Hidden Filesystem Design and Improvement

Master's Thesis in Cybersecurity

Elia Anzuoni

Supervised by:
Edouard Bugnion (DCSL EPFL)
Tommaso Gagliardoni (Kudelski Security)

Academic year 2021/2022

Acknowledgements

The outcome of this Master's Project would have been very different without the oversight of my university supervisor, Professor Edouard Bugnion: I want to especially thank him for his constant and vigilant involvement, and for the precious advice and insight he was always able to give me, all throughout the stages of this journey.
Special thanks also go to my company supervisor, Tommaso Gagliardoni, who came up with the research idea in the first place and closely monitored its gradual unfolding, tirelessly guiding me from the initial acquainting phase, through the design and development of the main work, to the crucial final refinements.
Naturally, this project would not have been possible had it not been for the aid of my company, Kudelski Security, who generously fostered it as a paid internship. My heartfelt thanks to all the people in the Research Team, who constantly supported me and created a perfect environment for my personal and professional growth.

Abstract

Plausible deniability is a strong cryptographic security property which, in the context of data storage, offers protection against invasive and coercive adversaries who have the power to extort the passwords to encrypted data. Concretely, to defend against such threats, the storage must be formatted in such a way that multiple passwords can be used to access it, some disclosing innocent data, others unlocking secret contents: the user can then only reveal passwords from the first set to the adversary, and *plausibly deny* that more exist.

This problem has collected the interest of the computer security community for some time now. Several schemes have been devised over the last two decades, offering various levels of performance and security. Some of them, achieving very good performance, arguably do not provide satisfactory guarantees of deniability. Others, accomplishing full, bulletproof security, suffer instead from severe performance hits (especially in terms of I/O overhead and disk space utilisation).

We propose a novel design, a scheme called Shufflecake, which targets a more balanced compromise between peformance and security. The level of deniability it offers, while not protecting against attacks in the most stringent threat model, is sufficient in many practical scenarios. On the other hand, it achieves a 99.6% disk efficiency, and a 1x-3x slowdown over regular disk encryption tools, which makes it suited for real-world applications.

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 almost by default entrusted to an online third party, the last bastion of data confidentiality is local storage, as the physical disconnection greatly aids 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 have been devised over time.

In this work, we focus on a different, not-as-common threat scenario, that is less extensively covered by the existing literature. The context is still that of a person keeping sensitive files on a disk, but the adversary does not just steal it to gain "offline" access to it: they are in a position of power which they use to directly and aggressively confront the user about the contents of the disk, and, by means of (physical, legal, psychological) coercion, they can obtain the encryption keys to any encrypted content identifiable on it. The goal then becomes to still retain secrecy of some select files on the disk, by making them not even identifiable, thus allowing the user to make up a credible lie about the storage contents.

Albeit relatively unusual, this threat model adequately represents 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 [46, 34, 48, 30, 59], 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 [61, 7, 15, 4]. 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: the precise extent to which, in countries with a less-developed system of checks and balances, the sensitive data of activists and dissidents are violated, can only be hinted at by the sporadic cases that occasionally make it to the international headlines [14]. Finally, journalists investigating on vast criminal organisations are also likewise exposed to potential tense confrontations with coercive adversaries, who are interested in the data hosted on their storage devices, and have the power to force them to reveal the corresponding encryption passwords.

The aim of this Master's Project has been to study relevant solutions to properly secure data, in the context of local storage, under coercive or forensic threats.

1.1 General Landscape

The computer security community has long been interested in similar problems, and involved in the design of adequate protection measures. For example, disk encryption [24] is a popular and well-studied solution that is designed to protect some user's data at rest (i.e., saved on persistent storage) from unauthorised attempts to access it. This is usually accomplished by means of a password that is memorised and kept secret by the user: the data on the disk is encrypted with a key derived from the user's password.

This capability has been available in commercial products for over 20 years, and it is now offered as a standard software component in the most widespread operating systems (BitLocker [41] for Windows, LUKS [45] for Linux, etc.). Given all the experience that has been collectively accumulated on this subject, and the severe tests that the existing solutions have resisted over the

course of many years, it can safely be said that disk encryption is a solved problem, at least in ordinary threat scenarios, like a "simple" thief stealing the user's laptop while turned off and trying to read its contents.

Although many of today's schemes do a good job at protecting the user in the situations they were originally designed to address, real-world threats do not stop there: new attacks, that exceed the initial threat model, become more and more relevant over time.
For example, consider the hypothetical case of a journalist in an authoritarian state, who wishes to conduct some investigation on government corruption, and keeps all related files on a disk; the danger they face is indictment before a judge if these files are discovered. The journalist cannot rely on traditional disk encryption techniques, because they are subject to invasive searches by the authorities whereby their disk might be seized and forensically inspected at the physical level, and they might be aggressively interrogated in response to the outcome of this inspection; in particular, they might be coerced to surrender the encryption password if anything on the disk points towards the existence of encrypted content.
This completely breaks the security of traditional solutions, which hide the *content* of the data, but not its *existence*: for example, they usually leave some headers in clear containing some parameters that unambiguously identify the device as being, e.g., a LUKS partition. Therefore, even though the police may not be able to *break* the strong encryption used to protect the data, it is enough for them to *identify* the encrypted content and use that evidence to extort the encryption key.
As was previously mentioned, this is the kind of strong adversarial situation we will focus on in this work.

Quite clearly, a user needs stronger security guarantees than those offered by simple disk encryption against such invasive adversaries, who have the power to circumvent it by means of coercion. For about 20 years now, providing security in such severe threat scenarios has been the mission of commercial software products first [57, 23], and of more rigorous academic research later [19].
Although these efforts led to fairly diverse problem statements and concrete solutions, they all adhere to the same broad idea of *plausible deniability*: the data must be hidden in such a way that multiple keys are associated to its access, some of them revealing innocent decoy data (and that can be safely surrendered to the adversary), while the others unlock the actual secret information (whose mere existence is consistently denied). The security of the scheme must, of course, ensure the plausibility of this lie: no trace must be left on the disk - even after decryption of the decoy content - that hints towards the existence of more encrypted data. Usually, this formally translates to requiring some form of indistinguishability between a disk where only the decoy data is present, and one that also contains some secret, yet-unrevealed content.
Immediately from this same property, an implication follows on the actual security provided by such a scheme. All the adversary will ever be left with, once the decoy content is revealed, is the *impossibility to prove* that further data is present, not the *solid conviction* of its absence (it would be hard to mathematically convince them of a false statement, after all). What this means is that, judging by the disk contents, there could always be more unrevealed passwords - even after they have all truly been disclosed - and there is no way to conclusively rule out that possibility. Therefore, a user resists adversary pressure to disclose more passwords on the belief that they have no way of proving that more exist, and that they could never be totally satisfied anyway, even if they were all truly handed over.

This is likely enough in environments where the burden of proof rests on the prosecutor; however, in tenser situations, the user might have to "give them something" by including some mildly-incriminating content (like illegally-downloaded films) among the decoy data, so as to stop coercion by reasonably convincing the adversary that they have in fact given up what they were hiding. It is worth noting that, although the user is still made to go through some discomfort, this is the best that can be done. In general, the more severe a threat scenario, the more compromised a person already is, the more attention they will have to pay and the more they will have to endure in order to avoid complete defeat: all a protection scheme can do is help them in that regard, but some "collaboration" is inevitably required.

Plausible deniability was first concretely offered as an additional optional feature in early open-source disk encryption tools. The paramount example of such software, that is usually taken as a historical milestone in the field, is TrueCrypt [57]. First released in 2004, it went through a troubled development history which culminated in the project being abruptly discontinued in 2014 amid worries it might have been backdoored by the NSA [27]; a crowdfunded security audit later found no evidence of such tampering [5].
Although TrueCrypt attained widespread adoption and appreciation among the general public, it had serious transparency problems that partially crippled its trustworthiness [27]. Most notably, it was not classified as free software (despite being open source) because of its peculiar, non-standard licence that left many legal details unclear [56]; it also had severe issues with build reproducibility, to the point where the source code of version 7.0a for Windows could not be compiled to exactly match the distributed binary [28]. Additionally, it never gained much traction in the Linux community, mainly because some desirable functionality (encrypting an entire OS) was not available on Linux for technical reasons. Finally, while the core disk-encryption part was never found to be vulnerable to impactful attacks, its plausible-deniability guarantees were comparatively weak.
This served as a starting point for dedicated academic research, that specifically tackled the problem of plausible deniability from a rigorous point of view, by properly defining it with formal cryptographic methods, and by devising ad-hoc schemes that provably offered security even in very severe threat models. Since this is still a relatively young and marginal research area, though, the relevant academic literature is currently not plentiful: only a handful of papers provide some substantial contribution, either in the form of new schemes [12, 47, 65, 18, 16, 17], or through some theoretic systematisation or advancement [35, 6, 19]. Moreover, since a fairly solid link has been established [19] between plausibly-deniable storage (in the most severe threat scenario) and a separate cryptographic primitive (Write-Only ORAMs [35]), most of the research efforts have been directed at devising new schemes for this primitive, almost completely neglecting alternative directions. While the "full" security offered by ORAMs is certainly desirable, they invariably come with great performance hits [19], often in terms of I/O overhead and disk space efficiency, which greatly limit their real-world adoption.

1.2 Our Contribution

The objective of this Master's Project was to first survey the research field of plausibly-deniable storage to get an idea of the current state of the art and of the history that brought us there, and then to identify some gaps in the current landscape for us to possibly bridge with an original contribution. Indeed, the summary given in the previous section clearly hints at some interesting room for innovation: what is missing is a middle ground between TrueCrypt and Write-Only

ORAMs, providing a more balanced compromise between performance and security.

We designed a novel scheme for plausibly-deniable storage, precisely with this objective in mind; we called it Shufflecake.
Not being a Write-Only ORAM, it does not achieve "full" provable security: it is proven secure in a relaxed threat model, and can only achieve "operational", unproven security in the most severe one. In exchange for reduced (yet still acceptable) security, it achieves a great performance gain over ORAMs: it is only 1x-3x slower than the baseline (`dm-crypt` [36], the standard, regular disk encryption utility for Linux), and it utilises 99.6% of the underlying disk space.
It generally builds on ideas from TrueCrypt, fixing many of its limitations. Compared to it, Shufflecake offers better security: it supports several hidden volumes, it is filesystem-agnostic (making full Linux support possible and decoy volumes more plausible), and offers some degree of unproven multi-snapshot security.
Finally, an implementation of Shufflecake in C is provided as a device-mapper target for the 5.13 Linux kernel. It is released under GPLv3, hence being Free and Open-Source Software (FOSS). Benchmarks confirm the very good performance and immediate possibility of real-world adoption.

1.3 Report Layout

The next chapters are organised as follows.
Chapter 2 covers the required technical preliminaries. First, it offers a brief summary of basic cryptography. Then, it gives an overview of the necessary building blocks for a scheme offering plausibly-deniable storage.
Chapter 3 illustrates in depth the security notion of plausible deniability itself, both from a cryptographically-rigorous standpoint, and through the example of the earliest (and simplest) scheme offering it as a feature: TrueCrypt.
Chapter 4 covers Shufflecake, our original contribution to the field, in detail. It explains in depth the theoretical scheme itself, it gives some details on the implementation, it presents the benchmarks carried out against `dm-crypt` and against ORAM-based solutions, and it outlines some future research directions.

2 Technical Background

This chapter provides a quick summary of cryptography and its cornerstone topics along the road leading up to our area of interest, which is plausible deniability (treated in detail in Chapter 3). It can safely be skipped by the experienced reader.

2.1 Cryptography

Cryptography is a theoretical discipline in computing, focusing on the design and on the formal analysis of ideal schemes assuring that the interaction between malicious parties (*adversaries*) and honest parties (*users*) does not result in detrimental outcomes for the users, no matter what strategies and computations the adversaries employ. In each specific context, concrete instantiations of this idea specify the interaction model, define what qualifies as a detrimental outcome for the users, and place constraints on the power the adversaries have in choosing their malicious behaviour.

Up until the first half of the twentieth century, cryptography was only ever really concerned with preserving the secrecy of confidential data, primarily in military contexts. This would usually amount to ciphering sensitive messages according to some more or less intricate procedure, only known to the sender and the intended recipient, leaving nothing but incomprehensible garbled text to an enemy intercepting the ciphered message without having the secret code.
Since then, massive progress has been made, both in depth and breadth: better algorithms have been developed to preserve data confidentiality, and different cryptographic primitives altogether have been introduced. Also, the whole discipline has undergone a thorough formal systematisation, effectively going from being an art to being a science with its own rigorous mathematical methods.
The following sections introduce the fundamental concepts of modern cryptography, and quickly go over some of its most important primitives.

2.1.1 Fundamental Concepts

Cryptography is mostly about preventing certain actions to an adversary while allowing them for the honest user. To have any hope in that regard, there must be some a-priori asymmetry between the two parties: if the adversary has exactly as much information as the honest user, nothing can prevent them from performing the exact same steps as the honest user. Therefore, there must be some secret that the user has and the adversary does not have, and cannot (efficiently) compute with the data they manage to gather.
Modern cryptographic algorithms are tasked with amplifying this initial asymmetry, to achieve the desired security property based on the secrecy of this value (that we can call a *key*).

Kerchoff's principle This is a very high-level guiding principle for the design of cryptographic systems, which can be simply considered as glorified common sense. It is followed by most of the systems currently in existence: exceptions do exist, but are heavily frowned upon.

The principle simply states that the security offered by a cryptographic scheme should not rely on the secrecy of the algorithms it employs, but only on the secrecy of the key (as defined above, which is just an **input** to the algorithms).

The rationale behind this principle is that the algorithms themselves cannot be trusted to remain unknown to a real-world adversary for a long time: to give just one example, suffice it to think of the Enigma machine, used by Germany in WWII to encrypt confidential messages; one of these machines eventually fell in the hands of the Allied powers, who could then study its internals, identify some of its flaws, and break its security.
A public-domain cryptographic scheme, instead, undergoes severe scrutiny by multiple, independent, skilled eyes: the arguments for its security are then usually phrased as "no practical and impactful attack against this scheme is *yet* known to the public community", which is a fairly solid guarantee in itself, and certainly more that can be said about "home-made" schemes.

To sum up, the security arguments for a sensibly-designed cryptographic scheme should assume that the adversary knows about the scheme, and is able to execute its algorithms with custom inputs. This work is no exception.

Game-based security definitions As previously mentioned, cryptography has long become a scientific discipline with its own mathematical methods. In particular, the precise definition of a security property is often given in the form of a *game*, formalising the interaction taking place between the adversary and the victim, as well as the general context under which the interaction takes place (e.g., public values).
The game also defines the *advantage* of the adversary when playing against it, which is a quantitative measure of how well the adversary performed at the end of the game: a cryptographic scheme will then be said to be secure (under the definition given by the game), if every adversary (limited to polynomial-time computations) achieves a low advantage (for a suitable definition of "low") when playing the game.
The requirements on the complexities of the algorithms executed by the user and the adversary are often expressed in asymptotic terms, controlled by a single *security parameter* called λ, which implicitly tends to infinity. The *functional* requirements on the algorithms of the cryptographic scheme itself mandate that their running time be polynomial in λ; in particular, this implies that all of their inputs (including the key) must have a size polynomial in λ. The *security* requirements on the scheme mandate that, no matter what polynomial-time (in λ) computations the adversary performs, his advantage in the game must be a *negligible function* of λ (i.e., it has to be $o(\lambda^{-k})$ for any $k \in \mathbb{N}$).

Adversary goal and power There is a distinction to be made between a loosely-defined security property like "data confidentiality", often approximately described in plain human language, and a security property formally described by a game, like IND-CPA: there is a one-to-many relationship between these concepts. We can loosely define data confidentiality as the property of keeping data unreadable to adversaries, but that translates to a wealth of possible concrete game-based security definitions, all matching the broad idea of confidentiality.

There are two main axes along which a game can move to modulate the exact security property it intends to describe: the adversary power and the adversary goal.

The adversary power defines how invasive the adversary can be in its interaction with the victim. For example, in the context of data confidentiality, the adversary might only have access to a series of ciphered messages (COA, ciphertext-only attack), or he might have access to some sort of

engine (usually called an *oracle*) that ciphers arbitrary messages and yields the result (CPA, chosen-plaintext attack). Of course, the adversary is at least as strong in the second scenario as in the first, since whatever they can do in the COA case, they can also do in the CPA case.
In general, the stronger an adversary, the stronger the security notion implied by the game (a scheme that is secure under that game can resist strong adversaries).

