who the adversary concretely is. The decoy volumes must also be otherwise "credible", in particular they must be formatted with realistic file systems, and they must be reasonably up to date. Periodic updates can be delegated to a background daemon or offloaded to the user.

Home alone. In normal operating conditions, when not confronted with the adversary, the recommended course of action for the user is to unlock all the volumes present on the device, in order to prevent data corruption as explained before. The reason behind the design choice of chaining the volumes into a linked list to help the user in that regard: this way, the user is able to open all volumes by just providing the password to V_ℓ. In our implementation, this is actually the mandated semantic of the `open` operation: the user only provides the password of the *last* volume they want to open, the previous ones are automatically opened. Other implementations are of course free to ignore the VMK_{i-1} field in the volume header, and give more flexibility to the user, if aware of the risks entailed.

Under interrogation. When questioned by the adversary and forced to reveal passwords, the user must obviously not surrender more than $\ell - 1$ of them (otherwise there would be nothing left to protect). Although irrelevant for the cryptographic security of the scheme, we stress that, in order for the user's lie to be credible, they must only reveal the decoy passwords under a certain amount of pressure, or after some time has passed. Notice that a responsible and safe use of Shufflecake puts on the user the burden of being able recall quickly and reliably these decoy passwords, even under distress. This might be hard to get right given that the recommended course of action for the user in daily use is to only open the most hidden volume. It is up to the user to define the maximum number of volumes ℓ that makes them comfortable in this task. Shufflecake implementations might include additional features to aid the user in this sense, for example a function to check the password of a decoy volume without actually opening it, or even a puzzle which, with some random probability, prompts the user to also insert the password for a decoy volume when opening a hidden one.

Safeword. As we previously discussed, one big operational difference between Shufflecake (but also other solutions like, e.g., HiVE) and TrueCrypt is that with Shufflecake "the adversary does not know when she can stop questioning you", because there is no way to prove that a given password unlocks all the existing volumes on a given device (unless it's the password unlocking the max-th volume). In TrueCrypt, instead, there is either just a regular volume, or a decoy and a hidden volume. This distinction might be important in those scenarios where a user wants to avoid the possibility of looking uncooperative to a certain adversary. In such scenarios, the user might want to have the choice of surrendering all the volumes, and a method to convince the adversary that no other volumes exist, possibly using an additional "full disclosure password" that we will call *safeword*.

One simple way to implement this, even in Shufflecake, is to actually create all max volumes when initializing a device. Clearly, remembering max passwords would be quite cumbersome for the user, so the solution is to only remember $\ell+1$ passwords instead: those for the ℓ volumes that are actually desired, and the extra one for the last max-th volume, which is going to be the safeword. In fact, the user will never need to open more than ℓ volumes for regular use. As discussed in Section 4.2, this might harm the consistency of the other, unopened volumes, but this is not important: by using the safeword, the user would still be able to convince the adversary that all volumes have been revealed due to the linkage between them and the ciphertext authentication in the headers. Analogously, in solutions like TrueCrypt, a simple way to implement a safeword is to actually always create a hidden volume, even if only a standard volume is desired.

We stress, however, that using this feature is a dangerous proposition: If such possibility exists, and users are allowed to do that, then why not to? The adversary might arguably assume that a user *must* have a safeword, and pressure for its disclosure. This would put at risk those users who decide to not use this feature, who might then be pushed to its adoption. This, in turn, would ruin plausible deniability for everyone, because now we have a system where everyone has a safeword by default.

We believe there is no simple solution to this dilemma: One has either to accept the risk of looking uncooperative and be subject to further interrogation, or to give up PD at all. We remark that, as far as we know, the issue of a safeword feature (or even just its possibility) for plausible deniable filesystems has not been addressed in the literature before, as all implementations we are aware of (including WoORAM-based ones) employ some form of architectural hard limit on the number of possible nested levels of secrecy. We believe this to be a serious operational problem for the security of PD solutions. For this reason, not only do we discourage the use of this feature, but we also propose a way to make the implementation of *any* safeword-like system impossible. This is discussed in Section 6.7, and boils down to the idea of having an unbounded number of possible volumes per device.

4.3 Security

In this section, we prove that Shufflecake achieves single-snapshot security, as defined in Section 2.3.

Theorem 4 (Single-snapshot security of Shufflecake). *The Shufflecake scheme as described in Section 4.1 is a single-snapshot (SS) secure PD scheme according to Definition 3.*

Assumptions. In proving Theorem 4 we will make some assumptions in order to keep the proof compact and intuitive. We assume w.l.o.g. that all passwords are encoded as bitstrings of length λ. Notice how throughout all Section 4.1 we avoided giving concrete security parameters. Although in the real-world instantiation of Shufflecake we are going to have cryptographic primitives with input and output of fixed size (e.g., 128-bit IVs, 256-bit keys, etc), in the context of this proof we can consider them of variable length. This will allow us to produce an asymptotic bound, and to apply it as in Definition 3 in order to prove that the advantage of any (computationally bounded) adversary is indeed negligible in the security parameter. In so doing, we will treat the cryptographic primitives used in the Shufflecake design as *ideal*. More specifically:

- The KDF will be replaced by a *random oracle* $\mathcal{O}_K$, mapping λ-bit passwords and salts to truly-random λ-bit strings.
- The symmetric encryption scheme will be replaced by an *ideal cipher* $\mathcal{E}$, mapping λ-bit keys and IVs to truly-random permutations over $\{0,1\}^\lambda$.
- The authenticated encryption used in the DMB will be replaced by a pair of oracles: the oracle $\mathcal{O}_{AE}$, mapping λ-bit keys and IVs to truly-random injections between plaintext and ciphertext spaces, and its inverse which returns a constant $\perp$ failure symbol if queried outside of the codomain.

All these oracles will be initialised by the game and provided to the adversary $\mathcal{A}$ to be queried freely (at a unitary time cost).

The proof. Let us consider Experiment 2 under the constraints explained in Section 2.3, and let D be the "challenge disk snapshot" provided to the adversary $\mathcal{A}$ by the game (i.e., either D_0 or D_1 according to the secret bit b). For the given $\mathcal{A}$, we will consider ℓ, the decoy passwords $P_1, ..., P_{\ell-1}$, and the access patterns O_0 and O_1 as public parameters of the game instance.

Let us first notice that all the oracle queries performed by $\mathcal{A}$ *before* receiving the challenge disk snapshot D cannot change $\mathcal{A}$'s advantage, because they are completely uncorrelated with D (and the secret bit b). We can, therefore, safely disregard those queries in our analysis.

Then, let us define Q to be the ordered sequence of all queries $\{q_i\}_i$ made by $\mathcal{A}$ to the oracles *after* receiving the challenge D, and define $n := |Q|$. Also define Q_i to be the sequence of queries $(q_1, \ldots, q_i)$ up to query q_i. Analogously, let us define R to be the ordered sequence of all responses $\{r_i\}_i$ returned by the

oracles; also define R_i to be the sequence of responses $(r_1, \ldots, r_i)$ up to response r_i. We consider the execution of $\mathcal{A}$ as the execution of a sequence of single-query stateful adversaries, where the state is just the 'history' of the previous queries: $\mathcal{A}_1(D) \to q_1$, $\mathcal{A}_2(D, Q_1, R_1) \to q_2$, and so on until $\mathcal{A}_{n-1}(D, Q_{n-1}, R_{n-1}) \to q_n$ and $\mathcal{A}_n(D, Q_n, R_n) \to b'$.

