Chen Chen, Xiao Liang, Bogdan Carbunar*, and Radu Sion

SoK: Plausibly Deniable Storage

Abstract: Data privacy is critical in instilling trust and empowering the societal pacts of modern technology-driven democracies. Unfortunately it is under continuous attack by overreaching or outright oppressive governments, including some of the world's oldest democracies. Increasingly-intrusive anti-encryption laws severely limit the ability of standard encryption to protect privacy. New defense mechanisms are needed.

Plausible deniability (PD) is a powerful property, enabling users to hide the existence of sensitive information in a system under direct inspection by adversaries. Popular encrypted storage systems such as TrueCrypt and other research efforts have attempted to also provide plausible deniability. Unfortunately, these efforts have often operated under less well-defined assumptions and adversarial models. Careful analyses often uncover not only high overheads but also outright security compromise. Further, our understanding of adversaries, the underlying storage technologies, as well as the available plausible deniable solutions have evolved dramatically in the past two decades. The main goal of this work is to systematize this knowledge. It aims to: (1) identify key PD properties, requirements and approaches; (2) present a direly-needed unified framework for evaluating security and performance; (3) explore the challenges arising from the critical interplay between PD and modern system layered stacks; (4) propose a new "trace-oriented" PD paradigm, able to decouple security guarantees from the underlying systems and thus ensure a higher level of flexibility and security independent of the technology stack.

This work is meant also as a trusted guide for system and security practitioners around the major *challenges* in understanding, designing and implementing plausible deniability into new or existing systems.

Keywords: Plausible Deniability, Encryption Backdoor, Censorship, Censorship Resilience, ORAM

DOI Editor to enter DOI

Received ..; revised ..; accepted ...

Chen Chen: Stony Brook University, cynthia.us12@gmail.com

Xiao Liang: Stony Brook University, xiao.crypto@gmail.com

***Corresponding Author: Bogdan Carbunar:** FIU, carbunar@gmail.com

Radu Sion: Stony Brook University, r@zxr.io

1 Introduction

Data privacy has become essential maybe more so than at any other time in human history. Encryption can be used to defend against unauthorized disclosure of sensitive data, yet is not enough to handle adversaries empowered by law or rubber-hose (e.g. oppressive governments) to coerce the user into revealing encryption keys.

Unfortunately, numerous real-life examples show that protecting sensitive data in the presence of such coercive adversaries is often a matter of life and death. The Human Rights Group Network for Human Rights Documentation at Burma (ND-Burma) [34] documented large numbers of human rights violations. Proof was carried out of the country on mobile devices by ND-Burma activists, risking exposure at checkpoints and border crossings. In 2012, a videographer could smuggle evidence of human rights violations out of Syria by hiding a micro-SD card in a wound on his arm [31] etc. Threats of coercive attacks are not merely an Orwellian fantasy, but a real concern [3, 25, 36, 37, 43, 46, 49].

To address this, *plausible deniability* (PD) has been proposed. It is a powerful property, enabling users to hide the existence of sensitive information on a system under inspection by overreaching or coercive adversaries, democratically elected or otherwise.

In the context of secure storage[1], PD refers to the ability of a user to plausibly deny the existence of certain data stored on a storage device even when an adversary has access to the device. Since adversaries cannot conclude anything about the existence of sensitive data, they have no good excuse to perform the coercion further, thus leaving those data in safety.

PD was first proposed in 1998 [3]. Since then, popular encrypted file systems (FSes) such as TrueCrypt [2] (first released in 2004) and other PD research results have emerged [4, 24, 30, 33, 34, 42] attempting to balance the ever present security-efficiency trade-off.

Unfortunately, existing efforts were designed for very specific adversaries and contexts, and under some-

1 PD has been first formalized in a (mostly theoretical) context of encryption [7, 32], often involving small amounts of data and sometimes read-only. This work focuses on applied aspects as they relate to efficient, modern, high-capacity data storage.

times unclear security models and device assumptions. However, to achieve strong PD guarantees, it is important to understand and evaluate these contexts and limitations properly. This work aims to systematize knowledge and provide a more in-depth understanding for today's practitioners, and future research.

1.1 Challenges

Before diving in, it is important to understand some of high-level challenges facing plausibly deniable systems researchers and practitioners.

Security-Efficiency Trade-Off. Real-Life Adversaries. Previous PD literature has been focusing on *single-snapshot* adversaries who can check the storage device only once, and *multi-snapshot* adversaries who can checks the device at several different time points. While the former are relatively easy to handle (proof being practical systems such as TrueCrypt [2]), *practical* PD systems resilient against multi-snapshot adversaries turns out to be more difficult to design.

Ideally, researchers would like to obtain multi-snapshot security against all probabilistic polynomial time (PPT) adversaries[2] (referred to as "full security"). However, until today, only a few constructions [4, 9, 14] achieve this level of security, but are unfortunately significantly slower than the underlying storage device. Other solutions seek better performance by relaxing the security requirements. For example, some of them assume that a small area on the device is hidden from the adversary, and some put certain restrictions on the adversarial behavior (see Sec. 4 for details).

Overall, unfortunately, no *practically efficient* construction achieves multi-snapshot PD with *full security*. This may be also because existing adversarial models and associated solutions have been developed mostly ad-hoc and not designed to answer more general, fundamental questions regarding the security-efficiency trade-off. For example, is there a performance bottleneck inherent to the concept of PD? Are wORAMs necessary to achieve fully-secure PD? Are there multiple dimensions along which the PD security-efficiency trade-off can be optimized? We believe that answers to these questions are critical for both practitioners of today aiming to build in plausible deniability into modern system stacks, as well as for upcoming research in PD.

Dependency on System Layers. To complicate things further, modern systems feature layered structures all of which persist state and can compromise any security guarantees aimed for by other layers. Consider that ubiquitous stack of a typical FS, FS caches, LVM layers, LVM caches, block-devices (BD), block device caches, and flash translation layers (FTL) (see Sec. 2.2 for additional details). Existing PD works consider only a specific layer, e.g., DEFY [34] builds PD in the FS layer, TrueCrypt [2] works in the BD layer, DEFTL [24] works in the FTL layer.

Further, most schemes make ad-hoc case-specific assumptions about the devices and the adversary behavior, accordingly achieving PD in a restricted sense.

Such a layered structure complicates the security analysis. Schemes designed for a specific layer may lose their security guarantees if deployed at a "wrong" layer. As will be shown in Sec. 3.1, this fact can sometimes be overlooked unintentionally. Further, the existence of state in the other layers cannot be ignored since it often contains compromising information breaking the security of the overall scheme.

In most cases, **a realistic adversary with visibility into the state of one or more additional layers, may immediately compromise single-layer designs since the additional state can reveal access patterns and other security-sensitive information that a single-layer model simply cannot capture.**

It is thus critical to investigate the interplay between PD security and layers, and provide constructions and definitions with reduced or zero dependency on layers. Ideally, such an investigation can isolate PD as an independent security concept, and not only a layer/device-dependent property (Sec. 3 and 3.1).

Lack of Unified Security Framework. As discussed, full security as defined in [4] is achieved by only a few constructions which feature prohibitive performance overheads. Most other schemes restrict adversaries significantly and do not provide strong security or allow even for a comparative analysis of security. Very often also, the security arguments for such schemes contain heuristics, a very dangerous practice. For example, the security of DEFY [34] relied on the authors' claim that the hidden pages in their scheme were indistinguishable from secure-deleted public pages. However, with no formal proof given, it was not clear whether the asserted indistinguishability really held against all PPT

2 Security against all PPT adversaries is the golden rule for most cryptographic primitives and security tasks, e.g. one-way functions, encryption schemes, digital signatures etc.

coercive adversaries. Subsequently Jia et al. [24] showed that DEFY can be easily compromised with very little effort (if adversaries make several attempts to exhaust writing capacity).

Moreover, due to the lack of a unified security framework, different papers customize the definition of PD to serve their specific application or devices, making comparisons between systems difficult or outright impossible. This further leads to an unnecessary proliferation of threat models and definitions, with a polymorphous-yet-confusing naming style. For example, PD schemes deployed in the FS layer are called "steganographic file system" or "deniable file system", while schemes designed for the BD layer are named "hidden volume encryption" or "deniable encryption". In selecting a proper plausible deniability mechanism for their application, practitioners end up bewildered by such multifarious names, and the lack of structure or relationships among the security guarantees provided by those schemes. It is essential to unify these adversarial definitions and application scenarios, and thus enable comparison-based evaluations.