The adversary goal defines what an adversary has to do in order to achieve a high advantage. For example, again in the data confidentiality scenario, the adversary might content themselves with being able to distinguish a ciphered message from a random string (IND, for "indistinguishability"), or they might have to actually decrypt a ciphered message (DEC, for "decryption"). Of course, an adversary with high advantage in the DEC game also achieves a high advantage in the IND game, since the ability to decrypt ciphered messages clearly implies the ability to tell them apart from random strings.
In general, the lower the adversary goal, the stronger the security notion implied by the game (a scheme that is secure under that game protects even against "low-impact" attacks).

Data at rest vs data in flight Many data-protection primitives and concepts can be applied to two different, broad classes of contexts: data at rest, and data in flight.

In the first case, the user needs to store data on some vulnerable device (e.g., one that can be stolen) for later retrieval. The key used for protecting the data must obviously be kept off that device, in a secure storage area inaccessible to the adversary (which could be the user's brain memory).
This might look like a circular definition, but a crucial observation is that the key is much smaller than the data itself: therefore, we amplify the security of a **small** storage area to protect a much larger, insecure storage area.

In the second case, the user needs to send data to a recipient over an insecure network (e.g., the adversary could read and modify packets). The key used for protecting the data must be obviously kept away from the adversary, and pre-shared between the parties over a confidential channel.
This might, again, look like a circular definition, but the same observation comes to rescue: the channel for the key establishment can have a much lower **bandwidth** than the one used to transmit the data (e.g., keys could be exchanged by hand, in person), since the key is much smaller than the data to be transmitted. We then amplify the security of a narrow-band channel to protect a wide-band, insecure channel.

For many primitives (data authentication, data confidentiality, etc.), the game-based security definitions arising in these two contexts are exactly the same, since data at rest can be seen as a special case of data in flight, where the recipient is the sender himself, and the adversary interacts with the storage device instead of the network. A notable exception is plausible deniability: for this primitive, the quirks of the two scenarios are too important, and as a result the respective games are completely different. We will only see the game in the data-at-rest scenario, which is the one that concerns us.

2.1.2 Cryptographic Primitives

As was mentioned, modern cryptography has not only improved on the old schemes for data confidentiality, but also defined new threat scenarios that call for new protection primitives. This section informally describes some of them.

Data confidentiality This primitive has previously been roughly described as "making data unreadable to adversaries". This is effectively achieved by substituting the original data, called the *plaintext*, with a garbled, *encrypted* version of it, called the *ciphertext*, obtained by mixing the plaintext with a *key* using an *encryption algorithm*. In formulas, we have (with the obvious notation)

$$\mathsf{ct} = \mathtt{Enc}(k, \mathsf{pt}).$$

The same key can then be used, in conjunction with the ciphertext, to recover the original plaintext, by using a *decryption algorithm*. In formulas,

$$\mathsf{pt} = \mathtt{Dec}(k, \mathsf{ct}).$$

In other words, $\mathtt{Enc}(k, \cdot)$ and $\mathtt{Dec}(k, \cdot)$ are each other's inverse functions for any choice of the key k. For brevity, we do not discuss *asymmetric-key* encryption schemes, where the encryption key is different from the decryption key. Such schemes are also known as *public-key* encryption schemes, because the encryption key is public and specific to a receiver (anyone can encrypt messages for a specific user, by using their public key), whereas the decryption key must be kept private by the user [51]. In our scenario (*symmetric-key* encryption schemes), the key is only one, it is shared between sender and receiver, and must be therefore kept secret by both.
The security definition takes care of mandating (in some specific flavour) that an adversary without the key cannot decrypt ciphertexts.

A possible use scenario of this primitive is disk encryption [24]. This usually works by encrypting and decrypting data "on the fly", at an intermediate layer between the user and the disk, with a key that is kept off-disk (or possibly generated through a user-defined password). On-the-fly encryption and decryption allows the user to transparently use the disk as though nothing were happening in the middle, while only ever leaving encrypted data on the disk. This, in combination with the fact that the key is not accessible from the disk, protects against an adversary that might, for example, steal the disk and try to read its content.

Data authentication This primitive is somewhat complementary to data confidentiality: instead of making data unreadable, it aims at making data non-writable. To be clearer: in the data confidentiality scenario, the adversary has read access to the **protected** (encrypted) data, and the security of the scheme ensures that this does not result in read access to the original data; in the same way, in the authentication scenario, the adversary has write access to the protected (authenticated) data, and the security of the scheme ensures that this does not result in write access to the original data.
What this means is that the adversary clearly has the possibility to modify the authenticated data: the best we can hope for is **detect** this modification and **reject** the data. This is still good protection, as an adversary's goal would be not just to disrupt the user's data (which they can do anyway, and cannot be prevented), but to modify it in a smart way so that the user unknowingly

accepts it and is thus tricked into doing something useful for the adversary.

Protection against such attacks is often concretely achieved by juxtaposing an *authentication tag*, or MAC (*message-authentication code*) to the original data [11]. The tag is generated by mixing the message and the key using a *MAC algorithm*. In formulas,

$$t = \texttt{MAC}(k, m).$$

Upon reception of a message, the user accepts it only if accompanied by a valid MAC, i.e., based on the outcome of a *verification algorithm*. In formulas,

$$\texttt{Verify}(k, m, t) \rightarrow 0/1.$$

The security definition roughly ensures that authentication tags are *unforgeable*, i.e., cannot be generated without the key. To be precise, this does not just assure that the message was not tampered with and was received as it was sent (this is a simpler property called *data integrity*), but also that it was indeed sent by someone holding the same key k as the receiver (hence *data authentication*).

An application scenario of such a primitive is a communication session over an insecure network, between two parties sharing a common secret key, in the presence of an adversary that can intercept and modify the network traffic at will. The two parties might be, for example, a power plant and its remote controller issuing commands to it: if they employ no authentication on the transmitted data, the adversary could modify the packets being sent to the power plant, who would accept them and execute whatever command the adversary injected.
Adoption of message-authentication codes, with the use of the shared secret key, would thwart such an attack, because adversary-crafted packets could not be tagged with a valid MAC (because of the security property of the scheme), and would thus be rejected by the power plant.

Plausible deniability Plausible deniability also aims at protecting the secrecy of user data, but is a strictly stronger primitive than data confidentiality as defined before, as it aims not just at hiding the *content* but the very *existence* of the data (often times, no effort is made in data confidentiality schemes to conceal the fact that some encrypted data is present).
It is appropriate in situations where the adversary is a coercive one (e.g., the police) that might not only seize the user's disk, but also force them to surrender any secret they might be reasonably suspected to have (e.g., encryption keys). The goal then becomes for the user to be able to *plausibly deny* that the data even exists, for example by encrypting it and making it indistinguishable from some random background noise that "would be there anyway", with or without the user's data.
This primitive is much more deeply discussed in Chapter 3.

Hash functions This is a very common primitive, used to reduce arbitrary-length bitstrings to unpredictable fixed-length bitstrings [52]. Concretely, they are **deterministic** functions that map strings m of arbitrary length (or of length up to a certain, very high limit) to a random-looking string $H(m)$ of fixed length, e.g., 512 bits.
The security of a hash function is hard to formally define, but it intuitively mandates that it be "one-way", that is, it must be hard to find a message that hashes to a given hash. A stronger security property is "collision resistance", or the hardness to find any two messages with the same

hash.
Hash functions are normally used as a building block for other primitives: one such example is key derivation, which we will see next.

Key-derivation function Usually abbreviated as KDF, this primitive is used to derive high-entropy cryptographic keys starting from low-entropy, user-memorised passwords [63]. It is basically a hash function that takes two inputs: a secret, user-memorised, low-entropy password, and a public, random, high-entropy *salt* (a fixed-length bitstring); the output is a fixed-length bitstring that is safe to use as a cryptographic key.
Since passwords are low-entropy (and salts are public), bruteforce attacks are definitely a possibility. In order to mitigate them, a special sort of hash functions is needed, called *memory-hard* functions: they have the same properties as regular hash functions, but are also intentionally designed to require relatively large amounts of memory (e.g., 1 GiB) for their computation. Additionally, they are iterated several thousands or millions of times, in order to make their computation time-intensive too (e.g. several milliseconds, or a few seconds, on common CPUs). The rationale behind this deliberate performance degradation is the search for a compromise between usability (users will only seldom need to perform these operations) and security, by making every trial in a bruteforce attack so expensive that even low-entropy passwords cannot be cracked.
The role of the salt is to thwart *pre-computation attacks*, whereby an attacker goes through the pain of expending lots of memory and computing time to calculate the hash of many possible passwords, but does so just once, before "meeting" the users and efficiently checking every password hash against each one of them. This way, the attack scales by amortising the heavy cost over the vast set of victim users, all of which are affected by the one pre-computation carried out by the attacker. Using a salt frustrates this attempt, forcing the attacker to really perform the heavy computation individually for each user.

2.2 Confidentiality

This section examines in some depth the security notion of data confidentiality, and presents various concrete games associated to it as well as some well-established solutions.

2.2.1 Perfect Secrecy

A seminal work [50] published in 1949 by Claude Shannon, the father of information theory, laid the foundations of modern cryptography with its formal analysis of data confidentiality that proved some fundamental bounds on what can be achieved. His framework is now outdated, as it is unable to capture a wide spectrum of nuanced security notions; nonetheless, his work and results are still relevant, and also constitute a good gateway into the subject.

Security definition Throughout his work, Shannon stuck to the case of data in flight, i.e., with one party (Alice) having to send a string of ℓ bits to another party (Bob). The parties have an insecure channel over which to send encrypted data, that can be read by an adversary (Eve), and a secure, confidential channel over which to send keys, which is inaccessible to Eve.

The security notion sought by Shannon was one of *perfect secrecy*, aiming at making it mathematically impossible (not just computationally infeasible) for the adversary to extract any

information about the plaintext, given the ciphertext.
For this to make any sense, there must be some a-priori uncertainty about the message that Alice is going to send: otherwise, if Alice always sends message $m \in \{0,1\}^\ell$, then there is no point in sending it to Bob, since he already knows it. What this means is that the message sent by Alice follows a certain *probability distribution*: call $M \in \{0,1\}^\ell$ the (random) message coming from this distribution. Also, suppose that Eve knows exactly this distribution $\mathbb{P}(M)$, but not (of course) the actual message M sampled from it by Alice and sent to Bob.
Call $C \in \{0,1\}^\ell$ the (random) ciphertext sent over the insecure network and intercepted by Eve. After reading the ciphertext, Eve can compute the a-posteriori distribution $\mathbb{P}(M|C)$, representing the new idea that she has about the plaintext now that she has collected the additional information given by the ciphertext.
Shannon's perfect secrecy definition requires that this additional information given by the ciphertext be 0. This formally translates to requiring that the a-posteriori distribution $\mathbb{P}(M|C)$ be the same as the a-priori one $\mathbb{P}(M)$. Equivalently, we can impose that the random variables M and C be independent. This has to hold irrespectively of the distribution $\mathbb{P}(M)$, because the encryption scheme has to be secure no matter which plaintexts are more likely.

This turned out to be a very stringent security definition: Shannon did indeed come up with a very simple scheme that offers this level of protection, but its drawbacks are too severe in most applications. He also proved that any scheme providing perfect secrecy has the same drawbacks.

One-time pad Shannon's scheme simply worked by having Alice sample a key K uniformly at random from $\{0,1\}^\ell$, send it to Bob over the confidential channel, and compute the ciphertext $C = K \oplus M$ (bitwise XOR) to be sent over the insecure channel. Bob would then decrypt by computing $M = C \oplus K$.
This scheme achieves perfect secrecy because the random variable C is uniformly distributed over $\{0,1\}^\ell$, no matter the value of M: thus, it is independent from M. C is uniform in $\{0,1\}^\ell$ because XOR-ing with a given M is a bijection (mapping K to C): since every K is equally likely, every C is equally likely (M just acts as an offset).

Although the security achieved by this scheme is perfect, its drawback is clear: for every message of length ℓ to be sent to Bob, Alice needs to send ℓ bits of key through the confidential channel. For this reason, this scheme is called *one-time pad*: a new key, exactly as long as the message, has to be generated for each message, and then discarded after the first use. Indeed, reusing a key K for two messages M_1 and M_2 leads to a very severe leak: the adversary can deduce the XOR of the messages using the ciphertexts, by computing $C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K) = M_1 \oplus M_2$.
Thus, the confidential channel needs to have as much bandwidth as the insecure one. In many settings, this removes any need to use a cryptographic scheme at all: we could just as well send the message itself, in the clear, through the confidential channel. The only scenario where the OTP retains applicability is one where the confidential channel is available at a certain time (e.g., exchanging by hand a USB stick filled with random data), but not at a later time, when communication has to take place [13]. For example, this is the case of nuclear launch codes, or the channels used for state-level espionage.

Necessity of long keys Shannon was able to prove that any scheme offering perfect secrecy would suffer from the same drawbacks as the one-time pad.

He carried out an information-theoretic argument to prove that, from the security property itself (i.e., that $C = \texttt{Enc}(M, K)$ has to be independent from M), it follows that the distribution of K needs to have at least as much entropy as the distribution of M. Since this needs to hold for any distribution of M, it also has to hold for the distribution that maximises the entropy, that is, the uniform distribution (over $\{0,1\}^\ell$). Thus, the distribution of K needs to have entropy ℓ, so K must be uniform over $\{0,1\}^\ell$.

2.2.2 Stream Ciphers

The fundamental reason why schemes offering perfect secrecy are so impractical is that they must leave no residual information whatsoever in the ciphertext. We can go a long way by relaxing this constraint, and only requiring computational infeasibility, instead of information-theoretic impossibility.
Modern encryption schemes can be divided into two broad classes: stream ciphers, and block ciphers. Here, we discuss the former: they emulate the behaviour of the one-time pad in that they still encrypt and decrypt by means of a simple XOR, but perfect secrecy is traded off for actual usability [49].

Intuitive definition The abovementioned trade-off is achieved by deterministically generating, instead of randomly sampling, the bitstring to be XOR-ed with the message M, called the *keystream*.
Concretely, a stream cipher is a finite-state automaton, with no input and an output; the initial state needs to be shared between Alice and Bob, and unknown to Eve: it will be the only initial asymmetry between victim and adversary. When the automaton is "ticked" it advances its state and produces an output word; Alice and Bob then do this repeatedly to generate a long-enough keystream.

One must observe that the automaton is deterministic, so it always generates the same keystream if fed with the same initial state. This is an issue, because if two messages M_1 and M_2 are encrypted with the same keystream T, then the ciphertexts C_1 and C_2 leak the XOR of the plaintexts, since $C_1 \oplus C_2 = (M_1 \oplus T) \oplus (M_2 \oplus T) = M_1 \oplus M_2$.
To avoid this, the initial state S_0 of the automaton must be both secret and fresh: for this reason, it is usually generated by mixing a secret value (the shared secret key K), that does not need to be fresh and can stay the same across encryptions, and a fresh value (usually called the IV, for Initialisation Vector), that does not need to be secret and can be sent in the clear alongside the ciphertext. Of course, IV reuse is a serious error, that again leads to keystream reuse.
The high-level idea is depicted in Figure 1.

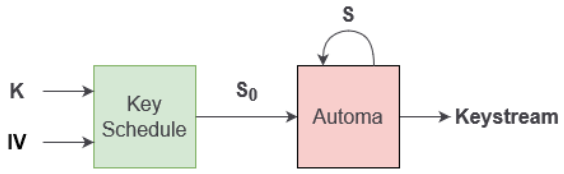


Figure 1: High-level functional diagram of a stream cipher

Intuitive security property We will talk about encryption security more formally in a following section; however, specifically for stream ciphers security games, the adversary usually has access to a portion of the keystream, and its goals can be (in descending order of hardness, so in ascending order of security) to recover the secret key, to recover the internal state of the automaton, to predict a future output of the automaton, or just to distinguish the keystream from a random string.

2.2.3 Block Ciphers

Block ciphers take a different approach to the problem: they restrict themselves to the domain of bitstrings of fixed size (say n), called blocks [9]. The encryption and decryption algorithms `Enc` and `Dec` map $\{0,1\}^k \times \{0,1\}^n$ to $\{0,1\}^n$, where k is the key size and n is the block size.
The functional requirement is that

$$\texttt{Dec}(K, \texttt{Enc}(K, X)) = X \qquad \forall\, X \in \{0,1\}^n, \forall\, K \in \{0,1\}^k$$

This means that, for any choice of the key $K \in \{0,1\}^k$, the function $\texttt{Enc}(K, \cdot)$ is a permutation over $\{0,1\}^n$, and $\texttt{Dec}(K, \cdot)$ is its inverse permutation. This is shown in Figure 2.