Let KEK_i, VMK_i and VEK_i be, respectively, the key-encryption key, the VMB key, and the volume encryption key of volume V_i. Rigorously speaking, in the security game, the values $P_\ell, \mathsf{KEK}_\ell, \mathsf{VMK}_\ell$, and VEK_ℓ are only sampled if $b = 0$. Let us instead consider them to be sampled anyway, and left unused in the case $b = 1$. Let us denote by $\mathcal{S}$ the tuple $(P_\ell, \mathsf{KEK}_\ell, \mathsf{VMK}_\ell, \mathsf{VEK}_\ell) \in \{0, 1\}^{4\lambda}$. Let us define the event E_i as the event that either of $P_\ell, \mathsf{KEK}_\ell, \mathsf{VMK}_\ell$, or VEK_ℓ appear in query q_i (we say that query q_i *strikes*). Finally, let us define $E := E_1 \cup \ldots \cup E_n$ the event that *at least one* query strikes. We will first prove two lemmata.

Lemma 5. $\Pr[E] = \Pr[E \mid b = 0] = \Pr[E \mid b = 1] = \mathsf{negl}(\lambda)$. (The adversary can only guess one of the secrets of V_ℓ with negligible probability.)

Proof. Let us prove that $\mathcal{S}$ is *statistically independent* from any query $q_i \in Q$.

This tuple gets sampled uniformly at random from $\{0, 1\}^{4\lambda}$, so its distribution does not depend on the public values. Since the oracles implement *ideal* cryptographic primitives, it follows that their outputs are statistically independent from their inputs; this does not just hold *marginally* for single input-output pairs, but *jointly*: any tuple of inputs is statistically independent from the tuple of corresponding outputs (for the ideal cipher $\mathcal{E}$, this only holds for the *key* inputs). In particular, $\mathcal{S}$ is statistically independent from the whole tuple (D, R). Since $\mathcal{A}_i$'s inputs, namely the tuple (D, Q_{i-1}, R_{i-1}), are a (randomised) function of (D, R), we deduce that its output, namely q_i, must also be independent of $\mathcal{S}$.

Therefore, since $(P_\ell, \mathsf{KEK}_\ell, \mathsf{VMK}_\ell, \mathsf{VEK}_\ell)$ are uniform, the probability that q_i contains, say, P_ℓ, is $2^{-\lambda}$. Thus, by the union bound, $\Pr[E_i] \leq 4 \cdot 2^{-\lambda}$. Using the union bound again, we get:

$$\Pr[E] \leq \sum_{i=1}^{n} \Pr[E_i] \leq n \cdot 4 \cdot 2^{-\lambda}$$

This expression is clearly $\mathsf{negl}(\lambda)$, since n must be at most a polynomial in λ. The final claim on the equality of the conditional probabilities follows by simply observing that this reasoning holds irrespective of the value of b. $\square$

Lemma 6. $\Pr[\mathcal{A}(D) \to 1 \mid b = 0 \wedge \bar{E}] = \Pr[\mathcal{A}(D) \to 1 \mid b = 1 \wedge \bar{E}]$. (Unless she can guess one of V_ℓ's secrets, the adversary has the exact same *view* in the cases $b = 0$ and $b = 1$.)

Proof. It is sufficient to prove that $\mathcal{A}_n$'s inputs, namely D, Q_n, and R_n, follow the same *joint conditional distribution*, conditioned to the event $\bar{E}$, regardless of whether $b = 0$ or $b = 1$.

We prove this by induction. Using a concise notation, we want to prove that the following quantity does not depend on the bit b:

$$\Pr\left[D, Q_n, R_n \mid \bar{E}, b\right] = \Pr\left[q_n, r_n \mid D, Q_{n-1}, R_{n-1}, \bar{E}, b\right] \cdot \Pr\left[D, Q_{n-1}, R_{n-1} \mid \bar{E}, b\right]$$

Showing that the first factor is independent of b will prove the inductive step. To this end, let us further rewrite it as:

$$\Pr\left[q_n \mid D, Q_{n-1}, R_{n-1}, \bar{E}, b\right] \cdot \Pr\left[r_n \mid q_n, D, Q_{n-1}, R_{n-1}, \bar{E}, b\right]$$

The first factor is independent of b because q_n is the output of $\mathcal{A}_{n-1}$, which only takes D, Q_{n-1}, R_{n-1} as inputs, all of which are among the conditioning terms already. The second factor is independent of b because, given that $\bar{E}$ holds, the oracles behave the same whether $b = 0$ or $b = 1$. This is because striking queries are the only oracle inputs that could trigger responses with unequal distributions (i.e., correlated to D) in the cases $b = 0$ and $b = 1$. But if, instead, we rule these queries out by conditioning on $\bar{E}$, then the oracles instantiated when $b = 0$ are perfectly interchangeable with the ones instantiated when $b = 1$.

We are now left to prove the base step for induction, corresponding to $\Pr\left[D \mid \bar{E}, b\right]$. Let us rewrite it, using Bayes' theorem, as:

$$\Pr\left[D \mid \bar{E}, b\right] = \frac{\Pr\left[\bar{E} \mid D, b\right] \cdot \Pr\left[D \mid b\right]}{\Pr\left[\bar{E} \mid b\right]}$$

The term $\Pr\left[\bar{E} \mid b\right]$ is independent of b by Lemma 5. The same is true of $\Pr\left[\bar{E} \mid D, b\right]$: the same proof applies as for Lemma 5, because the reasoning is unchanged when we condition the probabilities on a particular realisation for D.

We only have to prove that $\Pr\left[D \mid b\right]$ does not depend on the bit b, i.e. that the disk snapshot follows the same *a-priori* (non-conditioned) distribution, whether $b = 0$ or $b = 1$. To prove this, we will use the actual properties of the Shufflecake scheme. Let us proceed by analysing the disk layout region by region.

The blank spaces (empty DMB cells, unmapped slices, etc.) are filled with equally-distributed uniformly-random noise. The spaces occupied by volume V_ℓ, when $b = 0$, are filled with oracle responses to queries containing one of V_ℓ's secrets. These responses are fresh randomnesses, which follow the same distribution as the noise filling the same spaces when $b = 1$.

We are only left to prove that the (decrypted) logical contents and metadata of the decoy volumes follow the same distribution when $b = 0$ and when $b = 1$. Indeed, the *logical* contents of the volumes are fixed and identical in the two cases, determined by O_0 and O_1 (this is because they have to follow the constraint defined in Section 2.3). Also, the DMB cells and the VMBs contain equally-distributed uniformly-random oracle outputs (keys, etc.).

The last step is to show that the position maps of the decoy volumes follow the same distribution in the two cases $b = 0$ and $b = 1$. By the second constraint on the access pattern, defined in Section 2.3, we get that slice allocation is triggered for the same LSIs of the decoy volumes in both the two cases. Even though some more slice allocations are performed on V_ℓ's LSIs when $b = 0$, this does not impact the resulting observable distribution on the PSIs assigned to decoy LSIs. This is because slice allocation always takes a PSI randomly among the free ones, therefore the order in which the LSIs are mapped can be permuted freely without impacting the distribution. Thus, even in the case $b = 0$, we can equivalently imagine that the decoy LSIs get all mapped before V_ℓ's ones, yielding the same distribution as when $b = 1$. This concludes the proof. $\qquad\square$

Proof (Proof of Theorem 4). We use Lemmata 5 and 6 to prove that the advantage of $\mathcal{A}$, as defined in Equation 1 of Definition 3, is negligible. By conditioning both terms of Equation 1 to the events E and $\bar{E}$, we get:

$$\left| \mathsf{Pr}\left[\mathcal{A}\left(\mathsf{D}\right)\to 1 | b{=}0\right] - \mathsf{Pr}\left[\mathcal{A}\left(D\right)\to 1 | b{=}1\right] \right| =$$

$$= \left| \mathsf{Pr}\left[E\right]\left(\mathsf{Pr}\left[\mathcal{A}\left(D\right)\to 1 | b{=}0 \wedge E\right] - \mathsf{Pr}\left[\mathcal{A}\left(D\right)\to 1 | b{=}1 \wedge E\right]\right) + \right.$$

$$\left. + \mathsf{Pr}\left[\bar{E}\right]\left(\mathsf{Pr}\left[\mathcal{A}\left(D\right)\to 1 | b{=}0 \wedge \bar{E}\right] - \mathsf{Pr}\left[\mathcal{A}\left(D\right)\to 1 | b{=}1 \wedge \bar{E}\right]\right)\right| \le$$

$$\le \mathsf{Pr}\left[E\right] \cdot 1 + \mathsf{Pr}\left[\bar{E}\right] \cdot 0 = \mathsf{negl}(\lambda),$$

which concludes the proof. $\qquad\square$

5 Implementation and Benchmarks

We implemented the Shufflecake scheme in the C language as an open-source device-mapper-based driver for the Linux kernel. We published our code under the GPLv2+ license. The current release is v0.4.1 [45]. This section describes the programming environment and the structure of our implementation, and presents concrete performance measurements taking other popular disk encryption solutions as a baseline for comparison.