1.2 Contributions

This work synthesizes existing ideas into a guide for system and security practitioners helping to understand, design or implement plausible deniability into new or existing systems. Concretely:

1. We observe that a key point of PD lies in concealing users' hidden data access patterns. Often this happens using randomized (ORAMs) or canonical form I/O. We examine how these approaches affect the security and efficiency of the resulting PD schemes. We also survey another approach appeared recently—basing the secrecy of access patterns on inherent properties of storage systems/devices. This approach usually leads to lightweight solutions that are "native" to the underlying systems/devices.
2. We investigate the interplay between security assurances, adversarial models and modern multi-layer storage stacks. This reveals a set of general principles and definitions that can be deployed for better security-efficiency trade-offs.
3. We propose the concept of trace-oriented security to enable the design and evaluation of schemes providing layer-independent security guarantees. We show that trace-oriented security was achieved (though not claimed explicitly) by a few existing constructions [4, 9, 14]. We show that this stronger security notion comes with a price—equivalence to write-only ORAMs.
4. We provide a way to unify and evaluate solutions under a single framework, where the main differences are expressed as constraints on the power of the adversary. Saliently, this unified point of view provides a framework for the comparison and evaluation of PD solutions. We present a taxonomy of security for existing constructions.
5. Finally, we identify important under-explored areas, and suggest new directions for future research.

2 Model

In this section, we provide the problem setup for PD, then describe the system and adversary model.

2.1 The Plausible Deniability Problem

Plausibly deniable storage systems need to allow users to store public data, *and* sensitive hidden data. Public data does not require protection, and is potentially known by the adversary. Hidden data needs to be protected against coercive adversaries who can compel the user to hand over secret information (e.g. encryption keys). Under duress, the user may need to provide some information (e.g. keys to public data) that dismisses the adversary's suspicions, while most importantly denying the existence of the hidden data.

2.2 System Model

Modern storage systems comprise multiple layers that link the physical storage medium and the user applications. Example layers include the file system (FS), block device layer (BD), device mapper, flash translation layer (FTL), and the physical device, e.g., NAND flash or block device. The File System (FS) layer is mandatory. It organizes data as files for better management. The Block Device (BD) layer provides abstractions for block devices and maps multiple "virtual volumes" onto one block device, where a volume can be, for instance, a file system. It is optional and needed only if a block device is deployed as the storage media (e.g., the device mapper in the Linux kernel). Another optional layer is the Flash Translation (FT) layer, needed if NAND flash is used as the physical device.

Not every storage system contains all the above layers. For example, if the physical storage medium is a NAND flash, then the storage system could consist of an FS layer only, or both an FS layer and an FTL layer.

Operation Traces. The storage device allows `Read` and `Write` operations. An operation trace is an ordered sequence of operations the system performs on the physical device, independent of layers or device properties. For instance, the operation trace (`Read`, l_1), (`Write`, l_1, d_1), (`Write`, l_2, d_2) first reads the data from location l_1, then writes data d_1 into l_1 and d_2 into location l_2.

In the following, we assume that `Read` operations do not modify the storage medium. Since only `Write` operations leave traces on the storage medium, `Write` is the only type of operation that the trace-oriented definition introduced in Sec. 3 needs to consider.

Mount and Unmount. Several PD systems require users to `Mount` and `Unmount` volumes or partitions in order to switch between accessing public and hidden data [15, 16, 24], and even require them to `Unmount` the hidden partition before handing the device to the adversary.

2.3 Adversary Model

We first detail common adversary assumptions. We then provide a classification of adversaries based on their capabilities, and introduce a novel, trace-oriented adversary. Further, we describe the standard, CPA game-inspired, plausible deniability definition.

Adversary Assumptions and Capabilities. Adversaries are assumed to be able to access the device of a user, and attempt to compromise deniability, i.e., determine if the user is storing any secret data. Adversaries are generally assumed to be computationally bounded.

Adversaries are assumed to know the design of the deployed PD solution. They are also assumed not to know how many keys are used in the system, and to not have access to hidden user passwords or encryption keys. However, they can request the user to reveal passwords and encryption keys. The user is assumed in this case to reveal public passwords (including providing root privilege) and public keys. Adversaries can use such information to access and decrypt stored data. In addition, adversaries can use password cracking programs and perform forensics on the disk image.

Additional assumptions have been introduced to accommodate the applications or to trade for better performance. Those that significantly affect the design choices for PD are discussed as follows.

- **A1:** Adversaries are rational. Namely, an adversary will stop further coercion if it cannot prove the existence of any unrevealed data.
- **A2:** Adversaries cannot observe run-time system state (e.g., DRAM, caches).
- **A3:** Adversaries cannot perform malicious code injection on the system used by the user.

Assumption A1 draws a line on the adversary's coercive behavior, and was made (sometimes implicitly) in the majority of exiting work. Assumptions A2 and A3 limit PD to disk states only. This captures a wide class of application scenarios, including the motivational examples in the introduction.

Adversary Classification. Adversaries can be classified based on the data they can access on the user device:

- **Snapshot-Oriented Adversary**. The typical adversary is snapshot-oriented. Such an adversary can only access snapshots of the physical device.
- **Trace-Oriented Adversary**. We introduce a novel, trace-oriented adversary, that can access not only device snapshots, but also the operation traces (Sec. 2.2) that produce them.

Traces are the result of probabilistic polynomial time (PPT) run-time computations on user requests, i.e., sequences of compliant logical instructions to be executed at a layer (Sec. 2.2). For instance, traces at the BD layer traces may include block `Read` and block `Write` instructions, while at the FTL layer, traces may include page `Read`, page `Write`, and block `Erase` instructions. This is in contrast to run-time system state that includes the contents of memory and caches.

An example where an adversary can capture trace data is in flash. SSDs implement an FTL layer inside-the-box that *sees* all operation traces (e.g., which inode pages are updated) before they are executed on the actual flash cells. However, the complex wear-leveling logic inside the FTL maintains state both as meta information and on the device itself (e.g., un-mapped not-yet-`Erased` blocks containing compromising old data) that, when inspected, can directly reveal critical information about past traces or even the traces themselves.

Snapshot-oriented adversaries can be further classified based on their number of opportunities to inspect the user device:

- **Single-Snapshot Adversary**. This adversary can see the device only once before eventually confronting the user and demanding access to information. This makes the design of efficient PD schemes significantly easier. Indeed, a single snapshot (i.e., of a randomized encrypted device) does not leak

much (if any) information beyond its size. For PD then, it may be sufficient to hide the sensitive data "encrypted"[3], e.g., indistinguishable from random "free" device areas.

– **Multi-Snapshot Adversary**. This adversary can take multiple snapshots of the device at different time points [17]. Examples multi-snapshot adversaries include customs officers or hotel personnel with regular access. Data center servers may also face inspections by overreaching authorities empowered by rubberhose or ill-devised laws. For multi-snapshot PD, it is exponentially more difficult to balance the security-efficiency trade-off. Exploring this will be one of the main themes of Sec. 3.1 and 4.

Standard, CPA-Game for PD. We now briefly describe the first formal definition of PD introduced by Blass et al. [4] and refined in [9]. The definition of PD is provided through a cryptographic game, analogous to the one used to define encryption against *chosen-plaintext attacks* (CPA). We expand this to provide a unified definition of PD in Sec. 3.

The security game is played between a coercive adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ running the underlying PD scheme Σ. The adversary holds the credentials needed to access the public data, but is ignorant of the ones for hidden data. At the beginning, $\mathcal{C}$ picks a random bit $b \xleftarrow{\$} \{0,1\}$. $\mathcal{A}$ is allowed to interact with $\mathcal{C}$ for polynomial-many rounds. In each round, $\mathcal{A}$ issues access patterns $\mathcal{P}^0$ and $\mathcal{P}^1$ that share the same access requests to public data, but may contain different access requests to the hidden data. $\mathcal{C}$ will always execute $\mathcal{P}^b$. At the end of these interactions, $\mathcal{A}$ gets the snapshot of the physical device. $\mathcal{A}$ wins if it can guess the value b correctly. The scheme Σ is said to achieve single-snapshot PD if the winning probability of $\mathcal{A}$ is $\leq \frac{1}{2} + \mathsf{negl}(\lambda)$, where $\mathsf{negl}(\lambda)$ is a *negligible function* on the security parameter λ.[4] This game extends to capture multi-snapshot security by allowing $\mathcal{A}$ to access the device state at the end of each round.