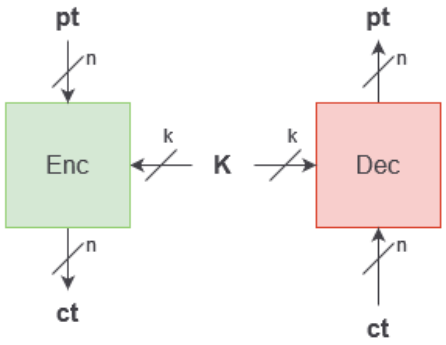


Figure 2: High-level functional diagram of a block cipher

We can think of the keys as indexing the 2^k available permutations (out of the $(2^n)!$ that exist in total). The security property can then be roughly expressed as: by choosing the key K uniformly at random, the resulting permutation is indistinguishable from a random permutation, i.e., it leaves almost no correlation between input-output pairs.

AES Many block cipher schemes have been designed over the years: the first to reach widespread success and to be adopted as a standard was the Data Encryption Standard [20], in the 1970s. To give some concrete numbers, its block size was 64 bits and the key size was 56 bits: this was sufficient at the time (despite claims about the NSA having interfered with the design to intentionally weaken it [26]), but with today's computing power it would not be hard to mount a bruteforce attack to recover the secret key.

Towards the end of the 1990s, the U.S. National Institute of Standards and Technology opened a public competition to select the successor of DES for encryption of government sensitive data. The winner, an algorithm called Rijndael, was selected in 2001 and announced as the new Advanced

Encryption Standard (AES), which is the name it more commonly goes by today [29].
We do not cover AES in any detail here, but we give some relevant figures and information. The block size is 128 bits, while the key size can be either 128, 192, or 256 bits. It is very efficiently implemented in hardware as a single CPU instruction on many platforms, including x86. Its security has been the object of intense research by the public community for the last 20 years, and yet no practical attack has yet been found. As a result, AES today is one of the most widespread encryption schemes.
In the implementation of our plausible deniability scheme, we use AES-256 (i.e., with 256-bit keys) as the underlying block cipher.

Modes of operation The restricted perspective adopted by block ciphers, i.e., fixed-size inputs, makes them somewhat easier to design and study, but leaves open the question of how to encrypt messages longer than a block. This is where modes of operation come to rescue: they are procedures that split a long message into blocks (usually by padding it beforehand to reach a multiple of the block size), and then encrypt the sequence of blocks in some way by employing the block cipher as a subroutine [54].
We now see some of them.

The first, most basic, and most insecure one, is ECB: electronic codebook. It works by simply encrypting all the blocks separately. Figure 3 depicts the idea.

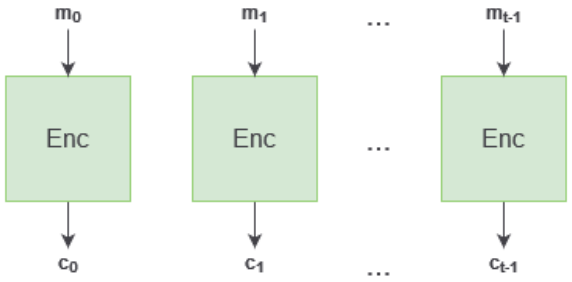


Figure 3: ECB mode of operation

Figure 4, showing an image of Tux (the Linux mascot) encrypted with ECB, illustrates why this mode of operation is irremediably insecure: if two blocks in the message happen to be the same (which can very much be the case, as for pictures where the blocks contain information on pixel colours), this equality is leaked by the ciphertext, no matter how good the underlying block cipher is.

The second mode of operation we discuss is a very popular one, called CBC: cipher-block chaining [8]. It works by XOR-ing a plaintext block with the previous ciphertext block, before passing it through the block cipher. This solves the problem of ECB, as now there is no way, just by looking at the ciphertext, to tell whether two blocks of the message are the same.
The first plaintext block is XOR-ed with an IV, that acts as a c_{-1}.

As with stream ciphers, the IV needs not be secret, but it has to be fresh for each encryption. In fact, almost all modes of operation need an IV (except for ECB) that ensures that successive encryptions of the same plaintext do not result in the same ciphertext. For this reason, an IV is

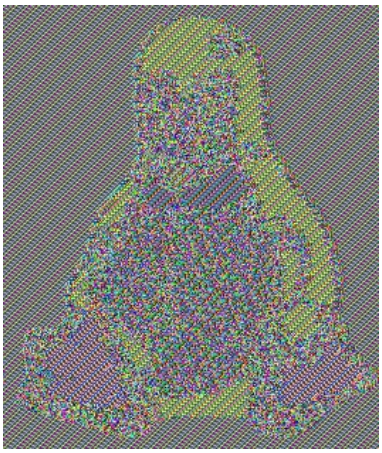


Figure 4: ECB preserves equality of blocks within the message

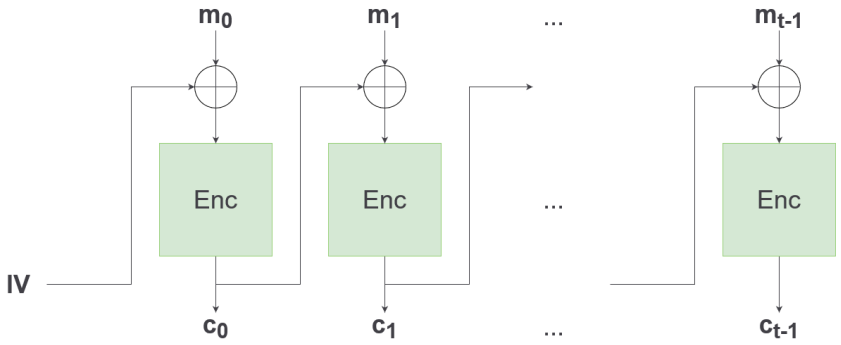


Figure 5: CBC mode of operation

sometimes called a *nonce*, for "Number used ONCE".

The last mode of operation we see is called CTR: counter mode [38]. It basically turns the block cipher into a stream cipher, by successively encrypting the IV and its successors (treating it as a number), and using that as a keystream. Figure 6 shows how it works.

Because of its design, it is inherently parallelisable: no previous ciphertext block is needed to compute the next one.
It is the mode of operation that we will use in our plausible deniability scheme, paired with AES.

The counter mode also serves as a basis for other, more advanced and very widespread modes of operation, like CCM [21] and GCM [40], which provide *authenticated encryption* (i.e., confidentiality and authentication together).

2.2.4 Security Games for Block Ciphers

We now see a couple of game-based security definitions, specifically framed in the case of block ciphers. Starting in this simpler scenario allows us to focus on the core concepts of security games: the context implied by the steps of the game, the oracle with which the adversary can interact, and

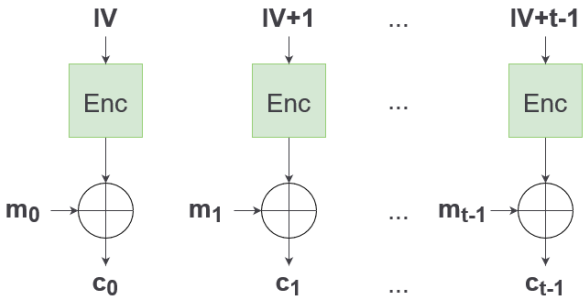


Figure 6: CTR mode of operation

the advantage as a measure of how much a given adversary "breaks" the scheme.

KR-KPA The so-called *Key Recovery under Known-Plaintext Attack* is the first example of security game we see. We start with this since it is very easy to describe.
It is one of the hardest ones to win for an adversary, and consequently conveys one of the weakest security notions. The adversary power is very low: it only has access to a list of plaintext-ciphertext pairs. The adversary goal is, on the other hand, very ambitious: it aims at recovering the secret encryption key. The security implied by this game is consequently very weak, because the impossibility to win it only reassures against catastrophic breakdowns, but gives no guarantees in more realistic scenarios.

Game $\Gamma^{\mathcal{A}}$: KR-KPA
1: $K \xleftarrow{\$} \{0,1\}^k$
2: Init oracle $\mathcal{O}$ with key K
3: $K' \leftarrow \mathcal{A}^{\mathcal{O}}()$
4: **return** $1_{K=K'}$

Oracle $\mathcal{O}()$: KPA
1: $\mathsf{pt} \xleftarrow{\$} \{0,1\}^n$
2: $\mathsf{ct} \leftarrow \mathtt{Enc}(K, \mathsf{pt})$
3: **return** $(\mathsf{pt}, \mathsf{ct})$

The steps of the game, on the left-hand side, define the environment under which the adversary algorithm is tested: a key is sampled uniformly at random by the game itself and given to the KPA oracle (an encryption engine that the adversary can interact with); the adversary is then run in interaction with said oracle; its return value is then tested for equality against the key originally chosen: the game returns 0 or 1 based on this test.
The steps of the oracle, on the right-hand side, define what the oracle does upon reception of a request coming from the adversary. The request is empty, and the response consists of a freshly-sampled random plaintext, paired with its encryption under the key generated by the game. The adversary thus has access to arbitrarily many (but still polynomial, since it has to run in polynomial time) plaintext-ciphertext pairs, with the plaintext not being chosen by him.

One piece is still missing: the advantage. We define the advantage of the adversarial algorithm $\mathcal{A}$ against the KR-KPA game Γ as $\mathsf{Adv}(\Gamma, \mathcal{A}) = \mathbb{P}(\Gamma^{\mathcal{A}} \to 1)$.
In other words, the advantage achieved by an adversarial algorithm is the probability (taken over the randomness present in the game, in the oracle, and possibly in the adversarial algorithm itself) of the adversary correctly guessing the key in the scenario above described: key chosen at random, polynomially-many random plaintext-ciphertext pairs.

As will always be the case, we say that a block cipher is secure under this game, or KR-KPA-secure, if any probabilistic polynomial-time (PPT) adversary achieves a negligible advantage (in terms of the security parameter λ, which is an implicit input to all the above algorithms).

KR-CPA Let us start making the game more interesting by increasing the adversary power: the game itself will remain the same (i.e., the adversary still has to guess the key), but the oracle will now encrypt any plaintext chosen by the adversary (the scenario is now a *Chosen-Plaintext Attack*). The pseudocode for the new oracle is:

Oracle $\mathcal{O}(\mathsf{pt})$: CPA
1: $\mathsf{ct} \leftarrow \mathtt{Enc}(K, \mathsf{pt})$
2: **return** ct

The advantage definition also remains the same.

We do not give a formal proof of the following fact: if a block cipher is KR-CPA-secure, then it is also KR-KPA-secure. This intuitively follows from the observation that, whatever an adversary can do in the KPA scenario, it can also do in the CPA scenario by just randomly sampling the plaintext to query to the oracle (CPA adversaries are more powerful, so they form a "superset" of the KPA adversaries).
A bit more precisely, any adversary playing the KPA game can be transformed in an adversary playing the CPA game (in the way described above) which achieves the same advantage; so if a high-advantage adversary exists for the KPA game, then one also exists for the CPA game. The final thesis then follows by contrapositive: if no high-advantage CPA adversary exists, then no high-advantage KPA adversary exists.

2.2.5 Security Definitions in the General Case

We have seen two simple security definitions in the block cipher setting, where no IV is required. We will now see three security games in the general arbitrary-length encryption case, where IVs will be part of the picture. For simplicity, we will impose that the ciphertext and the plaintext have the same length; this means that we will restrict the plaintext space to some set $\mathcal{D} \subset \{0,1\}^*$ that does not necessarily contain all binary strings (for example, it might only contain bitstrings whose length is a multiple of the block size, for block ciphers used in conjunction with some mode of operation).
In all three of these games, the adversary power will be represented by a CPA oracle, which will enforce the non-repetition of the nonces (since they are chosen by the adversary in the query, together with the plaintext). The nonce space is denoted by $\mathcal{N}$.

KR-CPA We start by revising the last game we have seen, in the more general scenario.

Game $\Gamma^{\mathcal{A}}$: KR-CPA
1: $K \xleftarrow{\$} \{0,1\}^k$
2: $\mathtt{Used} \leftarrow \emptyset$
3: Init oracle $\mathcal{O}$ with K and $\mathtt{Used}$
4: $K' \leftarrow \mathcal{A}^{\mathcal{O}}()$
5: **return** $1_{K=K'}$

Oracle $\mathcal{O}(\mathsf{pt}, N)$: CPA
1: **if** $N \in \mathtt{Used}$
2: **return** $\perp$
3: add N to $\mathtt{Used}$
4: $\mathsf{ct} \leftarrow \mathtt{Enc}(K, \mathsf{pt}, N)$
5: **return** ct

The advantage definition is still the same

$$\mathsf{Adv}(\Gamma, \mathcal{A}) = \mathbb{P}(\Gamma^{\mathcal{A}} \to 1)$$

Overall, nothing much has changed. In order to win the game, the adversary still has to correctly guess the randomly-chosen key, with non-negligible probability, in polynomial time, by issuing encryption queries to an oracle. The only difference is that now unique IVs have to be provided for each query by the adversary and kept track of by the oracle.

As before, this game does not convey a very strong security notion, because it is very hard for the adversary to win, so saying that some encryption scheme is secure under this notion is not very informative.
A toy example shows why this security definition is not strong enough. Consider a trivial encryption scheme that does not use the key at all; instead, it outputs a ciphertext equal to the plaintext. This scheme would clearly be KR-CPA-secure, because there is no way to guess the key, since it does not influence the ciphertexts. Nonetheless, it would be a terrible scheme to use in practice, because messages would always be in clear.
The takeaway is that an adversary might still be able to do interesting things (limit case, decrypt every message) even without the key. In other words, a scheme can be KR-CPA-secure and still be completely worthless.

DEC-CPA This stronger security definition aims at solving the problem of the previous one, by challenging the adversary to decrypt a randomly-chosen message instead of guessing the key.

Game $\Gamma^{\mathcal{A}}$: DEC-CPA
1: $K \xleftarrow{\$} \{0,1\}^k$
2: $\texttt{Used} \leftarrow \emptyset$
3: Init oracle $\mathcal{O}$ with K and $\texttt{Used}$
4: $X_0 \xleftarrow{\$} \mathcal{D}$, $N_0 \xleftarrow{\$} \mathcal{N}$
5: $Y_0 \leftarrow \mathcal{O}(X_0, N_0)$
6: $X' \leftarrow \mathcal{A}^{\mathcal{O}}(N_0, Y_0)$
7: **return** $1_{X'=X_0}$

Oracle $\mathcal{O}(\mathsf{pt}, N)$: CPA
1: **if** $N \in \texttt{Used}$
2: **return** $\perp$
3: add N to $\texttt{Used}$
4: $\mathsf{ct} \leftarrow \texttt{Enc}(K, \mathsf{pt}, N)$
5: **return** ct

The advantage definition is again the same, although the condition has changed for the game to return 1.

$$\mathsf{Adv}(\Gamma, \mathcal{A}) = \mathbb{P}(\Gamma^{\mathcal{A}} \to 1)$$

This time, besides sampling a random key to be used by the encryption oracle, the game also samples a challenge message X_0 and encrypts it with a random nonce N_0 (which immediately goes among the used ones): the result Y_0 is given, together with N_0, to the adversary, for him to guess the message X_0, again with the aid of an encryption oracle; again, he has to do so in polynomial time, and guess with non-negligible probability.
Compared to the KR-CPA game, this game implies a stronger security notion, in the following sense: if an encryption scheme is DEC-CPA-secure, it is also KR-CPA secure. As in the previous section, we don't give a formal proof of this fact.
Intuitively, it is true because if a high-advantage KR-CPA adversary exists, then it can be

trivially used to build a high-advantage DEC-CPA adversary that uses the key returned by the KR-CPA adversary to decrypt Y_0 (remember that the algorithms are public and executable by the adversary with arbitrary inputs, the oracle is only there to encapsulate access to the key, not to the algorithm). In other words, if we are able to extract the key from the cipher as an adversary, then we can use that key to decrypt challenge ciphertexts.
The thesis, then, again follows by contrapositive: if no high-advantage adversary exists for the DEC-CPA game, then no high-advantage adversary exists for the KR-CPA game.