5.1 Structure of the Implementation

Our implementation consists of two components: a `dm-sflc` kernel module (which does most of the job), and a `shufflecake` companion userland application (used to correctly manage the volumes). The kernel module is the component that actually implements the scheme, translating logical requests into physical requests, and persisting slice maps into the respective headers.

Cryptography. Cryptographic primitives are provided by the Libgcrypt library [19]. We target 128 bits of security. We use Argon2id as a KDF, which was implemented in Libgcrypt recently [18]. We use AES-GCM-256 as an authenticated cipher, and AES-CTR-256 for data encryption, with 128-bit IVs.

The userland application. This component is used to manage volumes creation, opening, and closing. To this end, it manages the DMB and VMB of each volume header. The VEK_i is passed to the kernel module to decrypt the slice map and data section of that volume, while VMK_{i-1} is used to iteratively open all the less-secret volumes, as described in Section 4.1.

This is offloaded to the userland application because key management is arguably better handled in user space: for example, we need to react to an incorrect password by asking the user to try again, not by emitting a kernel log message. There is also another technical hindrance to delegating everything to the kernel module: state-of-the-art KDFs like Argon2id [5] are not currently implemented in the Linux Kernel Crypto API [31], while they are available in user-space software libraries like Libgcrypt [19].

The other blocks of the volume header, which contain the slice map encrypted with the VEK, are managed by the kernel module (except at `init` time, when an empty position map is written by the userland tool).

Volume operations. The `shufflecake init` command takes as input a device path, and then interactively asks the user a number $\ell \leq$ `max` and ℓ passwords as input, correctly formats the first ℓ volume headers, and fills the remaining `max` $- \ell$ slots with random bytes; this way, the pre-existing volumes are wiped by erasing their headers (crypto-shredding). Unless a `--skip-randfill` option is provided (e.g., for testing or debugging purpose), the whole disk is filled with random bytes before formatting the header section. This command only formats the disk: it does not create the Linux virtual devices associated to the volumes.

The `shufflecake open` command takes a device path as input and asks one single password to the user, then looks up the volume headers, and opens all the volumes starting from the one whose password is provided, backwards up to the first one (walking up the chain using the VMK_{i-1} field in the VMB). This is the command that actually creates the Linux virtual devices representing the volumes, under `/dev/mapper`: the names are generated algorithmically. Notice that these virtual devices are not automatically mounted, it is up to the user to mount them and format them with a filesystem of choice when required.

Finally, the `shufflecake close` command takes a device path as input, and closes all the volumes open on that device.

Additional functionalities. In addition to standard features such as command-line usage help and printing on screen the current version, our implementation also offers two additional functionalities: a `changepwd` action, which allows the user to change a volume's password as described in Section 5.1, and a `testpwd` action, which tests whether a provided password unlocks a certain volume (and which one) without actually opening that volume. This might be helpful for the scrupulous user who wants to regularly recall the passwords to decoy volumes, as suggested in Section 4.2.

5.2 Space Utilisation

A few factors influence the disk and RAM space efficiency of Shufflecake, i.e., what part of the storage contains actual data coming from the upper layer, and what part contains metadata, or is otherwise wasted. Overall, with a sensible choice of the parameters, and with reasonable assumptions about the behaviour of the upper layer, we can attain a very low space overhead.

For our implementation, we fixed the block size to 4096 bytes, so as to better amortise the per-block space overhead determined by the IVs. We chose $S_L = 256$, and $\mathsf{max} = 15$. Since we use AES-CTR-256 as the underlying encryption scheme, we need 16-byte IVs. This led to a choice of $\Delta_S = 1$: a single 4-KiB IV block (containing 256 IVs) encrypts a 1-MiB slice. To provide a numerical summary of the space utilisation of Shufflecake, we observe that in the case of a 1 TiB disk, the resulting theoretical maximum utilisable space is 1019.91 GiB, equal to more than 99.6% of the physical storage space.

Headers. With the above parameters, the total size of a volume header is around $\frac{N}{S_P} \log \frac{N}{S_P}$, roughly equal to 4 MiB per volume header, for a 1-TiB disk: about 60 MiB for the total device header size.

IVs. As previously discussed, we store IVs on-disk. With the concrete choice of parameters of our implementation, we have 16-byte IVs encrypting 4096-byte blocks (256 times as much); therefore, we only use $\frac{1}{257}$ ($< 0.4\%$) of the physical data section to store IVs.

Internal fragmentation of slices. Internal fragmentation is a frequent problem in space allocation, and it is particularly well known and studied in file systems theory. For performance reasons, the block layer only works with the block granularity; the file system, therefore, has to allocate a whole block even if it needs less space to, e.g., host a file. Internal fragmentation is the problem arising from this "over-allocation". On top of this, Shufflecake adds another layer of internal fragmentation through its slice mechanism: when a volume requests a block, we reserve a whole slice of S_L blocks just for that volume. Moreover, we have no means of communicating this over-allocation to the file system layer, which therefore has no way of adapting its behaviour. Thus, we have to hope that a file system does not jump back and forth too wildly, and that it generally tries to fill a group of slices before requesting a new one.

Luckily, some file systems do exhibit this behaviour. For example, the commonly used ext4 file system defines the concept of a *block group*, i.e., a group consisting of 32768 consecutive blocks (which amounts to 128 MiB for 4096-byte blocks). The block allocator of ext4 tries its hardest to keep related files within the same block group; specifically, whenever possible, it stores all *inodes* of a directory in the same block group as the directory; also, it stores all blocks of a file in the same block group as its inode [27]. This feature plays nicely with the value of $S_L = 256$ we chose: a block group encompasses a whole number of slices, which will therefore not be too fragmented in the long run.

Releasing unused slices. Our implementation currently does not have a way to reclaim physical slices that were assigned to some volume but are no longer used, when all of the blocks within the corresponding logical slice have been deallocated by the file system. We discuss this in Section 6.6. We note, however, that deallocation of slices can occur more or less frequently depending on the filesystem in use: filesystems with good contiguity will tend to free up consecutive blocks (and hence whole slices) as the data is moved or erased, while filesystems with higher granularity might 'leave blocks around' more often. We cannot release a slice until all the physical blocks therein are freed up. Therefore, the efficiency of any slice-releasing mechanism must be evaluated carefully.

5.3 Benchmarks

We tested our implementation looking at I/O performance and space efficiency. The test environment was a fresh installation of Ubuntu 23.04 running kernel version 6.2.0 on a laptop equipped with an AMD Ryzen 7 PRO 6850U CPU with Spectre mitigations enabled, 32 GiB 4-channel 6400 MHz DDR5 RAM, and a low-power 1 TiB NVMe Micron MTFDKCD1T0TFK SSD. We tested the amount of slice fragmentation of Shufflecake (v0.4.1), its I/O performance, as well as the I/O performance of other two relevant disk-encryption tools for comparison: `dm-crypt`/LUKS (v6.2.0-26) and VeraCrypt (v1.25.9). All the tests were performed sequentially, on a physical primary SSD partition of size 8 GB, using the ext4 filesystem (which is the one most relevant for Shufflecake's envisioned final use case). In the case of Shufflecake, we `init`iated the partition with two volumes (one decoy and one hidden), and performed all tests on the hidden one. Analogously, in the case of VeraCrypt, we formatted the partition as a standard no-FS VeraCrypt volume, and created a 6.5 GB ext4 volume therein. In order to aid reproducibility, we also included in our implementation a suite of benchmark scripts performing the tests described here.