3 Unified PD Definition

3.1 Independence of Storage Layers

The layered structure of modern storage systems (Sec. 2.2) complicates the design of PD schemes. Yet, this is often overlooked and has not been studied in a systematic way. In the following we investigate how the security of PD solutions is affected by storage layers. We also introduce a new "trace-oriented" definition for PD. In the standard PD definition (Sec. 2.3) the adversary gets to see snapshots of devices; A trace-oriented notion allows the adversary to also learn operation traces. Trace-oriented PD provides stronger security guarantees and more flexible deployment choices due to its reduced dependence on storage layers.

Layer-Specific PD Solutions are Vulnerable. Because of the layered nature of modern technology stacks, PD solutions are often designed for a target layer $\mathcal{L}$, e.g., $\mathcal{L}$ could be FS, BD or FTL. Then, in the security analysis, even the very existence of the underlying layers is often simply ignored. Unfortunately this results in designs that can be easily compromised by an adversary with access to operation traces (see example of trace-oriented compromise in Sec. 2.3).

In general, an adversary with visibility into the state of one or more other layers, can compromise single-layer designs since that state can reveal access patterns and other security-sensitive information that a single-layer model simply cannot consider.

Trace-Oriented PD: Removing Layer Dependency. The above discussion leads to the following question: *Is it possible to achieve a stronger PD whose security is independent of individual technology stack layers?*

Layer-independence is preferable. First, it enables modularity and ensures across-the-layers security. Second, it enables the evaluation of different schemes based on overall security strength. Performance metrics (e.g. time/space efficiency) also make better sense when they are least interwoven with stack layers. Otherwise, it is difficult to compare PD solutions operating on two different layers. Third, the fewer dependencies on implementation specifics, the better the security abstraction. PD can now be compared with other security constructs such as ORAMs; Such a connection is hard to establish for layer-dependent PD.

We can then define trace-oriented plausible deniability by modifying the CPA-style security game of Sec. 2.3 in the following way: instead of device snapshots, the adversary will receive the operation traces

3 The double quotation marks are due to the fact that although most schemes use standard encryption, there are some schemes (e.g. [3]) using primitives such as secret-sharing instead.

4 The term 1/2 reflects the fact that the $\mathcal{A}$ can guess randomly and win the game with probability 1/2.

as the reply to its challenge requests in the security game (along with the device snapshots). Namely, it is stipulated that the adversary cannot tell which of the two challenge sequences were executed, even if it gets to learn the outputs of the PD logic (aka operation traces) before they are physically executed on the storage medium. Intuitively, this is a stronger requirement than that of standard PD because operation traces may contain more information than snapshots—it is totally possible that two different sequences of operation traces lead to the same snapshot.

Operation traces are comprised of `Read` and `Write` operations. As mentioned in Sec. 2.2, only `Write` operations leave traces on the storage medium. Thus, `Write` is the only type of operation that the trace-oriented definition needs to consider. Namely, it only requires that the `Write` traces reveal no information of the access requests to a PD scheme. Removing `Read` operations from traces is also preferable because an analogue of Lem. 2 will show that including `Read` will lead to a trace-oriented PD definition that is equivalent to ORAMs (instead of write-only ORAMs), thus suffering ORAMs' efficiency lower-bounds [5, 20, 22, 27, 47].

To achieve this we consider a function $\mathsf{WOnly}(\cdot)$ that filters out the `Read` operations but passes the `Write` operations; The above security game can then be modified to return to the adversary the result of applying $\mathsf{WOnly}(\cdot)$ on operation traces. This constitutes the final definition of trace-oriented PDs.

A PD scheme meeting the trace-oriented definition also satisfies the standard, CPA-style PD definition of Sec. 2.3. Indeed, `Write` traces (the output of WOnly) contain all the information to induce storage medium snapshots; If they are oblivious of the input access request, so are the snapshots. Furthermore, it resolves the issue of layer dependency: notice that lower-layer traces are always obtained from higher-layer traces, via an implementation-specific PPT procedure. Since indistinguishable operation traces remain indistinguishable after being processed by arbitrary PPT procedures, trace-oriented PD schemes allow the existence of extra layers between the PD logic and the physical devices.

Equivalence between Trace-Oriented PDs and Write-Only ORAMs. Blass et al. [4] constructed a trace-oriented PD scheme from wORAM. Further, in Appx. B we show that wORAM can also be constructed from trace-oriented PDs. This implies the following lemma:

Lemma 1. *Write-only ORAMs are both* sufficient and necessary *for trace-oriented PDs.*

3.2 Unified Definition

The CPA-Game for PD from [4, 9] defined in Sec. 2.3 is deeply integrated with the underlying application. New solutions have to repurpose this game to define PD at different system layer with specific underlying devices. Further, several constructions restricted the adversary's power in exchange for better efficiency, making it unclear how they fit into this game definition.

In this section we introduce a unified definition that (1) generalizes the CPA game in Sec. 2.3, thus inherits all its advantages, e.g. secure against CPA-style coercion attacks, applicable for both multi-snapshot and single-snapshot settings; (2) it encompasses existing constructions and admits comparisons among them (shown in Sec. 4); (3) it can be instantiated for both the traditional device-oriented security model and the trace-oriented one proposed in Sec. 3.1.

We present the definition for both device-oriented and trace-oriented settings, with multi-snapshot adversaries. We use the parameter $\mathcal{L}$ (e.g., $\mathcal{L}$ can be FS, BD, FTL) to restrict the game to the scenario where the adversary is attacking a storage device used at layer $\mathcal{L}$ (i.e., the device is directly connected to the layer $\mathcal{L}$).

The security game captures restrictions on the adversary's power through two parameters: (1) the number of rounds r (single-snapshot when $r = 1$, multi-snapshot when $r > 1$), and (2) a new parameter $\mathcal{R}$, which can be instantiated by the designer. This leads to a more unified point of view, as all PD schemes indeed share the same abstraction modulo the parameters $\mathcal{L}$ and $\mathcal{R}$. Further, comparisons of security strength among different schemes become possible by investigating the restrictiveness of their respective $\mathcal{R}$ parameters.

Layer-Specific Notations. An $\mathcal{L}$-request is a legitimate access (`Read` or `Write`) request to layer $\mathcal{L}$. An $\mathcal{L}$-pattern $\mathcal{P}$ is an ordered sequence of $\mathcal{L}$-requests. Let $\mathcal{P}_1 \cup \mathcal{P}_2$ denote the concatenation of requests in patterns $\mathcal{P}_1$ and $\mathcal{P}_2$. We define the function $\mathsf{OpTrace}(\mathcal{L}, \mathcal{P})$ that, for a layer $\mathcal{L}$ and access request $\mathcal{P}$, outputs a sequence of operations that are meant to be executed on the underlying physical device (i.e., the "*operation traces*").

Definition 1. *For a device $\mathcal{D}$ and layer $\mathcal{L}$, an* $\mathcal{L}$-layer PD *($\mathcal{L}$-PD) scheme Σ consists of the following two algorithms* $(\mathsf{Setup}, \mathsf{Oper})$*:*

- $\mathsf{Setup}(\lambda, \mathcal{D})$*: this function provides the initial setups. It takes as input the security parameter λ and the device $\mathcal{D}$. It outputs the tuple $(\mathcal{D}_{\mathsf{init}}, \mathsf{K}_{\mathsf{pub}}, \mathsf{K}_{\mathsf{hid}})$, where $\mathcal{D}_{\mathsf{init}}$ is the initialized device, $\mathsf{K}_{\mathsf{pub}}$ is the key used to protect*

Denote this game as $\mathsf{PD}^{\mathcal{L}}_{\Sigma,\mathcal{A}}(\lambda, r)$. It is parameterized by a security parameter λ, an $\mathcal{L}$-PD scheme $\Sigma = (\mathsf{Setup}, \mathsf{Oper})$, an adversary $\mathcal{A}$, and a number of rounds r.

Initialization: The challenger $\mathcal{C}$ executes the setup algorithm to get $(\mathcal{D}_{\mathsf{init}}, \mathsf{K}_{\mathsf{pub}}, \mathsf{K}_{\mathsf{hid}}) \leftarrow \mathsf{Setup}(\lambda, D)$. $\mathsf{K}_{\mathsf{pub}}$ is given to $\mathcal{A}$. The current state is set as $\mathcal{D}_{\mathsf{st}} := \mathcal{D}_{\mathsf{init}}$.