Although this game does express stronger security compared to the KR-CPA game, it is still not enough. As another toy example, imagine a very good cipher C that is indeed DEC-CPA-secure; from C, construct a stupid cipher C' that, when encrypting message m, encrypts the first half with C, and leaves the second half in clear, juxtaposing the two halves. Clearly, C' would also be DEC-CPA-secure, because an adversary would need to also recover the encrypted first half in order to win the game, which it cannot because C is DEC-CPA-secure. Nonetheless, this would again be a terrible cipher to use because it leaves half of the message in clear.
The takeaway is that, even without the ability to decrypt the whole message, an adversary can still deduce interesting information on the plaintext (limit case: discover a non-negligible fraction of its bits). Thus, a scheme can be DEC-CPA-secure and still be essentially worthless.

IND-CPA We finally get to the strongest security definition (in terms of adversary goal), that will leave no room for an adversary to gain even the slightest insight into the plaintext.
First, let us introduce the *ideal cipher*. Recall that, fixed a message length n, a cipher is essentially a collection of permutations over $\{0,1\}^n$, indexed by a key. A key of reasonable length k can only index 2^k such permutations, far less than their total number, i.e., $(2^n)!$; if we wanted to index all possible permutations, we would need $k = \log_2((2^n)!) \sim n \cdot 2^n$ bits of key, which would amount to roughly 256 EiB (Exbibytes) of key for the outdated DES blocksize $n = 64$; this is clearly infeasible. However, let us observe that if, ideally, our cipher indeed allowed us to sample a permutation over $\{0,1\}^n$ uniformly at random, that would solve all of our problems, since the adversarial observation of any (chosen) input-output pair would still leave completely undetermined the remaining mappings of the permutation: the adversary could not deduce any further information whatsoever based on the samples he manages to gather. We call this hypothetical construction the *ideal cipher*.
The IND-CPA game challenges the adversary to distinguish the actual cipher from the ideal cipher. This is the best we can hope for: as was said, the ideal cipher would need too long keys, but we can hope that our actual cipher is computationally indistinguishable from it.

Game $\Gamma_b^{\mathcal{A}}$: IND-CPA
1: $K \xleftarrow{\$} \{0,1\}^k$
2: $\texttt{Used} \leftarrow \emptyset$
3: Sample Π_N, a length-preserving permutation over $\mathcal{D}$, for every $N \in \mathcal{N}$
4: Init oracle $\mathcal{O}_b$ with K, $\texttt{Used}$, and the Π_N's
5: **return** $\mathcal{A}^{\mathcal{O}}()$

Oracle $\mathcal{O}_b(\texttt{pt}, N)$: CPA
1: **if** $N \in \texttt{Used}$
2: **return** $\perp$
3: add N to $\texttt{Used}$
4: **if** $b = 1$
5: **return** $\texttt{Enc}(K, \texttt{pt}, N)$
6: **return** $\Pi_N(\texttt{pt})$

The advantage definition is

$$\mathsf{Adv}(\Gamma, \mathcal{A}) = \left|\mathbb{P}(\Gamma_1^{\mathcal{A}} \to 1) - \mathbb{P}(\Gamma_0^{\mathcal{A}} \to 1)\right|$$

Let us go through this new definition step-by-step.
In line 3 of the game, we are instantiating the ideal cipher; since the encryption also depends on the nonce, we actually have to sample a uniformly-random length-preserving permutation for every nonce N. The fact that these permutations might be prohibitively expensive to represent is irrelevant here: the game is an ideal test ground for the adversary, it is not the cryptographic scheme; as such, it is not limited to polynomial-time and -space computations.
The oracle, and thus the game itself, is parametrised by a bit b, governing whether the real cipher or the ideal cipher is used to respond to queries.
The advantage definition clarifies that the adversary's goal is to distinguish between the two situations. He achieves a high advantage by outputting two distinct symbols when made to play with the two versions of the oracle: the interface is the same, he has to deduce which one he is playing with based on the outputs he gets.

As before, a "stronger-than" relation holds between IND-CPA and DEC-CPA, i.e., if a scheme is IND-CPA-secure, it is also DEC-CPA-secure. The skeleton of the proof is the same as before (and the same as most reduction proofs in cryptography): by contrapositive, showing that a high-advantage DEC-CPA adversary can be used as a subroutine to build a high-advantage IND-CPA adversary.
Intuitively, this constructed distinguisher (synonym for IND-CPA adversary) randomly picks a message $X_0 \in \mathcal{D}$ (and a nonce) to encrypt using the oracle, receiving Y_0; it then runs the decryptor (synonym for DEC-CPA adversary) with Y_0 (and the nonce) as input, to see if it outputs X_0, in which case the distinguisher outputs 1, otherwise it outputs 0. In other words, if we are able to decrypt ciphertexts of the actual cipher, we are also able to tell whether we are playing with the actual cipher or with the ideal cipher, by simply checking whether we are able to decrypt the encryption of a random message.

This strong security property is often a minimum requirement for block ciphers: AES, just like many other competitive block ciphers, is believed to be IND-CPA-secure (although no formal proof exists).
One consequence of the property itself is that, broadly speaking, the ciphertext is indistinguishable from a random string. Most plausibly-deniable storage schemes, including our Shufflecake, heavily rely on this indistinguishability between encrypted data and empty blocks filled with random bytes to provide their guarantees.

2.3 Oblivious Random Access Memory

In this section, we briefly discuss ORAMs (Oblivious Random Access Memory), a cryptographic primitive that has been studied since the 1990s [43], and whose link with plausibly-deniable storage has only recently been established [12, 19]. Indeed, they constitute a more general construct, in that they give stronger security guarantees, at the price of a substantial overhead; a restricted, "write-only" version of this primitive has been shown to be necessary and sufficient [19] for plausibly-deniable storage in the most stringent threat model.
The first part specifies the problem statement in the general case ("full" ORAMs); the second part introduces and motivates Write-Only ORAMs, and briefly discusses the two main constructions.

2.3.1 Full ORAMs

ORAMs come in handy when a client C wants to outsource a large database to some external storage S, but the contents of the database are sensitive and must remain confidential, and the storage itself is the adversary. The adversarial behaviour (or adversary power) is often described as "honest but curious": the storage abides by the stipulated protocol (the storage does respond honestly to `read` and `write` requests), and does not try to maliciously craft answers to queries so as to "actively" extract more information from the client by the way it reacts. However, it does observe the sequence of incoming requests, and carries out all the inference it can, solely based on this flow of information. In other words, S can only carry out *passive attacks*.
The two most common scenarios in which this problem is relevant are the CPU-DRAM case and the cloud storage case. In the first one, the client is a trusted CPU which runs a program operating on sensitive data that are stored on a vulnerable DRAM bank (e.g., subject to bus probing by an adversary with physical access to the hardware). It is a realistic model, for example, in situations involving smart cards, where hardening the CPU die is much more feasible than shielding the memory bank or the communication channel between them [25]. In the second one, the client is a local party which needs to store large amounts of sensitive data (e.g., personally-identifiable information) in the cloud, i.e., on some remote premises owned by a vendor selling its high-capacity storage as a service to the client. Here, again, either the communication channel might be compromised, or the cloud provider itself might be malicious (or forced to collaborate with invasive authorities).
As a final addition, note that the client does possess some small amount of local, private memory of its own; usually, this is appropriately named *client memory*. It often plays a key role in the security of the ORAM schemes, because it is inaccessible to the adversary, and so can hold the state of the ORAM instance completely in the clear.

Access patterns Clearly, in order to defend against the party that is storing the data, one needs to encrypt those data, otherwise the adversary could just read everything. However, the reason why a whole new primitive is needed in this scenario is that encryption alone is not enough. A simple encrypted database, with data divided in chunks called *blocks* that are the unit of encryption and requests (i.e., each block is encrypted with a different IV, and `read`/`write` requests deal with blocks), would leak the *access patterns* of the client: even though the adversary could not see the content of the data it is serving, it could still see which locations are being requested and whether the request is a `read` or `write` operation.
Several research papers [64, 31, 33] have shown how some prior knowledge by the adversary on the structure of the data and on the code accessing it can lead to severe leaks if the code performs data-dependent memory accesses. ORAMs protect exactly against these attacks, acting as an intermediate layer between the client and the storage that "shuffles" the layout of the data and translates requests accordingly, thus making the *physical* access patterns (observed by the adversary) independent from the *logical* access patterns (issued by the client).

Security definition Let us define an *access* as a triple $o = (\texttt{op}, b, d)$ consisting of the operation `op` (`read` or `write`), the block address b, and the data d (if $\texttt{op} = \texttt{read}$, then d is the return value). Let us also define an *access pattern* as an ordered sequence of accesses $\mathbf{y} =< o_1, \dots o_n >$. Finally, let us define as $A(\mathbf{y})$ the physical access pattern executed by the ORAM scheme A upon reception of the logical access pattern $\mathbf{y}$ from the client.
The security property of an ORAM can then be phrased as follows: for any two logical access

patterns $\mathbf{y}$ and $\mathbf{z}$ of the same length, the resulting access patterns $A(\mathbf{y})$ and $A(\mathbf{z})$ must be computationally indistinguishable. This concise wording conceals a distinguishing game where the adversary is made to choose $\mathbf{y}$ and $\mathbf{z}$, the game only translates one of the two through the ORAM A and returns the result to the adversary, who has to guess which of the two was translated. Quite evidently, security in this sense means that the adversary is not able to infer the logical access patterns from the physical access patterns.

PathORAM PathORAM [55] is an ORAM scheme devised in 2013 that has reached widespread adoption in many applications, both for its simplicity and for its performance (that is near-optimal in some specific technical sense). We do not describe it here, for brevity; instead, we summarise its performance metrics, to give an idea of the price to pay for the security offered by ORAMs.
Let N be the total number of data blocks - each of size B bits - of the "exposed" logical storage at the client. PathORAM uses $O(B \log N)$ bits of client memory, and has a communication overhead of $O(\log^2 N)$ both for `read`s and `write`s. On the other hand, it uses 100% of the storage space (up to what is needed to store the IVs, which is not explicitly mentioned in the paper, since it is considered an implementation detail), so it requires N block of data on the remote storage S.

2.3.2 Write-Only ORAMs

Write-Only ORAMs are a relaxed variant of ORAMs, with a weaker threat model: adversaries only see physical `write` requests, but not `read`s. Revisiting the security definition given earlier for ORAMs, we require that $\texttt{WriteOnly}(A(\mathbf{y}))$ be indistinguishable from $\texttt{WriteOnly}(A(\mathbf{z}))$ for any two access patterns $\mathbf{y}$ and $\mathbf{z}$ that have the same number of `write`s, where $\texttt{WriteOnly}(\cdot)$ is a filter that discards `read`s and preserves `write`s.
This primitive has limited applicability in the cloud scenario: the client C does hold the entire database locally, and only uses the remote untrusted storage as a backup to be kept in sync. This way, the client can perform `read`s locally, and only issue `write` requests to the storage S to synchronise the local changes. In the CPU-DRAM case, this model corresponds to an adversary that, instead of probing the bus to see each incoming request, can only take several snapshots of the DRAM and infer, from the successive deltas, a somewhat-aggregated version of the `write`-trace (the sequence of physical `write` requests).
For the same reason, this is a valid abstraction to use for the plausibly-deniable storage case: physical `read` requests leave no trace on the disk, so all the adversary can observe through snapshots of the disk is a somewhat-aggregated version of the `write`-trace. Therefore, modelling the adversary as having access to the entire detailed `write`-trace only makes it stronger, increasing the security of the scheme: this is what has been previously called "the most restrictive threat scenario", where Write-Only ORAMs are necessary and sufficient [19].

HiVE A foundational paper [12] from 2014 proposed a new scheme called HiVE (Hidden Volume Encryption) that rekindled the interest of researchers towards Write-Only ORAMs. It was specifically framed in the context of plausibly-deniable storage, and since then a few new schemes have been proposed that remediate to its performance problems. Again, we only give the final metrics, to show how this relaxed threat model can lead to more efficient solutions.
Let us again denote the number of usable blocks by N, and the block size by B; let us also recall that λ is the security parameter, implicitly tending to infinity. Under the assumption that $B = \Omega(\log^3 N)$, HiVE uses $O(B \cdot \lambda)$ bits of client memory, and has $O(B)$ communication

complexity. On the other hand, in order to achieve this performance, half of the storage space needs to be wasted (the remote storage S has to hold $2N$ blocks).

DetWoOram This scheme [47], proposed in 2017, was a major breakthrough: it was deterministic, it had a sequential `write` trace, and a partially sequential `read` trace; also, it achieved a very low communication complexity. All this, at the expense of a fairly low disk space utilisation (25% with their final choice of parameters).

DetWoOram entails $O(\log N)$ physical block `read`s for each logical access (`read` or `write`); additionally, logical `write`s also incur 2 physical `write`s. The client memory is $O(B)$ bits. The remote storage S has to hold $4N$ blocks.

3 Plausible Deniability

In this chapter we illustrate in depth the concept of plausible deniability (abbreviated as PD) for local storage, first through the example of TrueCrypt (the first commercial software product to offer it as a feature), and then through a rigorous game-based definition.

3.1 TrueCrypt

TrueCrypt [57] was the first disk encryption software (now discontinued) to offer PD capabilities. It was developed around the early 2000s, before BitLocker [41] and LUKS [45] 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 [60]) that has kept most of the design principles, and improved on some minor aspects (like a stronger key derivation).

3.1.1 Design

In this section, we outline the principles behind TrueCrypt's design, and its operational model.

Choice of the block layer TrueCrypt operates at the block layer, i.e., it implements a block device driver that encrypts and decrypts data on the fly, exposing a virtual block device to be used by the file-system layer; this is the same choice adopted for Shufflecake, and it allows to deal with simple low-level `blockRead` and `blockWrite` requests. Alternative solutions exist that work at the file-system layer [39, 44], but they have to implement a richer 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 [32] (flash-translation layer, for SSDs), but this approach clearly leads to highly vendor-specific solutions.
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, directing *logical* `read` and `write` requests to it; the stacking driver then executes its algorithm to map these requests to *physical* `read` and `write` 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.

Disk layout The layout of a TrueCrypt-formatted disk is shown in Figure 7.
The first part illustrates the case of the disk only containing one volume, the Standard Volume; this was indeed a very common scenario, since TrueCrypt was primarily a disk encryption software (plausible deniability was just an additional optional feature). The first initialisation operation performed by TrueCrypt is to fill the disk with random bytes; incidentally, this is also the case for regular disk encryption tools. The first part of the disk contains the encrypted header of the Standard Volume (Header 1 in the picture), and an empty slot filled with random bytes (remaining from the initialisation procedure). Then comes the actual encrypted data section of the Standard Volume (Volume 1 in the picture), followed by some empty space, also filled with random bytes (also coming from the initialisation procedure).
TrueCrypt optionally allows to "embed" a Hidden Volume (Volume 2 in the picture) in the empty space left by the Standard Volume: this is the mechanism providing plausible deniability. Its

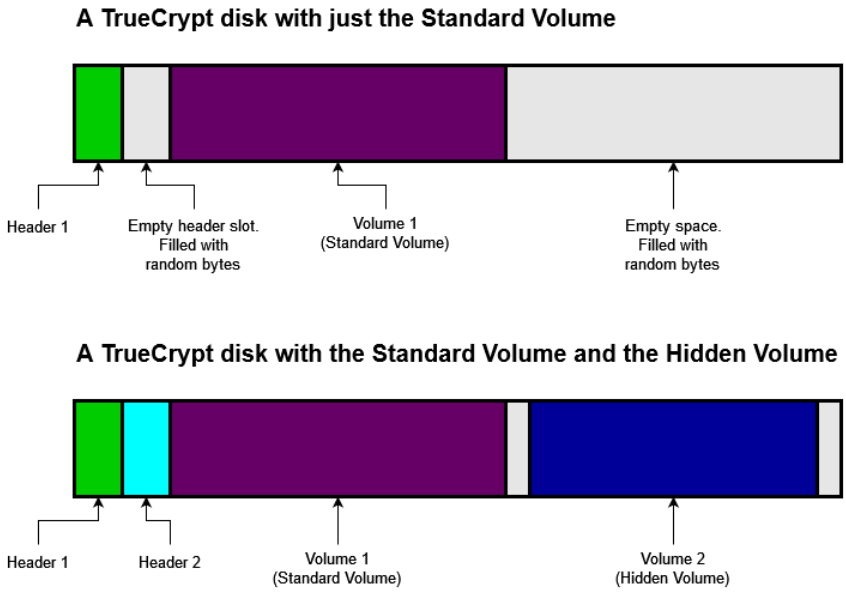


Figure 7: TrueCrypt disk layout

encrypted header (Header 2 in the picture) then fits in the empty slot left after Header 1. The Standard Volume and the Hidden Volume are encrypted with two different passwords.
Some relevant terminology in our PD scenario: synonyms of Standard Volume are outer volume (since it "embeds" the Hidden Volume within its empty space), and decoy volume (since it contains data that is meant to deceive the authorities). This last one is more general, since it applies to the volumes to be surrendered to the authorities, not just in TrueCrypt, but also in other schemes that do not necessarily place volumes "one within the other".