Fragmentation In order to evaluate the fragmentation caused by Shufflecake's allocation of slices, we filled the ext4 filesystem with incrementally large amount of random files and directories up to saturating the space, and at every step we measured the space given by the increasing number of slices allocated by `dm-sflc` for the hidden volume VS the total amount of data written therein. We define *space efficiency* as the ratio between real data written on disk and slice-allocated space (0 = bad, 1 = good).

The results are shown in Figure 7. As we can see, even when the disk is initially empty, some slices are immediately allocated for the ext4 journal and metadata. However, as data is written on disk, the effect of fragmentation quickly disappears: already at 10% of data capacity the space efficiency is above 90%, and at 25% of data written it reaches 95%. We conclude that the slicing algorithm of Shufflecake behaves very well in our simulated random usage pattern, at least with the ext4 filesystem, and slice fragmentation can be considered negligible.
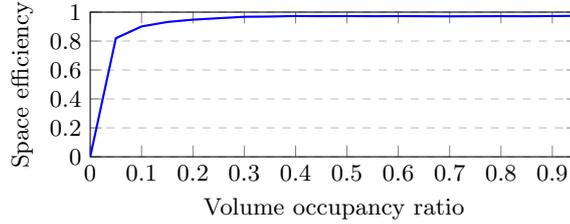
Fig. 7: Shufflecake space efficiency as the ext4 filesystem fills up.

I/O and bandwidth. For testing the I/O performance of Shufflecake against `dm-crypt`/LUKS and VeraCrypt, we used the `fio` benchmarking tool, which can flexibly measure various metrics. For each of the three disk encryption tools, we performed both random and sequential read/write operations with large amount of data on the filesystem. We fixed the same parameters for all tests, such as a queue of 32 operations and a block size of 4 kiB, which are commonly recommended to evaluate real-world performance of disks. Under these conditions, we found no observable difference in metrics between IOPs (I/O operations per second) and bandwidth (expressed in MB/s), hence we report the results looking at the bandwidth only.

	Shufflecake	`dm-crypt`/LUKS	VeraCrypt
random write	26.77	38.43	39.07
random read	26.78	38.44	39.09
sequential write	176.87	247.14	247.75
sequential read	177.10	247.43	248.04

Table 1: I/O performance (in MB/s) of Shufflecake, dm-crypt/LUKS, and VeraCrypt (higher = better).

The results are shown in Table 1. As we can see, Shufflecake incurs in an I/O slowdown of roughly 30% compared to the other tested tools. We believe this overhead to be acceptable in daily use.

Comparison with WoORAMs. The comparison with some popular ORAM-based PD solutions more convincingly shows the real-world efficiency advantages offered by Shufflecake. Of course it has to be stressed that these ORAM-based solutions aim at achieving PD in a more strict scenario than the single-snapshot security offered by the current version of Shufflecake. We have just seen how Shufflecake achieves a slowdown of roughly 30% I/O throughput over `dm-crypt` and uses almost all space available. On the other hand, HiVE [6] has a heavy 200x I/O overhead and wastes 50% of the disk space. DetWoOram [39] has an overhead of 2.5x for `read`s and 10x-14x for `write`s, and wastes 75% of space.

37

6 Conclusions and Future Directions

We have seen how Shufflecake represents a usable PD scheme with many operational advantages over solutions like TrueCrypt. We released it as an open source tool in the hope of building trust and adoption in the community, and possibly encouraging contribution to future work. In fact, many possibilities for further improvement exist. We are going to mention some in this section.

6.1 Crash Consistency

As it is right now, the main obstacle to reach maturity and adoption for daily use is that Shufflecake is not crash-consistent. This means that if the program crashes during operation with one or more open volumes, data corruption is possible, because some volume state changes happen in-RAM and are cached for some time before being written on disk. This is a problem for Shufflecake, because if a crash occurs between a write of encrypted data to disk and its associated IV, the encryption becomes unrecoverable. The situation is different for solutions like LUKS or TrueCrypt, which use the XTS mode of operation and are therefore immune to this problem because they do not use explicit random per-block IVs. To fix this while keeping the property of block re-randomisation, we should make the individual logical write requests atomic, i.e. we should mask the fact that they map to several physical requests (which need to be assumed atomic): if a crash happens at any point between two of these physical requests, the old content of the logical block should still be recoverable, the disk should not be left in a "limbo" state that does not correspond to any of the logical contents written by the upper layer. We discuss here some ideas for future improvements to address these concerns.

Shufflecake incurs crash-inconsistencies when a crash happens in the time window between the update of a data block and the update of the corresponding IV block. As was discussed, Shufflecake adopts a write-on-flush approach for the IV cache, whereas data blocks are immediately written to disk, encrypted with the new IV (which is not immediately persisted on-disk); therefore, the disk is in a "vulnerable" (inconsistency-prone) state whenever the upper layer has written on a file and has not yet synced it: this is, reasonably, a large fraction of the total operating time. To solve this, it would be necessary to make the IV cache write-through; the performance impact of such a choice has not been evaluated but it would probably be heavy. It would not be sufficient, anyway, because it would only reduce the "vulnerability window" between the update of a data block and that of the IV block, it would not eliminate it.

The final solution would be to also duplicate each IV block into a circular log of length 2: the update of the IV block synchronously precedes the update of the data block, and overwrites the older of the two versions; this way, if the crash happens right afterwards, and the data block is not updated, it is still decryptable because the corresponding IV block has not been touched. Disambiguation (i.e., deciding which of the two versions of the IV to use) would be based on an additional MAC on the data block (stored alongside the IV); this would only be

needed when the block is read for the first time since volume opening: afterwards, the state can be kept in RAM (it is just one bit for each slice).

An alternative solution, which wastes more disk space but we believe to be overall better, would be to store the IV alongside the data block itself, so that the two updates can be merged into a single physical request. We believe that, in addition to mitigate the issue of crash inconsistency, this approach would probably lead to better I/O performance, as it would not need separate operations for writing IV blocks during data writes. The minimum addressable unit of disk storage space, at least on Linux systems, is usually the 512-byte sector. Therefore, the least wasteful option is to map a logical 4096-byte block (8 sectors, as was already the case for our implementation) onto 9 consecutive physical sectors: the first one contains the IV, the other ones constitute the data block. This would lead to a waste of disk space (fraction of disk not used for upper-layer data) equal to $\frac{1}{9} = 11.1\%$. Since an IV only occupies 16 bytes, much of the first sector would be left unused; we could use the rest of the free space to also contain additional useful data, for example a MAC to detect IV alterations, and a "reverse map", indicating which logical block B of which volume V_i that physical block corresponds to (this information would of course be encrypted). Additionally, we could build an even more fault-resilient system by again having a circular log of two IVs in the first sector (each accompanied by the corresponding MAC), plus 16 MACs, one for each IV and for each of the 8 data sectors. This would allow us to disambiguate with the sector granularity, in case the underlying disk can ensure atomicity of the sector writes, but not of the physical requests writing on several adjacent sectors.