Challenge: $\mathcal{C}$ picks a random bit $b \xleftarrow{\$} \{0,1\}$ and then executes the following steps for r rounds with $\mathcal{A}$ ($i = [1..r]$):

1. The adversary $\mathcal{A}$ sends to $\mathcal{C}$ two $\mathcal{L}$-patterns:
$$\mathcal{P}_0 := \mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}^2_{\mathsf{pub}} \quad \text{and} \quad \mathcal{P}_1 := \mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}_{\mathsf{hid}},$$
where $(\mathcal{P}^1_{\mathsf{pub}}, \mathcal{P}^2_{\mathsf{pub}}, \mathcal{P}_{\mathsf{hid}})$ satisfy the following requirements:
   (a) $\mathcal{P}^1_{\mathsf{pub}}$ and $\mathcal{P}^2_{\mathsf{pub}}$ contain only public requests;
   (b) $\mathcal{P}_{\mathsf{hid}}$ contains only hidden requests;
   (c) $\mathcal{P}^2_{\mathsf{pub}}$ must be $\emptyset$ if $\mathcal{P}_{\mathsf{hid}}$ is $\emptyset$;
   (d) $\mathcal{P}^1_{\mathsf{pub}}$ and $\mathcal{P}^2_{\mathsf{pub}}$ additionally satisfy some *scheme-specific* requirements $\mathcal{R}^1$ and $\mathcal{R}^2$ respectively;
2. Based on the selected bit b, $\mathcal{C}$ executes the request pattern $\mathcal{P}_b$ on the device, in an order of its choice, and updates the current device state as:
$$\mathcal{D}_{\mathsf{st}} \leftarrow \mathsf{Oper}(\mathcal{D}_{\mathsf{st}}, \mathcal{P}_b, \mathsf{K}_{\mathsf{pub}}, \mathsf{K}_{\mathsf{hid}}).$$
3. $\mathcal{C}$ sends $\mathcal{D}_{\mathsf{st}}$ and/or $\mathsf{WOnly}(\mathsf{OpTrace}(\mathcal{L}, \mathcal{P}_{b_i}))$ to $\mathcal{A}$.

Output: Finally, $\mathcal{A}$ outputs a bit b^*. The game then terminates with the output defined as $\mathsf{PD}^{\mathcal{L}}_{\Sigma,\mathcal{A}}(\lambda, r) := (b == b^*)$.

Fig. 1. Security game for multi-snapshot, device and trace-oriented plausible deniable system. The game involves r rounds to model both single and multi-snapshot adversaries.

the public data and $\mathsf{K}_{\mathsf{hid}}$ *is the key used to protect the hidden data.*

– $\mathsf{Oper}(\mathcal{D}_{\mathsf{st}}, \mathcal{P}, \mathsf{K}_{\mathsf{pub}}, \mathsf{K}_{\mathsf{hid}})$*:* Oper *is a* stateful algorithm, *i.e., it may maintain* internal *state across consecutive invocations*[5]*. It takes as input the current state* $\mathcal{D}_{\mathsf{st}}$*, an* $\mathcal{L}$*-pattern* $\mathcal{P}$*, and the key-pair* $(\mathsf{K}_{\mathsf{pub}}, \mathsf{K}_{\mathsf{hid}})$*. If* $\mathcal{P}$ *is not a valid* $\mathcal{L}$*-pattern, the algorithm outputs* $\perp$ *and halts; Otherwise, it generates a new state* $\mathcal{D}'$ *accordingly, and updates the current state to* $\mathcal{D}_{\mathsf{st}} := \mathcal{D}'$*. It outputs the updated state* $\mathcal{D}_{\mathsf{st}}$*:*

$$\mathcal{D}_{\mathsf{st}} \leftarrow \mathsf{Oper}(\mathcal{D}_{\mathsf{st}}, \mathcal{P}, \mathsf{K}_{\mathsf{pub}}, \mathsf{K}_{\mathsf{hid}}).$$

Device and Trace-Oriented $\mathcal{L}$-Layer PD. The security for a PD scheme can now be formalized through the CPA-style game in Fig. 1. This game is played between a coercive adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ running a PD scheme Σ. $\mathcal{A}$ only knows $\mathsf{K}_{\mathsf{pub}}$ (for the public data that the scheme is not trying to hide), but not $\mathsf{K}_{\mathsf{hid}}$[6]. The game is played for r rounds: when $r = 1$ the game models a single-snapshot adversary, when $r = \mathsf{poly}(\lambda)$ it models a multi-snapshot adversary.

At each round $i = [1..r]$, $\mathcal{A}$ is allowed to send two patterns $\mathcal{P}_0$ and $\mathcal{P}_1$. $\mathcal{P}_0$ is the concatenation of two public parts $\mathcal{P}^1_{\mathsf{pub}}$ and $\mathcal{P}^2_{\mathsf{pub}}$,[7] while $\mathcal{P}_1$ is the concatenation of $\mathcal{P}^1_{\mathsf{pub}}$ and an arbitrary hidden request pattern $\mathcal{P}_{\mathsf{hid}}$ (up to some restrictions that will be discussed soon). The challenger executes $\mathcal{P}_b$ by picking public and hidden requests in an order of its choice. The challenger then sends back the snapshot of the device and/or the operation traces.

The adversary should not be able to tell which patterns are executed. More specifically, we define the advantage of the adversary in the game to be $Adv(\mathcal{A}) := |\Pr[\mathsf{PD}^{\mathcal{L}}_{\Sigma,\mathcal{A}}(\lambda, r) = 1] - 1/2|$. This captures the exact requirement of PD—the execution of hidden requests $\mathcal{P}_{\mathsf{hid}}$ now can be interpreted as some other public requests $\mathcal{P}^2_{\mathsf{pub}}$. Indeed, $\mathcal{A}$ cannot tell the difference by investigating the snapshots and/or operation traces.

As discussed in Sec. 3.1, the $\mathsf{WOnly}(\cdot)$ function needs to be applied to screen out the `Read` operations from the traces before they are sent to $\mathcal{A}$. Lem. 2 states that if a PD solution is secure in a setup where `Read` instructions do not leave traces, it can be converted to a secure write-only ORAM. However, if a PD solution is provably secure even if `Read` instructions *leave* traces on the storage device, then it can be converted to a full ORAM via an analog of Lem. 2. Thus, it will suffer from ORAMs' efficiency lower bound [5, 20, 22, 27, 47]. For example, HIVE [4] can be proven secure even if `Read` leaves traces; indeed, it employs this actively by explaining a hidden access as a public `Read`. Unsurprisingly, HIVE is constructed based on fully-secure ORAMs.

The Adversary Requests. Note the requirements put on the adversary's request patterns. First, $\mathcal{P}_0$ and $\mathcal{P}_1$ must share the same $\mathcal{P}^1_{\mathsf{pub}}$, as otherwise $\mathcal{A}$ (with $\mathsf{K}_{\mathsf{pub}}$) can always win the game by checking the public data in the received snapshot. For a similar reason, requirement (c) (Fig. 1) is also necessary.

An essential difference between this security game and previous ones lies in requirement (d) (Fig. 1). Ideally, a scheme should be secure against all PPT adversaries. However, this is usually not easy to achieve in practice. Instead, previous attempts proved the security of their solutions by making various additional

5 This internal state should not be confused with the device state $\mathcal{D}_{\mathsf{st}}$ in the input to Oper.

6 Otherwise, there is nothing to protect.

7 Note that in the CPA game in [9], $\mathcal{P}_0$ is also allowed to contain hidden requests. While this seems to make our definition weaker, we show in Appx. A that the CPA game defined in Fig. 1 is actually equivalent to that in [9] in this aspect.

assumptions on the adversary. In this paper we show that these assumptions can be viewed as requirements on the $\mathcal{P}^1_{\mathsf{pub}}$ and $\mathcal{P}^2_{\mathsf{pub}}$ part of the adversary's requests in the security game. Thus, the game in Fig. 1 generalizes them as two parameters $\mathcal{R}^1$ and $\mathcal{R}^2$. In Sec. 4 we show that by instantiating these two parameters properly, the game can capture the security requirements of all existing PD systems. Moreover, this approach provides a way to compare different PD solutions, where schemes with less restrictive $\mathcal{R}^1$ and $\mathcal{R}^2$ are preferable in terms of security.

We choose not to provide restrictions for $\mathcal{R}^1$ and $\mathcal{R}^2$. Such guidelines are not possible nor useful since $\mathcal{R}^1$ and $\mathcal{R}^2$ are solution-dependent.