Hidden volume TrueCrypt only supports a hidden volume if the outer volume is formatted as a FAT file system. This is because, for the situation to be as simple as shown in Figure 7, we need the empty space left by Volume 1 to be contiguous. If we allowed the empty space to be made of "holes" interleaved with encrypted content from Volume 1, we would need to resort to an indirection layer, logically "coalescing" these holes into a contiguous segment of storage space to be used by Volume 2; this would be more expensive, both in terms of computing time (mapping a logical block to a physical block) and of storage overhead (this mapping would need to be stored in the header). Instead, we can afford to have Volume 2 start at a certain offset, after the end of Volume 1, and then follow linearly, exactly because the empty space left by Volume 1 is physically contiguous.
Allowing, for Volume 1, file systems like ext4, that "jump back and forth" on the disk leaving lots of empty blocks in the middle, would disrupt this property; FAT, instead, 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 is why, if a Hidden Volume is desired, the Standard Volume must be formatted as a FAT file system.
As long as this holds, a Hidden Volume can be created at any moment. TrueCrypt automatically calculates a convenient starting position for the Hidden Volume (leaving some leeway after the end

of the Standard Volume), and places it among the metadata of Header 2. 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 not to 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 are inspected by the adversary.

Headers Truecrypt headers are meant to be indistinguishable from random noise when encrypted (so as to preserve deniability), whereas headers of regular disk encryption tools usually contain encrypted fields but also cleartext metadata, and so they are clearly identifiable.
The decrypted headers contain various metadata about the volume they represent, plus the key used to encrypt the volume data itself [58]. Keys are decoupled: the one used to encrypt the volume data is different from the one used to encrypt the header. This last one is derived through the user-memorised password, using a KDF, whose salt is stored in clear (but indistinguishable from random) at the beginning of the header. Therefore, the user-memorised passwords actually decrypt the headers: the keys to decrypt the volumes are contained in the headers.

Operational model Using TrueCrypt for plausible deniability comes with a few restrictions on what the user can do in order to preserve data integrity and deniability.
Once the Hidden Volume is created, its starting position is final, and it "freezes" the end of the 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 not to let it grow too much and overwrite the Hidden Volume. This is achieved both by frequent de-fragmentation of the FAT file system within the Standard Volume, and by actually not writing too much data into it.
The Standard Volume must contain "decoy" data, that will reasonably convince the authorities that it is the only volume existing on the disk. Clearly, the user only surrenders the password to the Standard Volume to the authorities, when under investigation. This obviously opens the door for corruption of the Hidden Volume, if they start writing data in the Standard Volume. This is inevitable, and can only be mitigated by frequent backups of the disks.

3.1.2 Security

We will see a rigorous formulation of the plausible deniability property in section 3.2; here, we only informally describe the level of security enjoyed by TrueCrypt. This, together with some of the weaknesses presented in section 3.1.3, will inspire the formal game-based security definition.

Single-snapshot Let us tackle the *single-snapshot* threat model, in which the adversary only obtains one full snapshot of the disk contents at the byte level (besides the password of Volume 1). Quite simply, once the user surrenders the password of Volume 1 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 Volume 2, it will be filled with random bytes that are not readable with the password of Volume 1. More precisely, IND-CPA security ensures that the adversary is not able to distinguish between the random bytes used to fill the empty space from those resulting from the encryption of Volume 2. Therefore, even if Volume 2 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 contents.
In summary, no evidence is left on the disk, even after unlocking the Standard Volume, that a Hidden Volume exists, because the snapshots of a disk with just the Standard Volume and one with a Hidden Volume too look exactly the same.

Multi-snapshot The *multi-snapshot* model is a more severe threat scenario, in which the attacker obtains several full snapshots of the disk across time, with the user possibly making changes to all volumes between the times at which these snapshots are taken.
There are several reasons why this is a realistic threat model. First, it is perfectly possible to get stopped more than once by the police, or anyway to have your disk regularly scanned (e.g., at hotel check-ins or at airport security). Second, the FTL on SSDs may leave old versions of some blocks of data lingering about in the flash: even though they are not logically accessible in software (not even at the block layer), an advanced forensic adversary might be able to scan the disk at the flash level and thus recover pieces of old snapshots. Lastly, if the underlying disk on which volumes are created is actually a file-backed virtual device, and the backing file is contained in a journaling file system like ext4, old versions of it are trivially recoverable through the file system journal.
We show in section 3.1.3 that TrueCrypt is **not** secure in this threat model.

3.1.3 Weaknesses

Let us clarify that the adversary we are playing against has a comparatively easy goal: *identify* a hidden volume, distinguish between the case where it exists and the case where it does not. We have just argued that a TrueCrypt disk with just the outer volume and one with both an outer and a hidden volume are indistinguishable, even when the password of the outer volume is provided. However, depending on the situation, this might not be enough to quell all suspicions.
Here we show three problematic aspects of TrueCrypt that, if exploited, would break its security and allow an adversary to identify a hidden volume. It is to be noted that some of them only apply now, but were not relevant in the early 2000s, when TrueCrypt was first conceived.

Mandating FAT for the outer volume As was discussed, the Standard Volume must be formatted as a FAT file system, but only if a Hidden Volume is desired.
However, FAT is now outdated: it used to be very widespread, but today there is no real plausible reason to use it anymore. Therefore, once we let the authorities decrypt the decoy volume, although the free blocks do look the same whether or not there is a Hidden Volume, it is the content of the decoy volume itself that gives us away: why, having a choice, would we have formatted it with FAT?
The immediate, obvious lesson to learn from this is that a PD scheme must allow its volumes to be formatted with any arbitrary file system, or at least with plausible ones. This is particularly relevant when devising the allocation policy that governs how to reserve some disk space for this or that volume.
A generalisation of this guideline, that is unfortunately hard to formalise, is that the 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 it would become very suspicious seeing that the most recent updates are months if not years old.

Only allowing one hidden volume Just like the previous one, this weakness is not due to some mismanagement of the storage area containing the hidden volume, but to some public information that we do disclose to the adversary. In this case, it is the very fact that we are using TrueCrypt. Let us note that, before surrendering the outer volume's password, the whole disk is filled with random bytes, as everything is encrypted except for some empty spaces (also filled with random bytes) and the KDF salt (which is also random). Thus, when asked for the first password we could even plausibly claim that the disk is not formatted with TrueCrypt at all, but is instead the result of a secure wiping procedure (indeed, overwriting the disk with random bytes is a common way to achieve it). However, a real-world adversary might still be able to gather hints that TrueCrypt is being used, e.g., by searching the user's laptop and discovering a TrueCrypt installation. 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.
This is a problem because the adversary might reasonably suspect that such schemes are in use *exactly* to hide something: if we meant to encrypt just a single volume, it would be slightly weird, although not unbelievable, that we use a non-standard solution to that end (again, this is only relevant now that standard disk encryption tools exist). A user could well claim that they prefer using independent 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. This, of course, breaks the PD guarantees of TrueCrypt.
The short answer to this problem is: allow more than two volumes. That way, we can create a series of volumes with increasingly "secret" 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.

Multi-snapshot insecurity Contrary to the first two weaknesses, this is indeed due to TrueCrypt not concealing the hidden volume well enough. Also, while the first two issues have been conclusively solved by newer PD schemes, this one remains a thorn in researchers' side: the reason is that, in a recent research paper [19], security against this attack has been proven equivalent (necessary and sufficient) to WoORAMs (in the most stringent threat scenario), for which no time- and space-efficient solution exists yet.
From the definition we have seen before, 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 "empty" blocks have changed, which immediately tells them that a second volume exists, because TrueCrypt never re-randomises the actually-free space.
We have already discussed how this threat model is well-suited for some real-world adversarial situations. Nonetheless, there have been no (public) reports of successful identification of TrueCrypt hidden volumes in this way; also, TrueCrypt's successor, VeraCrypt, continues to be relatively widespread despite being exactly as insecure against multi-snapshot adversaries. If one wishes to be fully protected against such attacks, the only way is to resort to Write-Only ORAMs, and to pay the price of their performance overheads. Alternatively, some "operational", unproven security can be achieved by effectively re-randomising the actually-free space: this way, a user can claim that the changes occurred to the "empty" blocks are due to this re-randomisation, and not to a hidden volume.

3.2 Security Definition

In this section, we present the 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 one framework (SoK [19]), but here we will follow the arguably more intuitive one given in a foundational paper from 2014 (HiVE [12]).

3.2.1 Framework

In this setting, a user employs a PD scheme to multiplex a single storage device D into ℓ independent volumes $V_1, V_2, \dots V_\ell$, 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 obviously publicly known. The security goal of the scheme will be to hide the value of ℓ from an adversary, i.e., hide the actual number of volumes that exist on the device. More precisely, in order to mimic the real-world situation where all passwords are disclosed to the adversary except one (that of V_ℓ, the "most secret" volume), the game will challenge the adversary to distinguish a disk with $\ell - 1$ volumes from one with ℓ volumes (with ℓ being chosen by the adversary itself), given the first $\ell - 1$ passwords.
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 (e.g., 4096 bytes).
The semantics of the scheme is given by the following three methods.

`PDSetup`(D, $< P_1, \dots P_\ell >$):
Initialises the disk to host ℓ volumes $V_1, \dots V_\ell$, encrypted with passwords $P_1, \dots P_\ell$.
Returns an instance Σ encapsulating everything.

`PDRead`(Σ, i, b):
Reads block b from volume V_i.

`PDWrite`(Σ, i, b, d):
Writes data d into block b of volume V_i.

The obvious functional requirements of consistency apply to these methods: `PDRead`(Σ, i, b) must return the d that was passed as argument to the last `PDWrite` call with the same Σ, i, and b (for simplicity, we consider operations atomic).

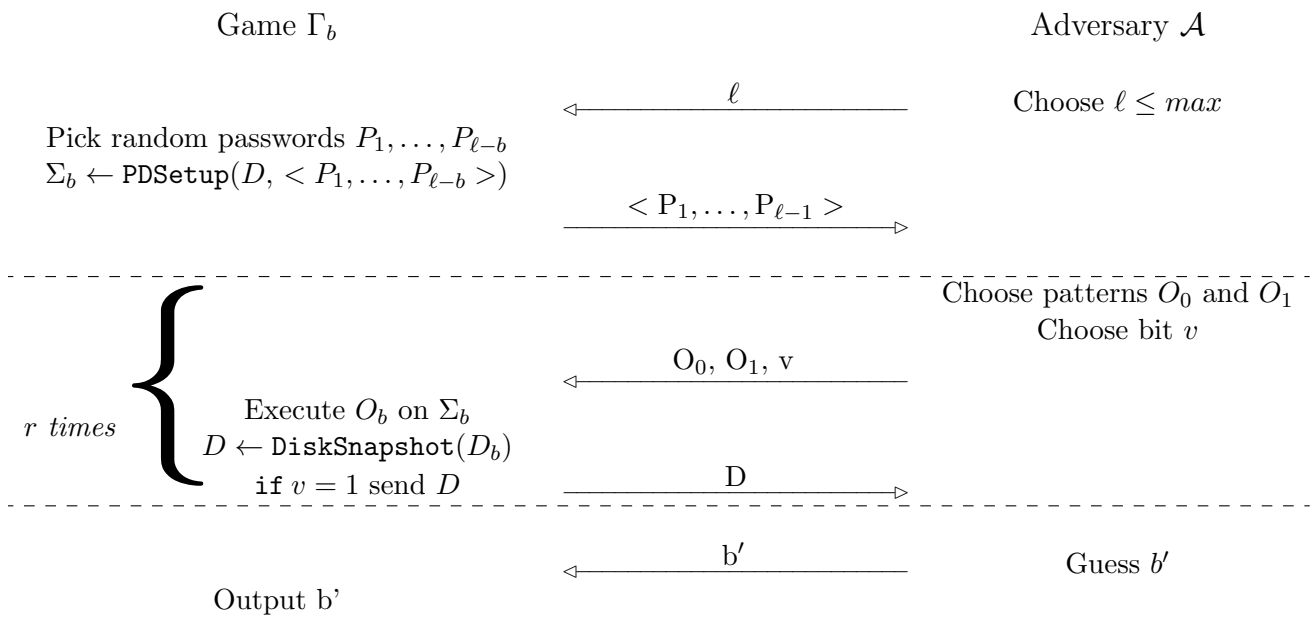
3.2.2 Security Game

The security game inherits some high-level concepts from the IND-CPA game. The adversary is a distinguisher, and is challenged with deducing whether he is interacting with a Σ encapsulating ℓ or $\ell - 1$ volumes; also, he chooses the `read` and `write` operations to submit to the Σ.

Access patterns Let us define an *access* as the tuple o = (`op`, i, b, d), with `op` being either `read` or `write` (if `op` = `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 an ordered sequence of accesses

$O =< o_1, \ldots o_n >$. An empty access $o = \perp$ is also defined, which is simply ignored by the Σ.

Security game Since the game is structured in several interactive phases, it will be more conveniently represented in a different form than the ones we have seen so far.



As usual with distinguishing games, the advantage is

$$\mathsf{Adv}(\Gamma, \mathcal{A}) = \left|\mathbb{P}(\Gamma_1^{\mathcal{A}} \to 1) - \mathbb{P}(\Gamma_0^{\mathcal{A}} \to 1)\right|$$

In words, the steps of the game are as follows:

1. The adversary chooses ℓ and sends it to the game (or *challenger*). Its goal will be to guess whether the game instantiated ℓ or $\ell - 1$ volumes.
2. Based on the bit b, the game randomly samples either ℓ or $\ell - 1$ passwords (more concisely, it samples $\ell - b$ passwords).
3. The game creates $\ell - b$ volumes on the device, using the passwords sampled in step 2. This results in either Σ_0 or Σ_1 being instantiated.
4. The game sends the first $\ell - 1$ passwords to the adversary. These are all the passwords if $b = 1$. Otherwise, if $b = 0$, the password to V_ℓ is not disclosed.
5. r interactive rounds are performed; r is polynomial in the (implicit) security parameter λ.
   (a) In each round, the adversary 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 $V_1, \ldots, V_{\ell-1}$). He also chooses a bit v, signalling whether he wishes a snapshot of the disk at the end of this round.
   (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 to the adversary.
6. At the end of all rounds, the adversary decides on a bit b' to output, representing whether it thinks it is playing with Σ_0 or Σ_1. The bit is output as-is by the game.

For the sake of brevity, we have omitted the constraints that the adversary is subject to in step 6, when choosing the access patterns and when choosing the bit v; we will present them in Sections 3.2.3 and 3.2.4. Without any such constraints, we will see that security would be impossible to achieve; also, the exact set of constraints will specify the adversary power and the adversary goal.

3.2.3 Constraints on the Bit v

This constraint governs the snapshotting capabilities of the adversary, thus the adversary power.

Arbitrary No constraint: the adversary is allowed to set $v = 1$ in all of the r interactive rounds. This is closely akin to the even more severe threat model defined in [19], where the adversary gets access to the entire *physical write trace*, i.e., the entire, detailed, ordered sequence of physical `write` requests issued by the Σ to the disk. Security in this more severe threat scenario is proven to be equivalent to WoORAM security, i.e., secure WoORAMs are necessary and sufficient to achieve PD security in this scenario.
An *arbitrary* snapshotting capability, paired with the ability to submit very short access patterns, indeed yields to the adversary a pretty fine-grained view of the history of physical `write`s (the physical `write` trace is only loosely aggregated).

On-Event Same as before, but an additional `Unmount` operation is performed on the Σ by the game before forwarding the disk snapshot to the adversary. In the real world, this translates to the police only obtaining snapshots of your disk when it is unmounted, instead of breaking into your house to get a "live" snapshot.
This model leads to some interesting considerations: security in the *Arbitrary* case requires WoORAMs, which are expensive exactly because they need to maintain the disk state secure at all times; however, if the threat only comes once we unmount the disk, we can afford to keep the disk in a "vulnerable" state while we operate on it, and push the (possibly expensive) securing operations to the `Unmount`, which we only seldom perform.