6.2 Multi-Snapshot Security

The way it has been presented so far, Shufflecake is completely vulnerable to multi-snapshot attacks in the exact same way as TrueCrypt (and its successor, VeraCrypt): when the adversary sees "empty" slices change across snapshots, the only possible explanation is that there still is a hidden volume whose password has not been provided. We have already argued in Section 1.2 why, in practice, this level of security might be enough in most cases, and also why we believe current WoORAM-based solutions base their promise of stronger security on somewhat hard to justify assumptions. We already know [10] that achieving "complete" multi-snapshot security requires the use of WoORAMs, which have serious performance drawbacks. Regardless, as mentioned in Section 4, we designed Shufflecake with the idea of being able to add features that might help to reach an unproven, "operational" level of multi-snapshot security. In this section, we explore the possibility to achieve this goal through some separate, "orthogonal" pattern obfuscation procedure that operates independently of the main scheme and does not interfere with its single-snapshot security. We present three high-level ideas to accomplish this. They are not currently implemented, nor are they precisely specified from a conceptual point of view; instead, they are left as pointers for future research, since this area is greatly under-studied.

Security Notion Revisited. Let us first clarify what we mean by "operational" multi-snapshot security. The rationale is the hope that even though the distinguishing advantage of the adversary in the multi-snapshot game is not negligible (and so the scheme is not cryptographically secure in the strict sense), it is still low enough for an investigation to be inconclusive against an adversary in practice. In other words, we argue that *legal* or anyway *operational* security, in this context, is not the same as *cryptographic* security in the theoretical sense.

Physical slices have known boundaries, so the adversary can compare the disk snapshots she has on a slice basis. This amounts to inspecting the subsequent changes, or *diffs*, each physical slice goes through across snapshots. Recall that a physical slice is essentially an array of S_P blocks; when one of these blocks changes, it can either be because of the re-encryption of a data block (with a different IV, and possibly a different content), or because of the re-randomisation of an empty block: the IND-CPA security of the encryption scheme guarantees the indistinguishability of these two situations. This means that no hint on the nature of a block (i.e., whether it is a data or an empty block) is leaked by its encrypted content or by the history of its encrypted contents. Therefore, when comparing two snapshots of a physical slice, the only information that an adversary gets is *which* of the S_P blocks have changed: *how* they have changed is completely inconsequential and uninformative. In other words, the diff between two snapshots of a physical slice boils down to a bitmask of S_P bits, representing which blocks have changed and which remained the same.

The task of the adversary then becomes to distinguish between "data slices" (i.e., belonging to some volume) and "free slices" (not mapped to any volume) based on a sequence of such diffs for each slice. The point is that, after unlocking the first $\ell - 1$ volumes, the rest of the space may or may not contain another volume (there may or may not be some more data slices among the free slices): if we want the adversary to be incapable of distinguishing between the two cases, we need the diffs of data slices and free slices to look the same. Once we frame the problem in this way, we can rephrase the weakness of Shufflecake as follows: the diff bitmask of a free slice is always all-zeros, and as such is clearly distinguishable from that of a data slice, which might have some bits set to 1.

Our task then becomes to "obfuscate" the changes that occurred in the data slices (especially those belonging to V_ℓ) by artificially creating a non-zero diff bitmask in the free slices, through a re-randomisation of some selected blocks. This way, a user will hopefully be able to claim that all the changes happened to the "empty" blocks are due to this obfuscation procedure, and not to the existence of V_ℓ. Nothing prevents us, of course, from also touching the data slices during the obfuscation procedure, if that helps making the diff bitmasks look more alike. It is to be noted, however, that if a block was modified by the upper layer during the normal operational phase, the corresponding bit will be set to 1 in the diff bitmask and there is no way for the obfuscation procedure to "undo" that change: when we touch a data slice, we cannot turn the 1s of the diff bitmask into 0s. Instead, we can turn some 0s into 1s by simply re-encrypting the same content of a block with a different IV.

Trivial Random Refresh. A very simple first idea for an obfuscation procedure is to take all physical blocks belonging to free slices, and re-randomise them all, independently at random, each with probability p. This operation could be either performed upon volume `close` or, for better resilience, spread across the normal operations. It is the easiest way to achieve a non-zero diff bitmask for free slices, but it is definitely too crude to work: nothing guarantees that the diff bitmasks of data slices will "look random" like the ones artificially generated for free slices. Also, it might very well be the case that many data slices do not change across two snapshots, in which case it becomes very easy to tell them apart from free slices, which often have a non-zero diff bitmask.

Such consideration suggests a refinement: besides re-randomising some blocks in the free slices, we could re-encrypt (with a different IV but same plaintext) some blocks in the data slices of existing volumes, again independently at random, each with probability q. This way, we also randomise the diff bitmasks of the data slices, making them more similar to those generated for free slices.

Insecurity with many snapshots. The procedure we just illustrated could well succeed, for a suitable choice of p and q, in rendering the diff bitmasks of data slices and free slices roughly indistinguishable, but only if the adversary gets just *one* diff for each slice, i.e., if she only gets two snapshots. This is because we are essentially playing a hopeless game: roughly speaking, we are aiming at making signal+noise (the diffs of a data slice) indistinguishable from noise alone (the diff of a free slice). As discussed, we cannot turn the 1s of the diff bitmasks of data slices into 0s: the "signal" given by which blocks were modified by the upper layer stays there, we can only hope to bury it in enough noise by turning some 0s into 1s through re-encryption. However, with enough snapshots, the signal will eventually emerge. Imagine, for instance, that there is one particular block in a data slice that is very often modified (maybe it contains some sort of file system index): the corresponding bit in the diff bitmask will often be set to 1, which would be hard to justify through the obfuscation procedure, which only hits one given block with probability p each time.

Subsampling. The previous discussion teaches us a valuable lesson: in our setting, we cannot hope to disguise the accesses performed by the upper layer as random noise, as is commonly the case for ORAMs. The only option we have left is to take the opposite approach: let us make the diffs of free slices look like they were *also* generated by some file system workload. This way, we are still making the diffs for the two kinds of slices similar, but we are not trying to erase the signal from existing data slices, which we cannot. Instead, we "copy this signal" onto the free slices, so that the adversary will *always* see this signal, even if the last ℓ-th volume has been surrendered. A tedious, convoluted, and yet very imprecise way of doing it would be for us to sit down, study the access patterns resulting from typical file system workloads, model them as a probability distribution, and hardcode that into the obfuscation procedure. Probably, a better idea would be to use a ML approach and let a daemon run in the background to adaptively learn and simulate such distribution.

Instead, a simpler method that is likely to capture the patterns we want to imitate is to have the scheme itself "learn" them online, by simply subsampling the stream of incoming logical requests. More concretely: for each incoming logical `SflcWrite` request, we "imitate" it with probability p. Imitating a request means "learning" that the affected logical block b is likely to be written by file system workloads, and copy this signal onto a free slice: we retain the offset (b mod S_L) of the block within the slice, we choose a "target" free slice, and we re-randomise the block with the same offset within the target slice. This approach guarantees that, if a particular block is often updated by the upper layer, then we are very likely to catch this signal and correctly carry it over to a free slice. Note, however, that we need to choose the target free slice deterministically from some context information. Only in this way can we replicate the signal consistently across snapshots, always onto the same free slice; also, this allows us to correctly capture and copy a signal in case it consists of not just one, but several blocks in a slice being frequently updated.

Counting attacks. If we assume that the obfuscation procedure just described really succeeds in making the empty space look like it is occupied, we are still left with one problem. The special blocks that are often updated by the upper layer leave a very clear trace in the snapshot history, since their bits in the diff bitmasks are almost always set to 1. If we have ℓ volumes, the obfuscation procedure will generate ℓ such clear traces in the free space. Therefore, a simple attack would consist in counting these traces, and checking whether there are as many of them as there are disclosed volumes. To thwart this attack, we can rework the obfuscation procedure in such a way that the device's data section always looks like it's hosting `max` volumes. We can ideally assign the free slices to $\texttt{max} - \ell$ pairwise-disjoint sets, each one representing a "fake" volume, and have the obfuscation procedure be aware of this partitioning when choosing the target free slice, so as to really simulate $\texttt{max} - \ell$ volumes with the imitated logical `SflcWrite` requests.