Definition 2 (Device/Trace-Oriented $\mathcal{L}$-Layer PD)**.** *For a layer $\mathcal{L}$, a* PDS $\Sigma = (\mathsf{Setup}, \mathsf{Oper})$ *is device/trace-oriented $\mathcal{L}$-Layer* PD *if for any* PPT *adversary $\mathcal{A}$ in the game of Fig. 1, it holds that $Adv(\mathcal{A}) \leq \mathsf{negl}(\lambda)$, where $Adv(\mathcal{A}) := |\Pr[\mathsf{PD}^{\mathcal{L}}_{\Sigma,\mathcal{A}}(\lambda, r) = 1] - 1/2|$.*

4 Comparison

In this section we compare the security and performance of existing PD schemes. We leverage the unified definition in Sec. 3.2 to provide a framework for comparing the security of existing solutions. We further perform the comparison from a variety of aspects (summarized in Table 1), to provide a comprehensive understanding of these schemes.

Security Metrics. The security guarantees of PD schemes are related to the assumptions made on adversaries, which can be captured by the unified definition. Specifically, the snapshot frequency and the type of security (listed in Table 1) categorize the scheme in coarse granularity, and the constraints $\mathcal{R}_1$ and $\mathcal{R}_2$ are used to characterize the power of adversaries in a finer way. Before presenting the constraints on each scheme, let us interpret the meaning of these constraints:

- The ideal scheme should be secure against all PPT adversaries (corresponding to empty $\mathcal{R}_1$ and $\mathcal{R}_2$). No existing solutions achieve this level of security. $\mathcal{R}_1$ and $\mathcal{R}_2$ can be viewed as specifying a subset of all PPT adversaries against which a PD scheme is secure. Thus, they provide a criterion for security comparison: *the more constrictive, the fewer adversarial behaviors are ruled out, resulting in a more powerful adversary and a more secure scheme.*
- The constraints also define under which conditions hidden operations can be executed safely. For example, the $\mathcal{R}^1_{\mathsf{DEFTL}}$ and $\mathcal{R}^2_{\mathsf{DEFTL}}$ for DEFTL below essentially say that the hidden operations can be performed with *any* public operation, as long as an `Unmount` is performed (together with the trigger post-processing), before the device is handed over to the coercive adversary. The constraints for other schemes can also be interpreted similarly. Thus, *the less constrictive the constraints, the more flexibility a scheme has in performing hidden operations.*

The following interprets the security of existing schemes by specifying their corresponding constraints, and draw comparisons along the way.

4.1 Single vs. Multi-snapshot Adversary

To achieve single-snapshot security, existing solutions explore two major directions. The first direction is inspired by classical *steganography*, i.e., embedding relatively small messages within large cover-texts, such as adding imperceptible echoes at certain places in an audio recording [35]. Anderson et al. [3] explored steganographic file systems and proposed two approaches for hiding data. The first approach defines the target file as the (password-derived) linear combination of a set of cover files. The second approach encrypts the target file using a block cipher with password-derived secret keys, and then stores it at the location determined by a cryptographic hash of the filename. In both approaches, an adversary without the correct password can get no information about whether the protected file ever exists. The latter approach was later implemented and optimized by McDonald et al. [30] and Pang et al. [33]. Unfortunately, such approaches are not extremely space-effective, and come with potential data loss and high performance overheads. They are not suited for building modern systems handling large amounts of data at high speed.

The StegFS series [3, 30, 33] have the same model and share the same constraints in the unified definition:

- $\mathcal{R}^1_{\mathsf{StegFS}}$: no restrictions ($\mathcal{P}^1_{\mathsf{pub}}$ can be any pattern);
- $\mathcal{R}^2_{\mathsf{StegFS}}$: $\mathcal{P}^2_{\mathsf{pub}}$ must be an empty pattern.

The second direction [2, 24, 41] handles PD at block-device level by designing disk encryption tools that help users embed "hidden volumes" (together with "public volumes") within the device (e.g., in the free space regions), while preventing adversaries from learning how many such volumes the device actually contains. Dif-

Schemes	Year	Snapshot	Security Type	Layer	I/O Perf. (Pub/Hid)	Space Util.	Data Loss	No Add'l. Space	Invisible	Device
StegFS98 [3]	98	Sin	device	FS	-	≈15%	●	●	○	General
StegFS99 [30]	99	Sin	device	FS	0.86/0.06	-	●	●	◒	General
StegFS03 [33]	03	Sin	device	FS	0.06	>80%	●	●	○	General
TrueCrypt [2]	04	Sin	device	BD	-	100%	●	●	○	General
MobiFlage [41]	13	Sin	device	BD	0.95	100%	●	●	○	General
MobiPluto [12]	15	Sin	device	BD	-	100%	●	●	○	General
DEFTL [24]	17	Sin	device	FTL	-	100%	○	●	○	NAND flash
DEFY [34]	15	Mul	device	FS	-	100%	●	○	○	NAND flash
MobiCeal [13]	18	Mul	device	BD	0.78	-	●	●	○	General
INFUSE [15]	20	Mul	device	FS	0.94/0.03	>100%	●	●	◒	Certain NAND flash
PEARL [16]	21	Mul	device	FTL	0.6/0.15	80%	●	●	○	NAND flash
HIVE [4]	14	Mul	trace	BD	-	50%	○	●	○	General
HIVE-B [4]	14	Mul	trace	BD	0.004	50%	○	○	○	General
DataLair [9]	17	Mul	trace	BD	0.19/0.01	50%	○	○	○	General
ECD [51]	17	Mul	trace	FTL	*	52.5%	●	○	○	NAND flash
PD-DM [14]	19	Mul	trace	BD	0.10/0.07	≈50%	○	●	○	General

Table 1. Comparison of existing PD solutions. An empty circle signifies that the solution does not satisfy the property at the top, while a black circle denotes that the solution satisfies the property. A half-full circle in the Invisible column denotes that the respective solution (StegFS, INFUSE) tried to be invisible but did not completely succeed.

ferent keys are used to encrypt different volumes using randomized encryption indistinguishable from pseudo-random free space noise. Upon coercion, a user can provide the encryption keys for the public volumes, thus providing a plausible non-hidden use case for the disk. The adversary does not have any evidence for the existence of additional volumes.

TrueCrypt [2] successfully implemented this idea. It stores hidden volumes in the free space of public volumes. To hide their existence, TrueCrypt fills all free space with random data and encrypts the hidden data with a semantically secure encryption scheme that has pseudo-random ciphertexts. Upon coercion, the user can reveal the keys for the public volumes, and claim that the remaining space contains random free space. Rubberhose [23], MobiFlage [42] and DEFTL [24] are implementations following similar ideas targeted to different use cases (mobile devices, NAND flash).

In the unified security definition, the constraints of TrueCrypt and MobiFlage are the following:

– $\mathcal{R}^1_{\text{Tc\&Mf}}$: no restrictions;
– $\mathcal{R}^2_{\text{Tc\&Mf}}$: $\mathcal{P}^2_{\text{pub}}$ must be an empty pattern.

Further, DEFTL has the following constraints:

– $\mathcal{R}^1_{\text{DEFTL}}$: the last operation in $\mathcal{P}^1_{\text{pub}}$ must be `Unmount`;
– $\mathcal{R}^2_{\text{DEFTL}}$: $\mathcal{P}^2_{\text{pub}}$ must be an empty pattern.

Highlights: Single-snapshot security can be achieved with low overheads and high performance. Although these schemes are designed in different storage layers (FS vs BD), they share the same restrictions on the adversaries' choice of patterns, thus achieving identical security guarantee. However, all the aforementioned schemes fail to protect against multi-snapshot adversaries. For example, when TrueCrypt writes hidden data, the device "free space" changes unexplainably. When observed by a multi-snapshot adversary this cannot be plausibly explained away. After all, why did the disk free area change without corresponding substantial changes to the public data?

Thus, to protect against multi-snapshot adversaries, one needs to *hide not only the existence of hidden data, but also associated access patterns.*

Given this insight, progress in this area centers mainly around mechanisms that can consistently explain updates to both the public and hidden data across multiple snapshots. Currently, three major approaches exist: (i) using oblivious RAM mechanisms (Sec. 4.2), (ii) using canonical forms (Sec. 4.3) and (iii) relying on device/deployment-specific properties (Sec. 4.4). The

remainder of this section will explore these approaches, aiming to understand the fundamentals and distill insights to guide future designs.