Nevertheless, in the present work, we choose to overlook this specific threat model, because a clean termination cannot be expected to always take place in real-world applications, therefore relying on it for security seems like an unwise choice.
There is a valuable idea to be retained from this proposal, however: multi-snapshot security can be achieved through a completely different mechanism than single-snapshot security, and delegated to some separate procedure. The only caveat is that this procedure, for the aforementioned reasons, cannot be entirely pushed to Unmount-time, but needs to be spread out over the ordinary course of operations of the scheme (both to achieve more fault-tolerant security, and to de-amortise the possibly high cost of this procedure).
This is exactly the nature of the high-level ideas, given in Section 4.3, to "enhance" the vanilla Shufflecake construction with some degree of unproven, operational security in the multi-snapshot setting.

One-Time The adversary is single-snapshot, i.e., can only set $v = 1$ for one of the r interactive rounds. It is relatively easy to build a scheme that is secure in this model: it is the case for TrueCrypt (with $max = 2$).

3.2.4 Constraints on the Access Patterns

These constraints define the adversary goal, by specifying which two exact situations it 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). The plausibility of this claim in ensured by the indistinguishability of the snapshots.
The constraints on O_0 will represent what accesses the user will be able to actually perform, and which he will not, if he wants to stay secure; the constraints on O_1 will represent what he will be able to disguise O_0 as.

Minimal constraint Before diving into a couple of examples, let us settle that some constraint is necessary in order to have any hope in the PD game against the adversary. Without any limit, 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 basic rule that will have to be a "minimum common denominator" to all of the possible constraints we will subsequently present. The rule simply states that the resulting contents of volumes $V_1, \ldots, V_{\ell-1}$ must be the same, whether O_0 or O_1 is executed.
From the user's perspective, this basic requirement means that we do not try to disguise the `write`s to public 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.
Let us now see two concrete examples of such constraints.

Restricted hiding This constraint adds the following condition to the basic rule: if $o_{0,i}$ is in volume V_ℓ (whether it is a read or a write), then $o_{1,i} = \perp$. No limit is placed on O_0.
If a PD scheme is secure in this scenario, then a user is allowed to perform any operation they want with the secret volume (since no limit is placed on O_0), and they will be able to claim that the accesses to the secret volume simply never happened (O_1 is basically the same as O_0, with public volumes ending up having the same content, and operations on the hidden volume just being replaced by the empty operation).

Opportunistic hiding Although this was among the scenarios presented in [12], we do not repropose it here, because it imposes a constraint on O_0 that would arguably impair the usability of the PD scheme. In particular, it would force the user to perform more accesses on V_1 than on the rest of the volumes combined. However, the assumption that the "outermost" volume is used more frequently than the other, more secret ones, is simply too restrictive in our opinion.

Plausible hiding This constraint only requires that neither O_0 nor O_1 contain any $\perp$ operation, so that they have the same "actual" length (in addition to the basic rule).
As in the first scenario, O_0 can be chosen freely, so the user can perform whatever access pattern on the volumes and still be secure.

3.2.5 Resulting Threat Models

By combining a constraint on the snapshotting capability with one on the access patterns, one obtains different concrete threat scenarios. We quickly analyse some here.

The holy grail would clearly be a PD scheme achieving security in the most restrictive one, that is *Arbitrary* snapshotting capability in the *Restricted hiding* setting. However, this was proven impossible in [12].

The HiVE scheme achieves security in the *Arbitrary* + *Plausible hiding* scenario.
A variant of the scheme (called HiVE-B) is secure in the *Arbitrary* + *Opportunistic hiding* threat model.

TrueCrypt achieves security in the *One-Time* + *Restricted hiding* model.

4 Shufflecake

This chapter details our original contribution to the field of plausibly-deniable storage: a new scheme, called Shufflecake [3], that achieves proven security in the *One-Time + Restricted Hiding* scenario, and some unproven, operational security in the multi-snapshot case, if equipped with one of the "extensions" discussed in Section 4.3. Additionally, Shufflecake offers extremely good performance, which makes its adoption in real-world scenarios possible, unlike most WoORAM-based solutions. The scheme is implemented as a free and open-source device-mapper target for the 5.13 Linux kernel [2].

4.1 Design

First, we present a "vanilla" version of Shufflecake, that only achieves single-snapshot security: an extended construction, with a separate obfuscation procedure, that attains rough multi-snapshot security, is described in Section 4.3.
This simple version can be already considered a strong improvement on Truecrypt, since it fixes two of its crucial limitations: it allows multiple volumes, and it is filesystem-independent (it does not mandate volumes to be formatted with any particular file system in order to work correctly).

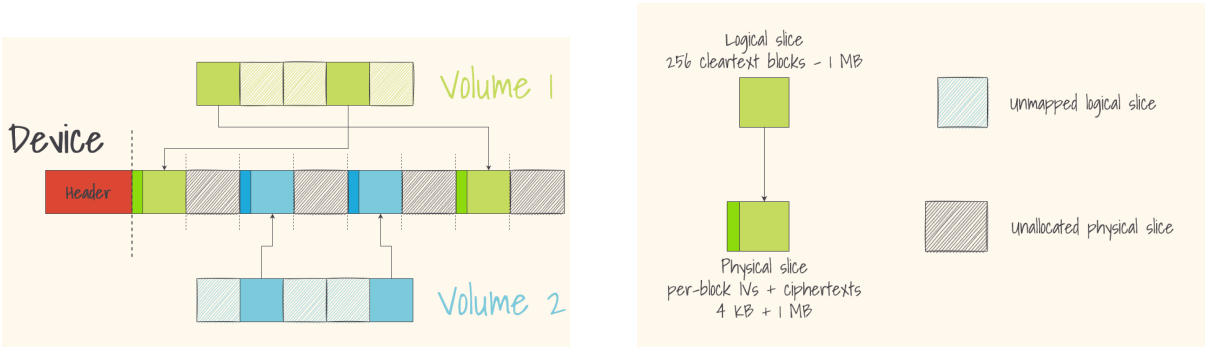
4.1.1 Disk Layout

Let us introduce some relevant terminology: 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 device's physical storage space is statically divided into a *header section* and a *data section*.

Indirection layer in the data section Instead of mandating that the volumes be physically adjacent, like in TrueCrypt, we randomly interleave them. Concretely, the layer of indirection, mapping logical block b of volume V_i to physical block β, works as follows.
We split the *logical* storage space of each volume into *slices* of S consecutive blocks; we also segment the data section of the *physical* storage space into slices of $S' = S + \Delta S$ consecutive blocks: physical slices are larger because they also comprise ΔS additional blocks at the beginning, which contain the per-block IVs encrypting the following S data blocks. We map a logical slice of a volume to a uniformly random physical slice in a simple way that is guaranteed to avoid conflict; this correspondence is maintained by a per-volume data structure called the *slice map*. The offset of a block *within* a slice is left unchanged by the mapping.
In other words, the *physical block address* β corresponding to a *logical block address* b of volume V_i is computed through this simple procedure:

1. Calculate the logical slice index (LSI) $\left\lfloor \frac{b}{S} \right\rfloor$, the index of the logical slice which b belongs to.
2. Obtain the physical slice index (PSI) corresponding to the given LSI of volume V_i by consulting V_i's slice map.
3. Reconstruct the physical block address (within the data section of the storage space) $\beta = S' \cdot \texttt{PSI} + \Delta S + (b \mod S)$

The layout is depicted in Figure 8. The picture refers to the choice of parameters $S = 256$ and $\Delta S = 1$.



(a) Logical-to-physical slice mapping

(b) Logical and physical slices

Figure 8: The layout of the data section

Slices The volumes' slice maps are kept entirely in RAM while the volumes are "live", and persisted in the header section, among the volumes' metadata, in a write-through fashion. This is required for crash consistency.
Writing new slice mappings immediately back to disk does not add a significant I/O overhead, since slice allocations are relatively rare (slices being relatively large chunks of consecutive blocks). The RAM and disk space requirement is also modest, as the size of a single slice map is just $O\left(\frac{N}{S}\log\frac{N}{S}\right)$, because there are $\frac{N}{S}$ slices, each requiring $O\left(\log\frac{N}{S}\right)$ bits to be indexed (N is the total number of blocks). This is in turn due to the very choice of shuffling the storage space with the slice granularity, instead of the block granularity, which would have entailed a position map of $O(N\log N)$ bits.

Each slice mapping is immutable: once a logical slice is assigned to a physical slice, this mapping never changes. This way, we never have to copy data over to its "new location".
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. It follows that, in step 2 of the indirection layer, what actually happens is that either the volume's slice map is consulted (if a mapping already exists), or a new mapping is created and added to the slice map. New slice mappings, however, are only created on `write` requests: in the limit case of a `read` request arriving for a block within an unmapped logical slice (therefore, the block has never been written before), we return a default value (e.g., all zeros) instead of allocating a physical slice; the rationale is to prevent `read` requests from leaving a trace on the disk.
To make this lazy sampling possible, we need an additional, per-device supporting data structure, that holds a list of the device's free physical slices. The size of this list is also $O\left(\frac{N}{S}\log\frac{N}{S}\right)$. The list is exclusively kept in RAM, as it does not need to be persisted. Every time a *device* is instantiated, its list is recreated starting from the full list (containing every physical slice): every time we open a volume within that device, and we reload the volume's slice map, we remove from the list the physical slices mapped to that volume; also, we obviously update the list every time we create a new slice mapping.
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.

IVs 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. Like many other PD schemes, we use random IVs to this effect: for this reason, they need to be stored on the disk. As was mentioned, for each physical slice, we pack the IVs into ΔS physical blocks, stored adjacently to the S data blocks they encrypt. There is a simple static correspondence between a physical block and the on-disk location of its IV: the m-th block within a slice is encrypted by the m-th IV within the ΔS blocks.
This is in contrast with standard, non-PD disk encryption schemes, that generate IVs deterministically from some public context information and the volume's secret key, in order to save the space and the I/O overhead needed to store and retrieve them. In our case, however, we do need random IVs because, as we will see in Section 4.3, we need the possibility to re-encrypt a block by just changing its IV.

The IV of a block is refreshed at each logical `write` for that block, because IV reuse is a very serious error for the CTR mode of operation (since it is a stream cipher). We leave for future research the study of the security implications of possibly refreshing the IVs less frequently.
This strategy introduces a potential performance problem: a naive implementation would translate each logical `write` to a physical `write` of the corresponding physical data block, plus an additional `read`-update-`write` of the **whole** corresponding IV block. This would be severely wasteful in terms of I/O overhead, because we only need to update one IV (16 bytes for AES-CTR), but we are forced to load and store whole blocks (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; instead, we adopt a write-on-flush approach: we intercept the FLUSH requests sent by the upper layer for logical blocks, and only then do we write back the corresponding IV block to disk. This way, we coalesce possibly many updates of the same IV block (triggered by many logical `write`s to the same data block, or by logical `write`s to many blocks within the same slice) into just one physical `write`, thereby lowering the I/O overhead.

Header section Shufflecake statically reserves space at the beginning of the device for max volume headers, irrespective of how many volumes there are effectively. This mild waste of space is necessary in order to to prevent the adversary from deducing the number of volumes from the size of the device header.
Each volume header begins with a random salt: this, combined with the user-supplied password through a KDF, yields the key that encrypts the rest of the header. This way, the header only contains random bytes (for someone that does not have its password); as with TrueCrypt, then, we can fill the unused header slots in the header section with random bytes to make them indistinguishable from actual headers. Another field that is in clear (yet also indistinguishable from random) is a MAC over some encrypted fields: it is used to verify the correctness of the supplied password. The header also contains, among the encrypted fields, the encryption key for the data section of the volume: as with TrueCrypt, the password decrypts the header, which then contains the key to decrypt the data section. Figure 9 depicts the structure of the header section.

Besides the volume's slice map, each volume header contains the password of the "previous" volume. This mechanism provides a hierarchy between volumes: as we have seen, the volumes are all treated equally in the data section, none of them trumps the others. The only ordering between them is

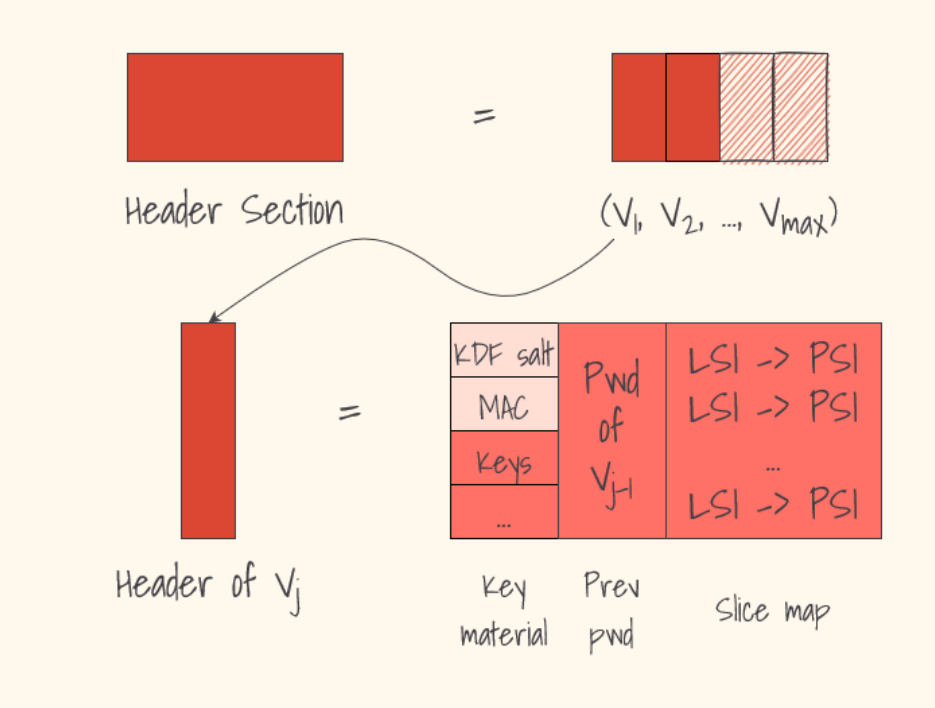


Figure 9: Shufflecake headers

indeed provided by the headers: having the password of the previous volume saved in the header creates a linked list that allows us to more easily mount all the volumes $V_1, \ldots, V_i$ when just the password of volume V_i is provided.
While this approach compromises deniability 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 4.1.3.

Crash consistency As it is right now, Shufflecake is not, strictly speaking, crash-consistent. To fix this, 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. Currently, Shufflecake does not address these concerns, which are left for future improvements.

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. It would not be sufficient, however, 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 is 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) is based on an additional MAC on the data block (stored alongside the IV); this is only 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 is simpler but wastes more disk space, is to store the IV alongside the data block itself, so that the two updates can be merged into a single physical request. 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) onto 9 consecutive physical sectors: the first one contains the IV, the other ones constitute the "data block". This would lead to a fraction of disk space 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 is unused; we could use the rest of the free space to also contain a MAC, 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 assure atomicity of the sector `write`s, but not of the physical requests writing on several adjacent sectors.

4.1.2 Algorithms

Given what has already been discussed, the algorithms for reading and writing are trivial: just follow the slice map (or create a new mapping, if `write` and unassigned), and encrypt/decrypt.

`SflcRead`(V_i, b)
1: `LSI` $\leftarrow b/S$
2: `PSI` $\leftarrow$ `getSliceMapping`$(V_i,$ `LSI`$)$
3: **if** `PSI` $= \perp$
4: **return** $\perp$
5: $\beta \leftarrow$ `PSI` $\cdot\, S' + \Delta S + b\%S$
6: $IV \leftarrow$ `LoadIV`(β)
7: $d \leftarrow$ `DiskRead`(β)
8: **return** `Decrypt`$(V_i.K,\ d,\ IV)$

`SflcWrite`(V_i, b, d)
1: `LSI` $\leftarrow b/S$
2: `PSI` $\leftarrow$ `getSliceMapping`$(V_i,$ `LSI`$)$
3: **if** `PSI` $= \perp$
4: `PSI` $\leftarrow$ `createSliceMapping`$(V_i,$ `LSI`$)$
5: $\beta \leftarrow$ `PSI` $\cdot\, S' + \Delta S + b\%S$
6: $IV \leftarrow$ `SampleAndStoreIV`(β)
7: $d \leftarrow$ `Encrypt`$(V_i.K,\ d,\ IV)$
8: `DiskWrite`$(\beta,\ d)$

Given a logical block b, the logical slice it belongs to is b/S (flooring is implicit), and its offset within the slice is $b\%S$

The function `getSliceMapping`, not shown here for brevity, simply returns the volume's mapping for the given logical slice, if one exists, or $\perp$ in case it does not. The function `createSliceMapping` creates a new slice mapping for the given logical slice. Both functions work by interacting with the write-through in-memory copy of the volume's slice map. As shown by the algorithms, Shufflecake never allocates slice mappings on `read` requests: instead, if the logical slice is yet unmapped (so the

logical block has never been written before), a default, non-error value is returned. Not throwing an error is necessary for the semantic of a volume: although it has never been written before, the block logically exists, it is within the logical size of the volume, so it would be incorrect to return an error. Not allocating a slice mapping is, as we will see in Section 4.1.4, necessary for security, and prevents logical `read`s to leave a trace on the disk.
The functions `LoadIV` and `SampleAndStoreIV` interact with the write-on-flush in-memory LRU cache of IV blocks.