Ghost File System. The obfuscation procedure described above may already offer good protection, although it might be non-trivial to translate the rough idea of "being aware of the partitioning into $\texttt{max} - \ell$ fake volumes" into a concrete algorithm for choosing a target free slice when we imitate a `SflcWrite` request. A valid, if "exotic", alternative, would be to actually create $\texttt{max} - \ell$ additional "ghost" Shufflecake volumes on the device, that behave in the exact same way as regular volumes, except that they do give up their slices when needed. On these volumes, a separate component (a daemon) could mount a file system and perform some typical sequences of accesses. The advantage of this solution is that, by definition, it will always look like there are `max` volumes on the device. However, when slices are reclaimed from ghost volumes, their file systems might suddenly get corrupted and start complaining quite loudly, thus impairing the practical usability of the system. A daemon operating on these ghost filesystems might therefore need to be aware of the new slice allocation requests from "real" volumes, and move or remap the ghost slices accordingly at runtime.

6.3 Shufflecake "Lite"

As we have seen in Section 6.1, crash inconsistency is a serious problem of the current embodiment of Shufflecake, caused by the use of the CTR mode of encryption which, in turn, is necessary to achieve block re-randomisation. And we have seen in Section 6.2 how this block re-randomisation is an essential ingredient for achieving some form of multi-snapshot security. However, currently no multi-snapshot security measure is implemented in Shufflecake, or even formally defined. We have already seen in Section 5.3 how the current CTR design brings some (minimal) performance hit, and how this performance hit will likely be exacerbated both by the discussed mechanisms for achieving crash consistency and by the proposed ideas for multi-snapshot security. Furthermore, we have argued in Section 1.2 how single-snapshot security might already offer a good enough security margin in many scenarios.

This brings us the following idea: proposing a *"lite"* mode of Shufflecake which sacrifices the feature of block re-randomisation (with all its pros and cons) and employs the XTS mode of operation instead of CTR for encryption, just like TrueCrypt. This would give up every hope of achieving any form of security better than single-snapshot, but it would bring the following advantages:

- It would avoid the need of writing IVs on disk, therefore avoiding completely the (minimal) space waste and I/O slowdown measured in Section 5.3.
- It would be *natively* crash-consistent, so all the countermeasures discussed in Section 6.1 would be unnecessary.
- Compared to TrueCrypt/VeraCrypt, it would still offer huge operational advantages: providing "real" plausible deniability by offering many nested layers of secret volumes, unlocking a whole hierarchy of volumes with a single password, and being filesystem-agnostic.

The idea would be eventually to provide both modes for Shufflecake, "lite" and "full" (the latter including both crash consistency countermeasures and multi-snapshot capabilities), and let the user choose which one is desired during the `init` operation. In practice, one of the two would be the default choice and the other one could be selected as optional, but deciding which of the two shall be the default and which one optional will require a careful security and usability study. We leave this decision for future work, after proper confrontation with representatives of the envisioned final user demographics.

6.4 Corruption Resistance

As discussed in Section 6.4, writing data to decoy volumes while not all the hidden volumes are open entails a high risk of corrupting the hidden volumes. There cannot be perfect mitigation to this problem, save for frequent backups. However, it could be possible to reduce this risk of corruption by using some form of *error-correction* on the unopened volumes (hence sacrificing some space), and then trying to recover the volume if corruption happens. We tested positively this idea using RAID[33], namely by partitioning a Shufflecake hidden

volume into different equal-sized logical partitions, and using them to assemble a RAID device. Other methods might be more suitable, such as *alpha entanglement codes*[14], but the support should be probably "baked-in" into Shufflecake itself in order to be really robust.

6.5 Use of Disk Metadata

We have seen in Section 4.1 how there is space for embedding volume-specific metadata in each volume's VMB. Here we discuss a couple of useful ideas on how to employ this extra space.

One option could be to embed a string specifying a user-defined name for the volume. Currently, our implementation assigns volume names procedurally in order to avoid name collisions, but a user might prefer to assign these names statically or with mnemonic IDs, to facilitate scripting etc.

Likewise, one could embed a string specifying a desired *mountpoint* for that volume. Notice in fact that, given the PD requirements, one cannot let these volumes be assigned at static mountpoints in a regular Linux way, e.g. using `fstab` or `crypttab`. Rather, the desired mountpoint should be hidden within the context of the volume itself. Then, if the implementation supports it, once the volume is `open`ed, it can also be automatically `mount`ed at a given position.

Another idea could be to embed *virtual quotas*, in order to artificially limit the maximum available size of decoy volumes. As it is now, Shufflecake performs maximum overcommitment on the visible available space of all volumes: each of them will appear as large as the underlying device. This can cause issues if the user (or the OS) mistakenly assumes that that space is actually available, and starts writing too much data on the volumes. In order to mitigate this, metadata could be used to limit the size of the block device seen by the OS. Importantly, this must only hold for decoy volumes, because overcommitment is substantially what allows for PD. Therefore, the correct way to implement this is: *each volume's VMB should specify a virtual quota for the volume below itself in the secrecy hierarchy, but not for itself.* When an ℓ-th volume is `open`ed, the virtual quota for all less secret volumes in the hierarchy will be recursively read this way (the lowest volume, e.g. volume 1, will not have this assigned metadata, or anyway it will be ignored). Then, the ℓ-th volume could be assigned a virtual size equal to the maximum available size on the device, minus the *sum* of the virtual quotas of all other volumes. This way, it will always be impossible to accidentally write too much data on the volume hierarchy, but an adversary will always see the most secret unlocked volume as *the* last one present.

It might also be possible, athough probably overkill and exceedingly complex to implement, to have different security or redundancy policies assigned *per-volume* rather than *per-device*, and use the metadata to disambiguate them. For example, it could in theory be possible to have different features in terms of crash consistency (as discussed in Section 6.1), security (Section 6.2) or corruption resistance (Section 6.4) assigned to different volumes within the same hierarchy.

Finally, all this metadata could be embedded in raw text, or a more robust and machine-friendly encoding such as JSON could be used.

6.6 Reclaiming Unused Slices

Currently, our implementation of Shufflecake does not have a mechanism for reclaiming slices that are no longer used: once a slice is allocated for a certain volume, it will always belong to that volume, even if the volume's filesystem is emptied of all data. It would be desirable to implement an operation to reassign empty slices to the pool of free available ones, in order to make space allocation across volumes more efficient and limiting the risk of overcommitment.

Clearly, we need some sort of hint from the upper layer in order to trigger this operation. To that effect, we need to intercept the `trim` requests emitted by the file system. These commands are effectively a third instruction accepted by hard disks, besides `read` and `write`; they serve as a way for the file system to indicate to the disk that some sectors no longer contain user data, and so the internal disk controller can avoid copying them over when reshuffling its own internal indirection layer [17]. These commands are also vital for the efficiency of disk virtualisation systems, such as Shufflecake, that overcommit the total underlying space and thus need to exploit every occasion to optimise the resource allocation.

Once we have this mechanism in place, we can design a way to reassign a freshly freed-up physical slice to the pool of available ones at a random position, so that the function `NewSlice` (Algorithm 3) will return it with uniformly random probability when a new slice is required. More concretely: suppose that Shufflecake intercepts an OS signal telling us that a logical slice for volume V_i at LSI σ is now free. We define a function `ReclaimSlice` (Algorithm 4) which operates on the same structures used by `NewSlice`, and also on the position map of the interested volume. This function clears the occupation bitfield of the reclaimed slice and the entry in the position map, then moves the PSI at a random position of `prmslices` (after `octr`) by doing another Fisher-Yates iteration. We use a subfunction `ReverseShuffle` which, given as input a PSI, returns the index of `prmslices` where this PSI is found, or error if not present. The way to implement this subfunction can vary, e.g. by keeping in-memory a reverse map of `prmslices`, or by doing a linear search every time.