4.2 ORAM-Based PD Schemes

Multi-snapshot secure PD requires mechanisms that hide users' access patterns to hidden data. ORAMs [20] are natural tools for this task.

ORAMs. Roughly, an ORAM ensures a database-hosting server cannot determine which database (the "RAM") entries are accessed by one of its client. Access patterns of any same-length access sequences are designed to be indistinguishable. As a simple example, a (highly inefficient yet secure) ORAM (with $O(n)$ asymptotic complexity per access) can be constructed by filling the database with randomized encrypted data; To access one of the elements, the client reads the entire database, re-encrypts it and writes it back to the server. More efficient solutions exist that enable complexities much lower than $O(n)$ [8, 10, 11, 44, 45]. An exhaustive treatment is out of scope here. The following summarizes how ORAMs were employed to obtain PD solutions.

HIVE [4] was the first to deploy ORAMs. It introduced *hidden volume encryption*. The main idea was similar to a previous work [2], i.e., to divide the storage into a public volume and a hidden volume[8], each volume being accessed using an ORAM mechanism. Additionally, for every access to a volume (either public or hidden), the system also executes *dummy* accesses to the other volume. Since ORAM accesses are indistinguishable from each other (whether dummy or not), adversaries cannot tell the difference between 1) accesses to the public volume and 2) accesses to the hidden volume, *which satisfies the exact requirement of the CPA game for PD.*

Moreover, HIVE leveraged the observation that `Read` operations are not visible to adversaries, since such operations do not leave any discernible traces (Sec. 2.2). Thus, it is sufficient to use write-only ORAM schemes (see Def. 3 in Appx. B) that only hide `Write` operations. HIVE [4] designed a specific write-only ORAM with a small stash of pending blocks in memory, i.e., where blocks are stored to be written later when a free block becomes available. The write-only ORAM stash can also behave as a queue for caching hidden data, and hidden volume accesses can be performed together with existing (if any) public volume accesses to minimize the need for additional dummy accesses. In this case, the adversary cannot tell the difference between 1) accesses to the public volume only and 2) accesses to both public and hidden volumes. The only associated requirement now becomes the need for enough plausible public accesses to pair with the hidden data in the stash when written to disk.

In the unified definition, HIVE has the following constraints:

- $\mathcal{R}^1_{\text{HIVE}}$: $\mathcal{P}^1_{\text{pub}}$ and $\mathcal{P}_{\text{hid}}$ must be of equal length;
- $\mathcal{R}^2_{\text{HIVE}}$: $\mathcal{P}^2_{\text{pub}}$ must be an empty pattern.

HIVE-B is another PD scheme proposed in the same paper as HIVE. It provides the same security guarantee as HIVE, but with different constraints:

- $\mathcal{R}^1_{\text{HIVEB}}$: no restrictions;
- $\mathcal{R}^2_{\text{HIVEB}}$: $\mathcal{P}^2_{\text{pub}}$ and $\mathcal{P}_{\text{hid}}$ must be of equal length (i.e. containing the same number of requests).

DataLair [9] extends these ideas and observes that operations on public data do not need to be hidden since they are anyway public. In fact, revealing operations on public data reinforces deniability as it shows plausible non-hidden device use. Therefore, DataLair only uses wORAMs for the hidden volumes, while allowing public data to be accessed (almost) directly without any oblivious access mechanism. Moreover, it designs a specific throughput-optimized wORAM. Following the strategy of HIVE, it pairs the operations on hidden data with those on public data, and ensures that such executions are indistinguishable from the operations on public data alone. Compared with its predecessors, DataLair accelerates public operations by two orders of magnitude, and also speeds up hidden operations.

In the unified definition, DataLair introduces a parameter ϕ in the constraints:

- $\mathcal{R}^1_{\text{DataLair}}$: $\mathcal{P}^1_{\text{pub}}$ should contain at least $\phi \times k$ public `Write` operations where k is the length of $\mathcal{P}_{\text{hid}}$ and ϕ is a pre-defined parameter;
- $\mathcal{R}^2_{\text{DataLair}}$: $\mathcal{P}^2_{\text{pub}}$ must be an empty pattern.

We note that DataLair's $\mathcal{R}^1$ constraints are stronger than, e.g., StegFS. However, this does not imply that StegFS provides stronger security because the security game models the adversary's power also through the r parameter: StegFS is designed for single-snapshot, while DataLair is designed for multi-snapshot adversaries.

MobiCeal [13] implements PD at the BD layer, and supports a broad deployment of any block-based file systems for mobile devices. MobiCeal improves perfor-

8 The original HIVE scheme supports multiple volumes. W.l.o.g., only the two-volume case is considered here (for simplicity).

mance by replacing wORAMs with *dummy* write operations coupled to public writes. In the unified definition, MobiCeal has the following constraints, where $f \in (0,1)$ is a random number and λ is a rate parameter:

– $\mathcal{R}^1_{\text{MobiCeal}}$: For each public write in $\mathcal{P}^1_{\text{pub}}$, also perform a dummy write with a certain probability. The dummy write contributes m dummy block writes, where m is chosen according to an exponential distribution, $m = \lfloor -(\ln(1-f))/\lambda \rfloor$.
– $\mathcal{R}^2_{\text{MobiCeal}}$: $\mathcal{P}^2_{\text{pub}}$ must be an empty pattern.

Where to Write. An important factor affecting both the security and efficiency of ORAM-based PD approaches [4, 9] is *free-block allocation* (FBA), i.e. the mechanism to keep track of free blocks to store new incoming data.

Note that HIVE uses separate ORAMs on public and hidden storage spaces, and DataLair uses ORAM only for hidden space. A naive approach would be to have separate FBA mechanisms for public and hidden spaces. Unfortunately, this can lead to storage capacity waste, as the hidden space must be allocated even if it is never used. Instead, a better solution uses a "global" FBA algorithm across all the storage space. In this case, both the public and the hidden volume can be of the same logical size as the underlying partition, and use all the available space for either hidden or public data.

However, this turns to be a delicate task due to the existence of hidden data. On the one hand, the FBA should avoid overwriting existing hidden data; On the other hand, such avoidance should be strategically hidden to not raise doubts from the adversary about the existence of hidden data.

Moreover, since it is in the data path, FBA must be efficient. Significant amount of work has been devoted [4, 9] to the design of FBA algorithms that meet the above criteria. The reader is referred to the original papers for details.

4.3 Replacing Randomization with Canonical Forms

ORAMs are used in PD designs because they can hide both the locations and contents of each access, mostly via inherently high-overhead randomization. Yet, randomization is not really necessary to achieve plausible deniability[14]. Simple canonical forms – e.g., such as used in log-structured file systems [18] always writing data sequentially, treating the logical address space as a circular buffer – may be enough to decouple the user's logical from physical access patterns.

Since canonical forms ensure pre-defined physical device write traces, an adversary is prevented from inferring the logical layer access patterns, of which the traces are independent of.

Further, importantly, an advantage of certain canonical forms (e.g., sequential) is the ability to retain data locality and thus result in significantly higher efficiency than randomization-based ORAM approaches.

PD-DM [14] is the first work that *explicitly* notes the above idea. Its design ensures that all writes to the physical device are located at sequentially increasing physical addresses, similar to `Append` operations in log-structure file systems. PD-DM stipulates that whenever a public data record D_{pub} is written on the device, an additional random string R (the "payload") is written immediately in the immediately adjacent next block. To store hidden data, PD-DM will first encrypt it (indistinguishably from random) and then write it as the payload of some public data write D_{pub}. In this case, device snapshots look like the following:

D_{pub}	R	D_{pub}	R	⋯	D_{pub}	R	⋯

Since the encrypted hidden data looks indistinguishable from the random "payload" of public writings, adversaries are unable to distinguish whether any hidden data exists or not.

In terms of the unified definition constraints, PD-DM has similar constraints to DataLair [9]. However, while DataLair requires a fixed-value parameter ϕ, in PD-DM the value of ϕ is system specific.