Volume operations In principle, nothing in the scheme inherently prevents us from creating, opening, and closing volumes freely and independently, at any time.
To create a new volume, what is needed is: the index i of the new volume, the chosen password of volume V_i, and the password of volume V_{i-1}. This way, we can correctly format the header by generating the relevant keys, filling the `prev_pwd` field, and initialising the slice map to the empty one. No operation is needed on the data section.
To open a volume, only its index i and its password are needed in order to decrypt the header, which then allows to load its slice map and to decrypt its slices.
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 or metadata.
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: 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. If the password was weak, in order to avoid the adversary later guessing it and decrypting the volume, we can overwrite the header with random bytes. This will erase the encryption key of the data section, which will conclusively solve the problem (the adversary would now have to perform a very difficult *Ciphertext-Only Attack*). This procedure, known as *crypto-shredding* [62], relieves us from the need to overwrite the contents of the data section: instead, we can just overwrite the much-smaller volume header, and still achieve very secure deletion.

Although, as we have seen, volumes can be managed independently, the user actions to be performed might be a bit cumbersome. For this reason, we will see in Section 4.2.2 how our implementation leverages the hierarchy between volumes created by the `prev_pwd` field, and limits the liberty in managing the volumes, to expose a more intuitive set of actions to the user.

4.1.3 Operational Model

In this section, we precisely define the operational model of Shufflecake, to provide a safe *mode of use* allowing the user to retain both 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 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 device's list 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, which would then overwrite their contents.
This can only be avoided by always opening every volume, regardless of which ones we are going to use: if the password to a volume is not provided, it follows from the PD guarantees themselves that the Shufflecake instance has no way of decrypting its slice map, or even to detect that it might exist. It then follows from the overcommitment of the physical storage space that we risk re-using its physical slices for some other volume.
The header's `prev_pwd` field helps in that regard. Our implementation, when given the password of V_i, opens all volumes $V_1, \ldots, V_i$.

General constraints The first thing to do when formatting a device with Shufflecake is to fill it completely with random bytes. Though long and tedious, this operation is crucial for single-snapshot security, as we will see in Section 4.1.4, just like it is for TrueCrypt.
The 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: any clear indication on our side would immediately nullify itself, by giving the adversary an idea about how many volumes to expect on a device (i.e., how many passwords to extort from the user).
The volumes of lower order, that will be disclosed to the adversary, need to 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 could be delegated to a background daemon, but the safest course of action is to offload them to the user.

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

Under investigation When questioned by authorities 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 $(\ell - 1)$-th password under some distress, or after waiting some time: while the exact extent of this is context-dependent, it is clear that it would be suspicious if the user gave up the password to a supposedly-secret volume right away, without being subject to adequate "pressure".

In the light of what has been discussed about data integrity, it is clear that corruption of volume V_ℓ is definitely a possibility when "under investigation", especially if the adversary starts writing files in the decoy volumes. Yet, we notice that this risk is completely unavoidable, as it is due to the combination of:

1. The impossibility to detect more volumes if the passwords are not provided (which follows from the PD security guarantees)
2. The overcommitment of the device's total physical storage space (which is arguably necessary for good disk space efficiency)
3. The obvious necessity not to surrender the ℓ-th password to the adversary

Therefore, the only mitigation against this risk is for the user to frequently backup the disk.

4.1.4 Security Proof

Let us formally prove that this constrution achieves security in the *Arbitrary + One-Time* threat model.
We recall that security in this case essentially boils down to the indistinguishability of the two disk snapshots resulting from the execution of either of two adversary-chosen access patterns, O_0 and O_1: O_1 only contains accesses to decoy volumes $V_1, \ldots, V_{\ell-1}$, while O_0 can also contain accesses to V_ℓ. The only constraint that O_0 and O_1 are subject to is the basic one: they must result in the same content being logically written to volumes $V_1, \ldots, V_{\ell-1}$ at the end of their execution.

In short, the argument can be phrased as follows: whether O_0 or O_1 is executed, the adversary sees the same decryptable content in the volumes $V_1, \ldots, V_{\ell-1}$, placed in uniformly-random physical slices on the disk, and random bytes in the other slices; in the header section, in both cases it manages to decrypt the first $\ell - 1$ volume headers, and sees the rest as random bytes.
The key properties around which security revolves are:

1. the indistinguishability between the encrypted content of V_ℓ and the random bytes used to fill the empty blocks. This is guaranteed by the IND-CPA property of the encryption scheme.
2. The fact that the joint distribution of the slice maps of volumes $V_1, \ldots, V_{\ell-1}$ is the same in the two cases. This ensures that the adversary obtains no information from the physical locations at which he sees decoy data stored. It is shown in the next paragraph.

Proof of point 2 For conciseness, let us call a "hidden access" an access to V_ℓ, and a "decoy access" an access to a decoy volume $V_1, \ldots, V_{\ell-1}$.
This paragraph proves point 2 of the security proof by confuting the following intuition. By interleaving, in O_0, some hidden accesses to the decoy accesses, the attacker can trigger some physical slice allocations that will interfere with the ones needed for the following decoy accesses (some physical slices that would have been free without the hidden accesses will instead be occupied). Therefore, the attacker can observe in the disk snapshot that some physical slices have **not** been allocated for decoy volumes, and somehow conclude that it is due to Shufflecake "going around" them because they are occupied, and thus deduce the existence of the volume V_ℓ.
Indeed, this intuition is wrong: despite the decoy volumes effectively having to go around the hidden volume, the resulting probability distribution on the slices assigned to the decoy volumes is the same whether O_0 or O_1 is executed.

To see this, let us denote by T the set of all physical slices; let us also denote by $T_n = \{(a_1, \ldots, a_n) \in T^n : a_i \neq a_j \; \forall\, i \neq j\}$ the set of n-tuples of pairwise-distinct physical slices. For any access pattern, on any number of volumes, which entails n slice allocations (i.e. writing to n distinct logical slices overall), it holds that the resulting allocated physical slices follow the uniform distribution over T_n (*). This is directly implied by the allocation policy, which randomly picks a physical slice among the free ones.
Let us notice that, from the basic constraint on O_0 and O_1, it follows that both overall write to the same set of logical slices in decoy volumes, since they have to write to the same logical blocks. The `read` requests might differ between O_0 and O_1, but they trigger no slice allocation. Therefore, for decoy volumes, the same logical slices will get mapped, whether O_0 or O_1 is executed.
Let us number these p logical slices $1, \ldots, p$, in an arbitrary order. Let us likewise number $1, \ldots, h$ the h logical slices of V_ℓ accessed by O_0. Call $X_1^b, \ldots, X_p^b \in T$ the random variables representing the physical slices assigned to logical slices $1, \ldots, p$ after the execution of O_b, for both values of $b \in \{0, 1\}$. Call $Y_1, \ldots, Y_h \in T$ the physical slices assigned to the h logical slices of V_ℓ after the execution of O_0.
Our claim is that the joint distribution of $(X_1^0, \ldots, X_p^0)$ is the same as that of $(X_1^1, \ldots, X_p^1)$: this guarantees that the adversary learns nothing from the observed positions at which he sees decoy data. The claim follows by applying (*) to O_0 and O_1. The distribution of $(X_1^1, \ldots, X_p^1)$ is uniform over T_p. The distribution of $(X_1^0, \ldots, X_p^0, Y_1, \ldots, Y_h)$ is uniform over T_{p+h}: the marginal $(X_1^0, \ldots, X_p^0)$ is therefore uniform over T_p. $\square$

4.1.5 Space Utilisation

A few factors influence the disk 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 open-source implementation, we fixed the block size to 4096 bytes, so as to better amortise the linear space overhead determined by the IVs. We chose to limit the total number of blocks in a device to 2^{28}, which limits the total size of the device to 1 TiB. We chose $S = 256$, and max (the maximum number of volumes hosted in a device) equal to 15. Choosing AES-256-CTR as the underlying encryption scheme implies that 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 good space utilisation performance of Shufflecake, we observe that in the case of a 1 TiB disk formatted with Shufflecake, the resulting utilisable space is 1019.91 GiB, equal to more than 99.6% of the physical storage space.

Headers As was discussed, for security reasons, the device's header section must allocate space for a fixed, static number of volume headers, regardless of how many volumes actually are on the device.
For our open-source implementation, we placed the limits discussed above, which result in a volume header size of around $\frac{N}{S} \log \frac{N}{S}$, roughly equal to 4 MiB; having chosen $max = 15$ yields a very reasonable 60 MiB for the total device header size.

IVs As was discussed, IVs need to be stored on the 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 $1/257$ ($< 0.4\%$) 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 the file system over-allocating space for its files, Shufflecake adds another layer of internal fragmentation through its slice mechanism: when a volume (which we can identify with a file system) requests a block, we reserve a whole slice of S 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 some group of slices before requesting a new one.
Luckily, some file systems do exhibit this behaviour. For example, the ext4 file system defines the concept of a block group, i.e., a group consisting of 32,768 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 [37].
This feature plays nicely with the value of $S = 256$ we chose for our implementation: a block group encompasses a whole number of slices, which will therefore not be too internally-fragmented, in the long run.

Releasing unused slices For this solution to be usable in real applications, it needs to 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.
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 [22]. 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.
Since we cannot assume that a block is de-allocated if it has a particular content, like all zero bytes (it would also be expensive to check), we need an additional supporting data structure to keep track of which blocks are in use solely based on the WRITE and TRIM requests coming from the upper layer. This just has to be a bitfield, storing a single bit for each physical block: for a 1-TiB disk, therefore, this amounts to 2^{28} bits, or 32 MiB (per device).

4.2 Implementation and Benchmarks

We implemented the Shufflecake scheme as an open-source device-mapper-based driver for the 5.13 Linux kernel. This section briefly describes the programming environment, it details the structure of our implementation, and presents concrete performance measurements of various metrics, taking the popular `dm-crypt` as a baseline for comparison.

4.2.1 Linux Programming

In Linux, the entirety of the I/O subsystem is part of the kernel, which means that its code executes in privileged mode (CPU protection ring 0). The most efficient way to implement a new device driver is, therefore, by extending the kernel; there exist frameworks that allow to implement drivers in user space, but they invariably come with great performance overheads, since every I/O request entails two context switches (the request always arrives to the kernel, which then has to forward it to the userland application that has to process it and send it back to the kernel).

Kernel modules Linux is a monolithic kernel: it is, at all effects, a single program running with the highest privileges. However, it can be extended at runtime through special shared libraries (that usually bear the `.ko` extension, for kernel object, in their file name), called *kernel modules*.
Unlike userland applications, that are terminated after the execution of their `main()`, kernel modules have an entry point, a function called `module_init()` that is executed when the module is loaded into the kernel (usually through the `insmod` command), and an exit point, a function called `module_exit()` that is executed when the module is removed from the kernel (usually through the `rmmod` command). Between these two moments, the module continues to exist, basically providing functions and variables that are visible to the rest of the kernel (just like a shared library).
Kernel modules are the most common way to implement device drivers (since they can be dynamically loaded and unloaded): in the `module_init()`, they inform some central kernel dispatcher that a new device driver has been added to the system, and provide it with their own callbacks to handle the I/O requests that it receives.

Block device drivers Block devices are storage units that divide the storage space into *blocks*, or *sectors* (usually either 512- or 4096-byte long), and mandate the I/O to be block-aligned: the length and the starting location of each I/O request must be a multiple of the block size. All modern SSDs and HDDs are block devices; also, only block devices can be formatted with file systems, in Linux. Therefore, they constitute the most common type of persistent storage device, and the kernel subsystem handling it (the block layer) is heavily optimised for performance. For these reasons, we implemented Shufflecake as a block device driver.
A block device driver is essentially a set of callbacks, usually provided to the kernel by a kernel module in its `module_init()` function, that intercept software `read()` and `write()` requests coming from the upper layer, and translate them to the appropriate specific READ and WRITE commands for the disk(s) they manage.

The device-mapper framework The device-mapper is a framework (a set of functions, structures, and variables), fully integrated within the Linux kernel, that allows to more easily

implement *stacking drivers*. A stacking driver is one that has no low-level knowledge of any particular physical disk, instead it acts as a translation layer for I/O requests, exposing a *virtual device* whose `read()` and `write()` requests get reworked into requests to another device. Instead of issuing low-level READ and WRITE commands to a particular physical disk, it issues `read()` and `write()` requests to the underlying device driver; as such, it can work on top of any device, because it interacts with its driver.
This is exactly what we need for our Shufflecake implementation: an intermediate translation layer that can act on top of any device to make its physical contents plausibly deniable.

4.2.2 Structure of the Implementation

Our implementation actually consists of two separate components: a kernel module (which does most of the job), and a companion userland application (used to correctly manage the volumes).

The kernel module This is the component that actually implements the scheme, translating logical requests into physical requests, and persisting the volumes' slice maps into the respective headers.
Currently, a reasonably wide subset of the Shufflecake scheme is implemented: we support multiple volumes in a device, we use freshly-random IVs stored on the disk, but we do not implement the TRIM-based slice reclamation procedure, and we do not provide any of the obfuscation mechanism for multi-snapshot security that we discuss later, in Section 4.3.

The userland application This component is only used to manage volumes creation, opening, and closing (no explicit volume destruction functionality is implemented). To this end, it manages the first block (appropriately called the *userland block*) of each volume header, where it stores:

1. A KDF salt used to generate a key-encryption-key (KEK) together with the user-provided password.
2. An encryption and a MAC (using the KEK) of a 64-byte plaintext containing 2 AES keys: the password-encryption-key (PEK) and the volume-encryption-key (VEK).
   (a) The MAC is checked to verify that the entered password is correct
   (b) The ciphertext is decrypted to recover the two keys. The VEK is given to the kernel module, which performs all encryption and decryption using that key. The PEK is used to decrypt the remainder of the userland block.
3. An encryption (using the PEK) of the password of the previous volume. This chains volumes in a linked list, creating a hierarchy between them.

The other blocks of the volume header are managed by the kernel module, and contain the volume's slice map encrypted with the VEK.
The first block is managed by the userland application since 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 Scrypt [1] and Argon2 [10] are not currently implemented in the Linux Kernel Crypto API [42], while they are available in user-space software libraries like Sodium [53].
The design intentionally decouples keys (especially the KEK and the VEK): this allows, for example, the user to change the password to a volume without having to re-encrypt it with a different key (even though this functionality is currently not implemented).

Volume operations We discussed how, in principle, the scheme would permit volumes to be managed independently; this flexibility is reflected in the kernel module, which allows to open and close volumes from a device at any time through the appropriate `ioctl` requests (which constitute the communication channel with user space). We also discussed, however, how a limitation of this freedom yields a less complex set of actions exposed to the user, and prevents data loss; for this reason, the userland application purposely hides this flexibility to the user, and forces them to create, open, and close all volumes on a device at the same time.

Specifically, the `create` command interactively takes ℓ passwords as input (and a device path), 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 `--no-init` option is provided, 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 `open` command takes just one password as input (and a device path), 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 `prev_pwd` field in the userland block of each volume header). This is the command that actually creates the Linux virtual devices representing the volumes, under `/dev/mapper`: the names are generated algorithmically.
The `close` command only takes the device path as input, and closes all the volumes open on that device.
Other possible commands, that are not currently implemented as they are not vital for the scheme, could be one to delete a volume (or just to free up its slices), and one to change a volume's password. Both of them would need to take particular care in keeping the "hierarchy" of volumes induced by the `prev_pwd` field in a consistent state, by propagating the changes on the affected volume to the relevant "neighbours".