Algorithm 4 `ReclaimSlice`(V_i, σ)

1: $\Psi \leftarrow$ `GetPosMap`(V_i, σ)
2: `bfld`$[\Psi] :=$ `free`
3: `PosMap`$_i[\sigma] := \bot$
4: $k \leftarrow$ `ReverseShuffle`(Ψ)
5: **if** $k >$ `octr` **then: return**   ▷ No need to reshuffle in this case.
6: `swap(prmslices`$[k]$, `prmslices[octr])`
7: $j \xleftarrow{\$} [$`octr`, `MaxSlices`$]$
8: `swap(prmslices`$[j]$, `prmslices[octr])`
9: **if** `bfld[prmslices[octr]]` $=$ `free` **then:** `octr` $:=$ `octr` $- 1$
10: **return**

6.7 Unbounded Number of Volumes

Shufflecake assumes a number `max` of possible volumes that can be provided by any device. Even if this limit can be chosen freely by implementations, it would be desirable to have a way for creating unlimited numbers of volumes (subject to space availability) per device. This would not only remove an artificial limitation on the scheme, but would also strengthen its operational security by making any kind of safeword-like technique (as discussed in Section 4.2) impossible.

Clearly, for this to be achievable, volume headers cannot be adjacent and packed at the beginning of the disk. One idea for further investigation would be to embed every header (except the first, 'less secret' one, which is still going to be at the beginning) at random positions within the device, and having them linked by the previous volume header through an ad-hoc pointer field which is *always* present, and indistinguishable from random without the correct password. Traversing this list of linked headers, however, presents some challenges. In particular, when the user provides one password on volume instantiation, how do we know whether the password is wrong? And, if not, how do we reach the right header unlocked by that password? It might be possible to devise some complex linking scheme for an arbitrary number of "bogus" headers on the device, but in any case the following limitations would apply.

First, bogus headers cannot "reserve" an area of the disk, otherwise we would waste too much space. They can be placed at any position during device initialization, but when a new slice allocation falls on their position, that space should be released. One can think of different ways to handle this, for example by dynamically moving out bogus headers to another free position if they are about to be overwritten, or simply accepting the risk of breaking the list at some random point during use (as this would not impact consistency for the "real" volumes, and it would still be enough to justify the impossibility of an a-priori generated safeword). In any case, except for this difference in reserving allocation, "bogus" and "real" headers should either be treated equally, or extra precautions should be adopted to maintain PD.

Second, once the user inserts a password to instantiate a device, we might not be able to tell anymore whether the password unlocks something or it's wrong (e.g., a typo). Instead, depending on the chosen solution, the program might continue to traverse the linked list in search of something to decrypt with that password, until either it finds the right header, or the list is broken (e.g., by a bogus header which was overwritten), in which case we can say the password was wrong. We might even envision that the user should expect to terminate manually the program in case a provided password does not succeed after some time, because any hardcoded timeout in the implementation could nullify this feature by inserting a de-facto artificial limit to the number of possible headers.

A possible way to implement this idea could be the following:

1. Coalesce DMB and VMBs into unified, per-volume headers.
2. Each header is one slice large.
3. Every header also contains a field with a random value `nxtptr`.

4. Except for the first one, headers are found at random disk positions that are a (public) function of the previous header's `nxtptr`.
5. During `init`, in case there is a collision during a `nxtptr` generation over the location of another pre-existing header, the currently generated `nxtptr` value is discarded and sampled again, until a suitable one is found by brute-forcing (since the total header size is supposed to be negligible in comparison to the device size, this should be very efficient).
6. All the headers are functionally equivalent and contain the same fields.
7. The first header contains a value that is the KDF's salt, while the same field in other headers is either left unused, or used to re-salt the password-derived key for every header using a (fast) hash function.
8. Shufflecake would allocate slices for the volumes in the usual way, just considering the slices at header locations as permanently occupied.

The above idea might very well work, but it remains to specify an efficient way to embed this way also the position maps, which might be larger than one slice. Many options are open to evaluation here, from linked lists starting at the header, to multiple branching pointers.

There might be other good ways to implement the possibility of having a virtually unlimited number of volumes, we leave this for future exploration.

6.8 Hidden Shufflecake OS

As discussed in Section 1.3, a PD solution that only provides volumes for data storage will never achieve a satisfying level of operational security due to leakage from the OS and other applications installed therein. In order to solve this issue, it is important that the OS itself is run from within a hidden volume, as it was done with TrueCrypt's concept of hidden OS. The natural evolution for Shufflecake would be to be launched at boot time (e.g., as a GRUB module [3]) and boot a whole Linux distribution installed within a volume. Alternatively, an ad-hoc, minimal Shufflecake bootloader could be deployed.

More concretely, eventually Shufflecake could become itself a full PD-focused Linux distribution, where during installation the user is guided in the process of creating volumes and installing other distributions therein in a guided way. For operational efficiency and security, every OS at layer k should be aware of the filesystem and OS in the volume at layers $j < k$ (which is made possible by the hierarchy among Shufflecake volumes). This would also allow a *butler daemon* to run from the currently running OS and operate in the background on lower-hierarchy OSes, e.g. by performing system updates, downloading emails, etc., so that all these decoy systems are kept up-to-date even if the user neglects to use them regularly. This would in turn allow to ease the suspicion of an adversary when surrendering a decoy password.

As an alternative to having a full Linux distribution for every volume, a hypervisor-based solution like Qubes OS [41] might be used instead. However, in order to validate this approach, further analysis is required to ensure that the hypervisor (which is not designed with PD in mind) does not leak the existence of hidden volumes.

References

1. Anderson, R., Needham, R., Shamir, A.: The Steganographic File System. In: Lecture Notes in Computer Science (01 2000). https://doi.org/10.1007/3-540-49380-8_6

2. Ashtana, A.: Revealed: British councils used Ripa to secretly spy on public. [Online; accessed 2023-10-04] https://www.theguardian.com/world/2016/dec/25/british-councils-used-investigatory-powers-ripa-to-secretly-spy-on-public (2016)

3. Babar, Y., Babar, Y.: GRUB bootloader. Hands-on Booting: Learn the Boot Process of Linux, Windows, and Unix pp. 133–181 (2020)

4. BBC: Man jailed over computer password refusal. [Online; accessed 2023-10-04] https://www.bbc.com/news/uk-england-11479831 (2010)

5. Biryukov, A.: Argon2. [Online; accessed 2023-10-04] https://www.password-hashing.net/#argon2 (2013)

6. Blass, E.O., Mayberry, T., Noubir, G., Onarlioglu, K.: Toward Robust Hidden Volumes Using Write-Only Oblivious RAM. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. p. 203–214. CCS '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2660267.2660313, https://doi.org/10.1145/2660267.2660313

7. Burke, J.: Kenyan election official 'tortured and murdered' as fears of violence grow. [Online; accessed 2023-10-04] https://www.theguardian.com/world/2017/jul/31/kenyan-election-official-christopher-msando-dead-before-national-vote (2017)

8. Casciani, D.: Why Cage director was guilty of withholding password. [Online; accessed 2023-10-04] https://www.bbc.com/news/uk-41394156 (2017)

9. Chakraborti, A., Sion, R.: SqORAM: Read-Optimized Sequential Write-Only Oblivious RAM. Proceedings on Privacy Enhancing Technologies **2020**, 216 – 234 (2017), https://api.semanticscholar.org/CorpusID:201070384

10. Chen, C., Liang, X., Carbunar, B., Sion, R.: SoK: Plausibly Deniable Storage. Proceedings on Privacy Enhancing Technologies **2022**, 132–151 (04 2022). https://doi.org/10.2478/popets-2022-0039