ECD [51] works in a related manner. The idea is to partition the device into a public and a hidden volume. The public volume is managed by the system in the standard way, independent of the hidden one. The hidden volume is divided into equal-size sequential *segments*, denoted as $\{s_1, s_2, \ldots, s_N\}$. Each s_i contains some free blocks containing random strings, while other blocks may be occupied by encrypted hidden data. ECD keeps moving data from s_{i-1} to s_i at a predetermined rate. During this procedure, any free blocks will be re-randomized, while existing hidden data blocks will be re-encrypted. To a polynomial adversary this looks like all of s_i is re-randomized. New hidden data is written encrypted into a free block during the migration from s_{i-1} to s_i. Overall this can be viewed as creating an artificial canonical form on the hidden volume, where segment s_i is periodically overwritten by its immediate predecessor s_{i-1}. In the unified model, ECD has the following constraints:

– $\mathcal{R}^1_{\text{ECD}}$: no restrictions;
– $\mathcal{R}^2_{\text{ECD}}$: $\mathcal{P}^2_{\text{pub}}$ must be an empty pattern.

ECD has the least restrictive rules compared with other schemes. This is because it employs a regular system behavior which periodically modifies the state of the device to cover up the hidden operation; Such a periodic system update is general enough to cover any type of hidden operations. In contrast, other schemes use public operations to cover up hidden operations, which naturally introduces constraints to ensure that the public and hidden operations are "paired" properly.

We conclude that HIVE, DataLair and PD-DM provide multi-snapshot security at the BD layer, with comparable constraints. They all achieve the stronger trace-oriented security.

4.4 Device-Specific Mechanisms

In pursuit of performance, a recent line of work emerged building plausible deniability guarantees on device-specific properties, e.g., electric charge levels in flash memories. This enjoys certain advantages over previous work: 1) it may avoid heavy machinery (e.g., ORAMs) and may lead to lightweight solutions; 2) resulting mechanisms may be closer or even native to the underlying device, allowing for higher performance and better plausibility.

This approach was often implicit in the literature. This section seeks to sublimate the essence of such constructions in a unified perspective. First, consider several existing constructions.

INFUSE [15] builds a PD scheme in the flash FTL layer. The main idea is to modulate additional information in charges and voltage levels of individual NAND cells, the minimal storage unit for NAND. A cell can hold one (SLC, single-level cell) or more (MLC, multiple-level cell) data bits. Bits are encoded and decoded by using a programmable *threshold voltage* V_{th} and a predefined *reference voltage* V_{r}. For example, an SLC cell with threshold voltage $V_{\mathsf{th}} = 3V$ will be interpreted as a logical "1" when the reference voltage level is $V_{\mathsf{r}} = 3.5V$, and as a "0" if either (i) the reference voltage level drops below $V_{\mathsf{r}} = 2.5V$ or (ii) the threshold voltage is increased to e.g., $V_{\mathsf{th}} = 4V$. MLC work similarly, with multiple levels to encode multiple values.

Some recent flash controllers are able to operate the same cell in both SLC and MLC mode [28]. This provides an opportunity to hide bits. Multiple bits can be stored in a particular cell using an "MLC-style" encoding but on inspection the system can claim that the cell is in SLC mode and provide only a single bit. Care needs to be taken to ensure device-wide indistinguishability between sets of cells in either SLC or MLC mode. This constitutes the core idea of INFUSE.

Under the (mostly empirical) assumption that an adversary cannot distinguish which cells are used in which mode or whether there are any inconsistencies in the distribution of SLC vs MLC cells, this scheme provides significant speedups. Public data operations are orders of magnitude faster than existing multi-snapshot resilient PD systems, and only 15% slower than a standard non-PD baseline and hidden data operations perform comparably to the-state-of-the-art PD systems.

In the unified definition, INFUSE has the following constraints:

– $\mathcal{R}^1_{\text{INFUSE}}$: the last operation in $\mathcal{P}^1_{\mathsf{pub}}$ must be `Unmount`;
– $\mathcal{R}^2_{\text{INFUSE}}$: $\mathcal{P}^2_{\mathsf{pub}}$ must be an empty pattern.

PEARL [16] is also operating in the FTL layer, but relies on a new smart *write-once memory* (WOM) encoding that does not require custom voltage programming.

Unfortunately, once written to, a NAND flash cell cannot be reprogrammed before an `Erase` of its containing block. Further, NAND flash is reliable only for a limited number of `Erase` cycles. This can severely limit device lifespan. Complex wear leveling algorithms are deployed to "even out" wear and maximize lifespan.

WOM codes [38] have been proposed to further optimize this wear. They use an important property of NAND flash: previously-unwritten-to cells can be written to even if they are in pages that have been written to before. WOM codes encode with enough redundancy (e.g. using 3 cells to store 2 bits) to allow *multiple* writes to the same page (i.e., with different data each time) without requiring an `Erase`.

At a high level, in the subsequent (e.g., second) `Write`, the idea is to modify only the bits that have not been written-to in the first `Write`. A well-designed encoding allows the second logical `Write` to be encoded in the resulting physical state with no ambiguity.

For example, consider the case of an encoding with 2-bit logical data records encoded onto 3 physical bits. For each 2-bit logical record $s \in \{0,1\}^2$ the encoding defines two possible physical 3-bit configurations $\mathsf{E}_1(s)$ and $\mathsf{E}_2(s)$. When logical record s is stored *for the first time*, $\mathsf{E}_1(s)$ is stored physically. If the logical record s needs to be replaced with a new value s' (at the same location) writing simply converts the physical value $\mathsf{E}_1(s)$ to $\mathsf{E}_2(s')$. The WOM encoding is designed unambiguously and in such a way that any such conversion does not require overwriting an existing written-to cell. Since such

a code allows two Write operations per erasure, it is called a 2-write WOM code[9].

PEARL [16] hides information by *modulating the written public data according to the data to be hidden*. To this end, it re-purposes WOM codes. When public data is written, the codeword is chosen based on the bits of the data that need to be hidden. This enables PEARL to surreptitiously hide information even in the presence of a powerful multi-snapshot adversary. The end-result is device state that is indistinguishable from the case of a device that was simply writing data multiple times using a WOM code. Much care needs to be taken in the design of the specific WOM code to not introduce device-wide bias. Overall however, the fact that WOM codes are widely deployed on NAND flash further strengthens plausibility. Most importantly, the resulting performance is comparable to the non-PD baseline on real-world workloads!

In the unified definition, PEARL has the following constraints:

- $\mathcal{R}^1_{\text{PEARL}}$: the last operation in $\mathcal{P}^1_{\text{pub}}$ must be Unmount;
- $\mathcal{R}^2_{\text{PEARL}}$: $\mathcal{P}^2_{\text{pub}}$ needs to generate k 1st invalid pages where k is the length of $\mathcal{P}_{\text{hid}}$.

A pattern that can generate a 1st invalid page can be: 1) one public Write followed by a public Delete to the same page; 2) one public Write which has a correspondingly public Write in $\mathcal{P}^1_{\text{pub}}$.

DEFY [34] is a log-structured FS for NAND flash that offers PD with a newly proposed secure deletion technology. It is based on WhisperYAFFS [48], a log structured FS which provides full disk encryption for flash. Log-structured FSes have two relevant properties:

1. Data (e.g., files, directories, links) and metadata are stored sequentially within the logical address space, and any access to data (including Read) can cause the update of its corresponding metadata;
2. Updates/deletes of data and metadata will not cause an actual deletion. Instead, a new address will be assigned to the updated version, and the old data/metadata is just marked as old; Subsequent garbage collection handles it.

DEFY achieves PD by exploiting the above properties in the following way. In DEFY, modifications to hidden data will cause the allocation of new records. Such allocations can be claimed as the results of metadata updating due to public Read/Write, since such updating can also lead to the assignment of new records. Once these records (for hidden data) become obsolete (i.e. succeeded by newly allocated records), the system can claim that they were due to public accesses and are now securely deleted. Due to the irreversibility of secure deletion, the adversary has no choice but to believe that these records were due to public accesses. The system thus denies the existence of hidden data successfully.

DEFY enjoys impressive efficiency. Read operation can be as fast as the Linux EXT4 file system.

Further, in the unified definition, DEFY offers PD conditioned on constraints:

- $\mathcal{R}^1_{\text{DEFY}}$: $\mathcal{P}^1_{\text{pub}}$ should contain some public operations, and the last operation in $\mathcal{P}^1_{\text{pub}}$ must be Unmount;
- $\mathcal{R}^2_{\text{DEFY}}$: $\mathcal{P}^2_{\text{pub}}$ should contain public accesses that generate enough deleted pages to cover accesses in $\mathcal{P}_{\text{hid}}$.

Unfortunately DEFY is not secure, and it can be compromised in a few attempts to exhaust the writing capacity [24]. Also, as hidden data are stored masqueraded as securely deleted obsolete (public) data, to maintain plausibility, the space occupied by them must be plausibly and frequently enough overwritten by public data. This results in data loss.