4.2.3 Benchmarks

We tested our implementation using the common `fio` benchmarking tool, used to flexibly measure various I/O metrics.
We ran the tests in two different settings. In both cases, the OS was Ubuntu 22.04 equipped with the 5.15 Linux kernel. In the first scenario, the machine was a VirtualBox VM (the CPU of the host being an Intel Core i5-4200U), and the underlying physical devices for Shufflecake were file-backed loop devices; in the second scenario, the machine was a native Ubuntu PC (with an Intel Core 2 Duo CPU), and the underlying physical devices for Shufflecake were physical partitions of the integrated SSD.

Against the baseline We used `dm-crypt`, the default, non-PD disk encryption tool (which, incidentally, is also a device-mapper target) as a baseline for comparison.
In both settings (VM and native machine), we used 4 of `fio`'s standard workloads: sequential reads, sequential writes, random reads, random writes. The results are synthesised in Table 1.

At a glance, we can immediately see that performance is better in the VM than on the native machine (probably because of the more recent CPU), both in absolute and in relative terms.
In both cases, though, the overhead for reads is significantly lower than for writes. A plausible reason for this imbalance is an implementation detail: `write` requests are unconditionally piped

	VirtualBox VM			Native machine		
	`dm-crypt`	Shufflecake	Overhead	`dm-crypt`	Shufflecake	Overhead
Seq. Reads	85.5	82.2	1.04x	35.8	30.3	1.18x
Seq. Writes	82.8	54.4	1.52x	69.5	26.8	2.59x
Rand. Reads	27.8	38.1	0.73x	26.4	11.1	2.38x
Rand. Writes	34.6	21.6	1.60x	28.6	8.0	3.58x

Table 1: Measured throughputs (in MiB/s) and corresponding performance overheads.

through a workqueue, whereas a possible optimisation would reduce the need for this quite dramatically, especially for sequential writes; `read` requests, on the other hand, never go through a workqueue. Still, this is a hypothetical explanation, since no fine-grained analysis based on micro-benchmarks has been carried out to validate it.
For sequential reads in the VM, Shufflecake fares even better than the baseline: this might be due to some fluctuations (although the experiment has consistently confirmed it), but it shows that Shufflecake is potentially competitive enough to outperform `dm-crypt` in some scenarios.

Against HiVE and DetWoOram The comparison with the most popular ORAM-based PD solutions more convincingly shows that Shufflecake is a valid option to be deployed in real-world systems.

Regarding I/O performance, we can synthesise the previous table by saying that Shufflecake achieves a slowdown of 1x-3x over `dm-crypt`. On the other hand, HiVE [12] (a famous scheme from 2014) had a heavy 200x I/O overhead. DetWoOram [47] (a very efficient scheme from 2017) had an overhead of 2.5x for `read`s and 10x-14x for `write`s.

To give a more complete picture, let us also mention the figures for disk space utilisation, a long-standing weakness of Write-Only ORAMs. While Shufflecake utilises 99.6% of the total storage space for user data, most ORAMs waste a large fraction of the disk space in order to keep their performance overheads low. Specifically, HiVE utilises 50% of the disk, while DetWoOram utilises 25%.

4.3 Ideas for 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.
In this section, we explore the possibility to achieve some degree of unproven, operational security in the multi-snapshot setting through some separate, "orthogonal" 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.

4.3.1 Security Game Revisited

Let us clarify what the multi-snapshot PD game concretely translates to for the Shufflecake scheme, and how an obfuscation procedure will make it operationally hard to win for an adversary. The rationale behind contenting ourselves with this reduced level of security is the hope that even though the distinguishing advantage of the adversary is not negligible (and so the scheme is not cryptographically secure in the strictest sense), it is still low enough for the findings of the attack to be inconclusive before a judge. In other words, we argue that "legal" or anyway "operational" security, in the context of PD, is not the same as "cryptographic" security in the theoretical sense.

Physical slices have statically-known boundaries, so the adversary can compare the disk snapshots he has on a slice basis. This amounts to inspecting, for each physical slice, the subsequent deltas it goes through across snapshots.
Recall that a slice is essentially an array of S 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 blocks have changed: *how* they have changed is completely inconsequential and uninformative. In other words, the delta between two snapshots of a physical slice boils down to a bitmask of S bits, representing which blocks have changed and which stayed 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 deltas 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 deltas 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 delta 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 delta bitmask in the free slices, through a simple re-randomisation of a few select blocks in the slice. 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 delta 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 delta 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 delta bitmask into zeros. Instead, we can turn some zeros into 1s by simply re-encrypting the same content of a block with a different IV.

4.3.2 Trivial Random Refresh

A very simple 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 `Unmount`, or, for better resilience, spread across the normal operations of the scheme.
It is the easiest way to achieve a non-zero delta bitmask for free slices, but it is definitely too crude to work: nothing guarantees that the delta 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 delta bitmask.

Such considerations immediately suggest 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, again independently at random, each with probability q. This way, we also randomise the delta bitmasks of the data slices, which makes 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 delta bitmasks of data slices and free slices roughly indistinguishable (one could even run a quantitative analysis to bound the statistical distance between the two distributions), but only if the adversary gets just one delta for each slice, i.e., if he only gets two snapshots.
This is because we are essentially playing a hopeless game: roughly speaking, we are aiming at making signal+noise (the delta of a data slice) indistinguishable from noise (the delta of a free slice). As was discussed, we cannot turn the 1s of the delta bitmasks of data slices into zeros, 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 zeros 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 by the upper layer (maybe because it contains some sort of file system index): the corresponding bit in the delta 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.

4.3.3 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 deltas of free slices look like they were also generated by some file system workload. This way, we are still making the deltas for the two kinds of slices similar, but we are not trying to erase the signal from data slices, which we cannot. Instead, we "copy" it onto the free slices.

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.
Instead, a simpler method that is also more likely to capture the patterns we want to imitate is to

have the scheme itself "learn" them online, by means of simply subsampling the stream of incoming logical requests.
For each incoming logical `write` 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 ($b \mod S$) 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 delta 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, instead of 2ℓ.
We can ideally assign the free slices to $max - \ell$ 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 $max - \ell$ volumes with the imitated logical `write` requests.

4.3.4 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 $max - \ell$ fake volumes" into a concrete algorithm for choosing a target free slice when we imitate a `write` request.
A valid, if "exotic", alternative, would be to actually create $max - \ell$ 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, it might be challenging to manage slices disappearing from ghost volumes, especially for their file systems which might suddenly get corrupted and start complaining quite loudly, thus impairing the practical usability of the system.

4.4 Conclusions and Future Work

We have shown how Shufflecake fills a hole in the current landscape of existing PD schemes, and how, compared to them, it strikes a more balanced compromise between performance and security, making it suited for adoption into real-world applications.
Nonetheless, there is still room for future research to improve over its current state.

The high-level ideas that were provided for rough multi-snapshot security could be better formalised, defined, and analysed, both in terms of I/O overhead, and in terms of the actual security they provide. As was discussed, "operational", "legal" security is still desirable, and could very well require much less overhead than cryptographic security.

The implementation could also be refined, both in terms of robustness by porting it to even more recent kernels, and in terms of performance (the memory footprint could be slightly lowered, and so could the overhead for `write` requests).

References

[1] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. Scrypt is maximally memory-hard. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 33–62. Springer, 2017.

[2] Elia Anzuoni and Tommaso Gagliardoni. Shufflecake repository. https://codeberg.org/shufflecake, 2022.

[3] Elia Anzuoni and Tommaso Gagliardoni. Shufflecake website. https://shufflecake.net, 2022.

[4] Anushka Ashtana. Revealed: British councils used ripa to secretly spy on public. https://www.theguardian.com/world/2016/dec/25/british-councils-used-investigatory-powers-ripa-to-secretly-spy-on-public, 2016.

[5] Open Crypto Audit. Truecrypt security audit report. https://opencryptoaudit.org/reports/TrueCrypt_Phase_II_NCC_OCAP_final.pdf, 2015.

[6] Austen Barker, Yash Gupta, James Hughes, Ethan L Miller, and Darrell DE Long. Rethinking the adversary and operational characteristics of deniable storage. *Journal of Surveillance, Security and Safety*, 2(2):42–65, 2021.

[7] BBC. Man jailed over computer password refusal. https://www.bbc.com/news/uk-england-11479831, 2010.

[8] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In *Annual International Cryptology Conference*, pages 341–358. Springer, 1994.

[9] Alex Biryukov. Block ciphers and stream ciphers: The state of the art. *Cryptology EPrint Archive*, 2004.

[10] Alex Biryukov. Argon2. https://www.password-hashing.net/#argon2, 2013.

[11] John Richard Black Jr. *Message authentication codes*. University of California, Davis, 2000.

[12] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214, 2014.

[13] Mariusz Borowski and Marek Leśniewicz. Modern usage of "old" one-time pad. In *2012 Military Communications and Information Systems Conference (MCC)*, pages 1–5. IEEE, 2012.

[14] Jason Burke. Kenyan election official 'tortured and murdered' as fears of violence grow. https://www.theguardian.com/world/2017/jul/31/kenyan-election-official-christopher-msando-dead-before-national-vote, 2017.

[15] Dominic Casciani. Why cage director was guilty of withholding password. https://www.bbc.com/news/uk-41394156, 2017.

[16] Anrin Chakraborti and Radu Sion. Sqoram: Read-optimized sequential write-only oblivious ram. *arXiv preprint arXiv:1707.01211*, 2017.

[17] Anrin Chakraborti, Chen Chen, and Radu Sion. Datalair: Efficient block storage with plausible deniability against multi-snapshot adversaries. *Proc. Priv. Enhancing Technol.*, 2017(3):179, 2017.

[18] Chen Chen, Anrin Chakraborti, and Radu Sion. Pd-dm: An efficient locality-preserving block device mapper with plausible deniability. *Proc. Priv. Enhancing Technol.*, 2019(1):153–171, 2019.

[19] Chen Chen, Xiao Liang, Bogdan Carbunar, and Radu Sion. Sok: Plausibly deniable storage. https://arxiv.org/abs/2111.12809, 2021.

[20] D. Coppersmith. The data encryption standard (des) and its strength against attacks. *IBM Journal of Research and Development*, 38(3):243–250, 1994. doi: 10.1147/rd.383.0243.

[21] Morris J Dworkin. Sp 800-38c. recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality, 2004.

[22] Tasha Frankie, Gordon Hughes, and Ken Kreutz-Delgado. A mathematical model of the trim command in nand-flash ssds. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE '12, page 59–64, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312035. doi: 10.1145/2184512.2184527. URL https://doi.org/10.1145/2184512.2184527.

[23] FreeOTFE. Freeotfe. https://www.wikiwand.com/en/FreeOTFE, 2010.

[24] Clemens Fruhwirth. *New methods in hard disk encryption*. na, 2005.

[25] Tommaso Gagliardoni. An introduction to oblivious ram (oram). https://research.kudelskisecurity.com/2020/04/22/an-introduction-to-oblivious-ram-oram/, 2020.

[26] Richard M. George. Nsa's role in the development of des. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, pages 120–120, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-19074-2.

[27] Matthew Green. Let's audit truecrypt. https://blog.cryptographyengineering.com/2013/10/14/lets-audit-truecrypt/, 2015.

[28] Ubuntu Privacy Group. Truecrypt analysis. https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwj5-tL9kej5AhW5QPEDHa-LBYIQFnoECAMQAQ&url=https://cyberside.net.ee/truecrypt/misc/truecrypt_7.0a-analysis-en.pdf&usg=AOvVaw1GsyCMngjoKdytF_IqVrdN, 2011.

[29] Simon Heron. Advanced encryption standard (aes). *Network Security*, 2009(12):8–12, 2009.

[30] In Re Boucher. In re boucher — Wikipedia, the free encyclopedia. [Online; accessed 21-June-2022] Available at: https://en.wikipedia.org/wiki/In_re_Boucher, 2022.

[31] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Ndss*, volume 20, page 12. Citeseer, 2012.

[32] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Deftl: Implementing plausibly deniable encryption in flash translation layer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2217–2229, 2017.

[33] Tara Merin John, Syed Kamran Haider, Hamza Omar, and Marten Van Dijk. Connecting the dots: Privacy leakage via write-access patterns to the main memory. *IEEE Transactions on Dependable and Secure Computing*, 17(2):436–442, 2017.

[34] Key Disclosure Law. Key disclosure law — Wikipedia, the free encyclopedia. [Online; accessed 21-June-2022] Available at: https://en.wikipedia.org/wiki/Key_disclosure_law, 2022.

[35] Lichun Li and Anwitaman Datta. Write-only oblivious ram-based privacy-preserved access of outsourced data. *International Journal of Information Security*, 16(1):23–42, 2017.

[36] Linux. Dm-crypt. https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt, 2022.

[37] Linux. ext4 high level design. https://docs.kernel.org/filesystems/ext4/overview.html, 2022.

[38] Helger Lipmaa, Phillip Rogaway, and David Wagner. Ctr-mode encryption. In *First NIST Workshop on Modes of Operation*, volume 39. Citeseer. MD, 2000.

[39] Andrew D McDonald and Markus G Kuhn. Stegfs: A steganographic file system for linux. In *International Workshop on Information Hiding*, pages 463–477. Springer, 1999.

[40] David McGrew and John Viega. The galois/counter mode of operation (gcm). *submission to NIST Modes of Operation Process*, 20:0278–0070, 2004.

[41] Microsoft. Bitlocker. https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview, 2022.

[42] Stephan Mueller. Kernel crypto api. https://www.kernel.org/doc/html/v4.16/crypto/index.html, 2022.

[43] Rafail Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 514–523, 1990.

[44] Timothy M Peters, Mark A Gondree, and Zachary NJ Peterson. Defy: A deniable, encrypted file system for log-structured storage. 2015.

[45] Fedora Project. Luks. https://docs.fedoraproject.org/en-US/quick-docs/encrypting-drives-using-LUKS/, 2022.

[46] RIPA. Regulation of investigatory powers act 2000 — Wikipedia, the free encyclopedia. [Online; accessed 20-June-2022] Available at: https://en.wikipedia.org/wiki/Regulation_of_Investigatory_Powers_Act_2000, 2022.

[47] Daniel S Roche, Adam Aviv, Seung Geol Choi, and Travis Mayberry. Deterministic, stash-free write-only oram. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 507–521, 2017.

[48] Rubber-hose cryptanalysis. Rubber-hose cryptanalysis — Wikipedia, the free encyclopedia. [Online; accessed 21-June-2022] Available at: https://en.wikipedia.org/wiki/Rubber-hose_cryptanalysis, 2022.

[49] Rainer A Rueppel. *Analysis and design of stream ciphers*. Springer Science & Business Media, 2012.

[50] Claude E Shannon. Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715, 1949.

[51] Gustavus J Simmons. Symmetric and asymmetric encryption. *ACM Computing Surveys (CSUR)*, 11(4):305–330, 1979.

[52] Rajeev Sobti and Ganesan Geetha. Cryptographic hash functions: a review. *International Journal of Computer Science Issues (IJCSI)*, 9(2):461, 2012.

[53] Sodium. Libsodium. https://doc.libsodium.org/, 2013.

[54] William Stallings. Nist block cipher modes of operation for confidentiality. *Cryptologia*, 34(2): 163–175, 2010.

[55] Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.

[56] TrueCrypt. Truecrypt licence. https://github.com/FreeApophis/TrueCrypt/blob/master/License.txt, 2015.

[57] TrueCrypt. Truecrypt homepage. https://www.truecrypt71a.com/, 2015.

[58] TrueCrypt. Truecrypt volume format. https://www.truecrypt71a.com/documentation/technical-details/truecrypt-volume-format-specification/, 2015.

[59] United States v. Fricosu. United states v. fricosu — Wikipedia, the free encyclopedia. [Online; accessed 21-June-2022] Available at: https://en.wikipedia.org/wiki/United_States_v._Fricosu, 2022.

[60] VeraCrypt. Veracrypt homepage. https://www.veracrypt.fr/en/Home.html, 2022.

[61] Mark Ward. Campaigners hit by decryption law. http://news.bbc.co.uk/2/hi/technology/7102180.stm, 2007.

[62] Wikipedia. Crypto-shredding. https://en.wikipedia.org/wiki/Crypto-shredding, 2022.

[63] Frances F Yao and Yiqun Lisa Yin. Design and analysis of password-based key derivation functions. In *Cryptographers' Track at the RSA Conference*, pages 245–261. Springer, 2005.

[64] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. *ACM SIGOPS Operating Systems Review*, 38(5):72–84, 2004.

[65] Aviad Zuck, Udi Shriki, Donald E Porter, and Dan Tsafrir. Preserving hidden data with an ever-changing disk. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 50–55, 2017.