11. Czeskis, A., Hilaire, D.J.S., Koscher, K., Gribble, S.D., Kohno, T., Schneier, B.: Defeating Encrypted and Deniable File Systems: TrueCrypt v5. 1a and the Case of the Tattling OS and Applications. In: USENIX Summit on Hot Topics in Security (HotSec) (2008)

12. Dworkin, M.: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions (2015-08-04 2015). https://doi.org/https://doi.org/10.6028/NIST.FIPS.202

13. Eberl, M.: Fisher–Yates shuffle. Archive of Formal Proofs (September 2016), https://isa-afp.org/entries/Fisher_Yates.html, Formal proof development

14. Estrada-Galiñanes, V., Miller, E., Felber, P., Pâris, J.F.: 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). pp. 183–194. IEEE Computer Society, Los Alamitos, CA, USA (jun 2018). https://doi.org/10.1109/DSN.2018.00030, https://doi.ieeecomputersociety.org/10.1109/DSN.2018.00030

15. Fedora Project: LUKS. [Online; accessed 2023-10-04] https://docs.fedoraproject.org/en-US/quick-docs/encrypting-drives-using-LUKS/ (2022)

16. Frankel, S., Kelly, S.G.: Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec. RFC 4868 (May 2007). https://doi.org/10.17487/RFC4868, https://www.rfc-editor.org/info/rfc4868

17. Frankie, T., Hughes, G., Kreutz-Delgado, K.: A Mathematical Model of the TRIM Command in NAND-Flash SSDs. In: Proceedings of the 50th Annual Southeast Regional Conference. p. 59–64. ACM-SE '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2184512.2184527, https://doi.org/10.1145/2184512.2184527

18. GNU Project: Libgcrypt 1.10.1 released. [Online; accessed 2023-10-04] https://lists.gnu.org/archive/html/info-gnu/2022-03/msg00007.html (2022)

19. GNU Project: Libgcrypt. [Online; accessed 2023-10-04] https://gnupg.org/software/libgcrypt/index.html (2023)

20. Heron, S.: Advanced Encryption Standard (AES). Network Security **2009**(12), 8–12 (2009). https://doi.org/https://doi.org/10.1016/S1353-4858(10)70006-4

21. In re Boucher: In re Boucher — Wikipedia, The Free Encyclopedia. [Online; accessed 2023-10-04] https://en.wikipedia.org/wiki/In_re_Boucher (2022)

22. Institute of Electrical and Electronics Engineers: IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices. IEEE Std 1619-2018 (Revision of IEEE Std 1619-2007) pp. 1–41 (2019). https://doi.org/10.1109/IEEESTD.2019.8637988

23. Jia, S., Xia, L., Chen, B., Liu, P.: DEFTL: Implementing Plausibly Deniable Encryption in Flash Translation Layer. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 2217–2229. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134011, https://doi.org/10.1145/3133956.3134011

24. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press (2007)

25. Key Disclosure Law: Key disclosure law — Wikipedia, The Free Encyclopedia. [Online; accessed 2023-10-04] https://en.wikipedia.org/wiki/Key_disclosure_law (2022)

26. Larsen, K.G., Nielsen, J.B.: Yes, There is an Oblivious RAM Lower Bound! In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 523–542. Springer International Publishing (2018)

27. Linux: ext4 High Level Design. [Online; accessed 2023-10-04] https://docs.kernel.org/filesystems/ext4/overview.html (2022)

28. McDonald, A.D., Kuhn, M.G.: StegFS: A steganographic file system for Linux. In: International Workshop on Information Hiding. pp. 463–477. Springer (1999)

29. McGrew, D., Viega, J.: The Galois/counter mode of operation (GCM). Submission to NIST Modes of Operation Process **20**, 0278–0070 (2004)

30. Microsoft: BitLocker. [Online; accessed 2023-10-04] https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview (2022)

31. Mueller, S.: Kernel Crypto API. [Online; accessed 2023-10-04] https://www.kernel.org/doc/html/v4.16/crypto/index.html (2022)

32. Oechslin, P.: Making a Faster Cryptanalytic Time-Memory Trade-Off. In: Boneh, D. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 2729, pp. 617–630. Springer (2003), http://dblp.uni-trier.de/db/conf/crypto/crypto2003.html#Oechslin03

33. Patterson, D., Gibson, G., Katz, R.: A case for Redundant Arrays of Inexpensive Disks (RAID). ACM SIGMOD Record **17** (07 1988). https://doi.org/10.1145/50202.50214

34. Percival, C., Josefsson, S.: The scrypt Password-Based Key Derivation Function. RFC 7914 (Aug 2016). https://doi.org/10.17487/RFC7914, https://www.rfc-editor.org/info/rfc7914

35. Peters, T., Gondree, M., Peterson, Z.: DEFY: A Deniable, Encrypted File System for Log-Structured Storage. In: Network and Distributed System Security Symposium (01 2015). https://doi.org/10.14722/ndss.2015.23078

36. Register, T.: Brazilian banker's crypto baffles FBI. [Online; accessed 2023-10-04] https://www.theregister.com/2010/06/28/brazil_banker_crypto_lock_out/ (2010)

37. Reuters: UK asked N.Y. Times to destroy Snowden material. [Online; accessed 2023-10-04] https://www.reuters.com/article/us-usa-security-snowden-nytimes-idUSBRE97T0RC20130830 (2013)

38. RIPA: Regulation of Investigatory Powers Act 2000 — Wikipedia, The Free Encyclopedia. [Online; accessed 2023-10-04] https://en.wikipedia.org/wiki/Regulation_of_Investigatory_Powers_Act_2000 (2022)

39. Roche, D.S., Aviv, A., Choi, S.G., Mayberry, T.: Deterministic, stash-free write-only ORAM. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 507–521 (2017)

40. Rubber-hose cryptanalysis: Rubber-hose cryptanalysis — Wikipedia, The Free Encyclopedia. [Online; accessed 2023-10-04] https://en.wikipedia.org/wiki/Rubber-hose_cryptanalysis (2022)

41. Rutkowska, J., Wojtczuk, R.: Qubes OS architecture. Invisible Things Lab Tech Rep **54**, 65 (2010)

42. Saarinen, M.J.O., Aumasson, J.P.: The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). RFC 7693 (Nov 2015). https://doi.org/10.17487/RFC7693, https://www.rfc-editor.org/info/rfc7693

43. Stallings, W.: NIST block cipher modes of operation for confidentiality. Cryptologia **34**(2), 163–175 (2010)

44. Star, T.: How a Syrian refugee risked his life to bear witness to atrocities. [Online; accessed 2023-10-04] https://www.thestar.com/news/world/2012/03/14/how_a_syrian_refugee_risked_his_life_to_bear_witness_to_atrocities.html (2012)

45. The Shufflecake Project: The Shufflecake Website. [Online; accessed 2023-10-04] https://shufflecake.net (2022)

46. The TrueCrypt Foundation: TrueCrypt Homepage. [Online; accessed 2023-10-04] https://www.truecrypt71a.com/ (2015)

47. United States Court of Appeals, Eleventh Circuit: UNITED STATES OF AMERICA v. JOHN DOE, Nos. 11–12268 & 11–15421 - D.C. Docket No. 3:11–mc–00041–MCR–CJK. [Online; accessed 2023-10-04] https://caselaw.findlaw.com/court/us-11th-circuit/1595245.html (2012)

48. United States v. Fricosu: United States v. Fricosu — Wikipedia, The Free Encyclopedia. [Online; accessed 2023-10-04] https://en.wikipedia.org/wiki/United_States_v._Fricosu (2022)

49. VeraCrypt: VeraCrypt Homepage. [Online; accessed 2023-10-04] https://www.veracrypt.fr/en/Home.html (2022)

50. Ward, M.: Campaigners hit by decryption law. [Online; accessed 2023-10-04] http://news.bbc.co.uk/2/hi/technology/7102180.stm (2007)