In addition, DEFY assumes the existence of a special, "tag storage area" on the device that is hidden from the adversary. This breaks security against multi-snapshot adversaries. Thus, the security provided by DEFY is considered weaker than the schemes that do not assume the existence of such a hidden area.

In summary, these schemes are optimized for specific deployment cases. INFUSE and PEARL exploit voltage variation and properties of WOM codes respectively, to encode hidden data together with public data at the same locations. DEFY plausibly encodes hidden data as securely-deleted obsolete public data. The resulting solutions are specific to the underlying devices but achieve performance comparable to the non-PD baseline on real-world workloads. Such efficiency is clearly out of the reach of ORAM-based solutions. Importantly, WOM code-based schemes provide an unusually favorable combination of strong security and high performance.

4.5 Access Pattern Hiding Techniques

As mentioned earlier, a key point of PD schemes is to conceal access patterns to hidden data. In order to hide the existence of hidden data, a PD scheme should prevent adversaries from learning not only *which hidden*

9 W.l.o.g., for simplicity this work focuses on 2-write WOM codes. There exist k-write WOM codes that admit k writings per erasure [19, 40, 50].

access happens, but also *how many hidden accesses happen*. This is in contrast to ORAMs where only that the access patterns of logical requests are not revealed, while the number of accesses can be public.

To hide *which hidden access happens*, existing PD schemes leverage one of the following two strategies: 1) randomizing the write trace on physical devices; 2) enforcing the write trace to follow certain canonical form (the commonly used one is log-structure). HIVE, DataLair and MobiCeal follow the first strategy; PD-DM, ECD, DEFY, INFUSE and PEARL follow the second strategy. The last 3 schemes are designed for NAND flash devices, where the device is written sequentially by default. To hide the *number of hidden accesses*, the first strategy is to make the change of device state due to a hidden access to be indistinguishable from that of some non-hidden accesses. Thus, any changes on the devices can be attributed to certain public accesses, and the number of hidden accesses can be claimed as 0. Note that the public accesses used to "explain" hidden accesses do not need to happen in reality. The PD schemes that use this strategy are HIVE, DEFY, PEARL and ECD. HIVE explains a hidden access as a public `Read`. DEFY and PEARL explains it either as a public `Write` or `Delete`, while ECD explains a hidden access as a system behavior that happens at a pre-defined rate.

Another strategy to hide the number of hidden accesses is to "pair" hidden accesses with some public accesses and ensure that the write trace of the public accesses alone is indistinguishable from that of both the public accesses and hidden accesses. As a result, for an adversary, only public accesses happen. Examples include HIVE-B, DataLair, PD-DM, Mobiceal and INFUSE.

4.6 Performance Metrics

Table 1 also looks at existing solutions from a performance standpoint.

I/O Performance. PD comes with both throughput and space overheads. Some schemes report performance separately for public operations and hidden operations – shown in Table 1 in the form of "x/y". It means that the public throughput is x times of the non-PD baseline, and the hidden throughput is y times of the baseline. Some other schemes reported only one overall performance number, and some schemes did not provide any explicit performance number or even have not been evaluated at all since they are designed in theory and no implementation is completed (shown as "-" in Table 1). ECD is a special case whose performance (marked with "*" in the table) depends on a system parameter. Recall that ECD covers up hidden operations by periodically updating the device state at a prefixed rate r. Thus, the number of hidden operations that the system is able to perform is determined by the updating rate r, rather than the public operations.

Space Utilization. The column "**Space Util.**" shows how efficiently the storage capacity of physical devices can be exploited by each PD scheme. It is computed as the ratio *between* the max size of data (both public and hidden) that can be stored in one device *and* the total capacity of the storage device as a metric for space utilization in Table 1 (space required by meta-data is excluded for simplicity). Note that INFUSE enjoys a space utilization larger than 100%. That is because INFUSE encodes hidden bits at physical storage cells that already contain some public data (see Sec. 4.4).

Additional Safe Space, Data Loss. As discussed earlier, some PD solutions assume the existence of an area on the devices that remains hidden from adversaries (e.g., the TSA block in DEFY, or the stash in HIVE). Some schemes suffer from data loss, i.e., the hidden data may be overwritten (maybe by public data) in some use cases. Table 1 also lists these caveats for each scheme.

5 Key Insights

PD solutions deployed in a layer do not necessarily ensure PD for the entire system (Sec. 3.1). However, we have shown that trace-oriented PD implies the standard PD security, and trace-oriented PD secure mechanisms can provide PD for the entire system. More specifically, since traces at a layer are converted through a PPT algorithm into traces at at lower layer (Sec. 3.1), indistinguishability of traces at a layer implies indistinguishability of traces and snapshots at any lower-layers. This addresses the issue of the SSD/FTL example in Sec. 2.3: if the PD solution is BD layer trace-oriented secure, it achieves plausible deniability even though the SSD has an FTL layer below.

A key point of multi-snapshot resilient PD systems lies in hiding access patterns to hidden data (Sec. 4.1). ORAMs have been used to build PD schemes that hide access patterns (Sec. 4.2). However, ORAM-based solutions are inefficient due to the inherently heavy randomization machinery. Further, ORAMs require carefully-designed free-block allocation algorithms.

To improve performance, existing work has explored two directions, canonical form and device-specific solutions. Canonical form-based PD solutions can hide user access patterns, and significantly increase throughput (Sec. 4.3). In particular, sequential approaches can preserve data locality and make good use of locality-optimized systems deploying caching and read-ahead mechanisms. Further, lightweight, device-specific PD solutions have been developed, that exploit specific devices and deployment settings to achieve efficiency comparable to the non-PD baselines, that does not always come at the expense of strong security (Sec. 4.4).

We also note that several PD solutions impose $\mathcal{P}^2_{\text{pub}}$ to be empty. This is because to conceal hidden accesses, some PD schemes make the device state associated with a hidden access be indistinguishable from that of some non-hidden accesses.

6 Future Directions

We leverage these insights to propose several promising directions for future work.

6.1 Trace-Oriented Security

Most existing PD solutions exploit the possibility that different traces may result in the same snapshot, which allows users to interpret hidden operations as public ones. However, such an advantage is lost in the trace-oriented setting, as the adversary obtains *actual* traces. Thus, it is necessary to remove any clues of the user's operations from the traces. Lem. 1 establishes the equivalence of trace-oriented PD to wORAMs. Thus, this goal is hard to achieve without relying on wORAMs.

Established lower-bounds for ORAMs can be viewed as a signal of the inefficiency of wORAMs, which translates into clues to the inefficiency of robust PD. More specifically, strongly-secure PD solutions feature inherent fundamental efficiency limits, and achieving efficient PD requires layer-dependency.

However, future progress on wORAM lower-bounds will also apply to trace-oriented PDs. Importantly, there is no *established* lower-bound for wORAMs yet. Thus, the optimistic interpretation of Lem. 1 encourages us to seek efficient trace-oriented PDs. Given that all exiting trace-oriented PD solutions are built on top of wORAMs, it will be interesting to have constructions that do not make *explicit* use of wORAMs. Such constructions may circumvent the ORAM lower-bound (if it turns out it applies to wORAMs). Thus, while it is challenging to build schemes achieving both trace-based security and good efficiency, this also yields the following insight: Instead of striving for trace-based PD, a more promising direction may be to take the approach illustrated in Sec. 4.4, and design PDs directly for the "right" layer.

More specifically, a careful selection of the layer at which the PD solution is implemented, if secure for traces from lower layers, may provide both trace-oriented security and efficiency. For example, for an HDD device whose block device manager does not shuffle the FS layout, an efficient FS-layer PD solution may be a better option than a trace-oriented PD solution. For an SSD device, or an HDD with a block device manager that shuffles the FS layout, an efficient PD solution may be implemented at the BD or FTL layers.

6.2 Invisible PD

Typically, PD systems only intend to hide the existence of hidden data, not the fact that *the system in use is PD*. However, the deployment of a PD system already raises suspicion about the existence of sensitive data. A similar issue also exists for deniable encryption [6, 7].

To equip the user with more credibility in the face of coercive authorities, future work may focus on *invisible* PD schemes that hide not only contents but also the evidence that the system is being used to hide data. This can be done by, e.g., making the scheme look indistinguishable to a off-the-shelf storage system. For instance, as shown in the "invisible" column of Table 1, StegFS [30] was designed to be indistinguishable to EXT2. However, StegFS [30] needs to also maintain a bitmap. Future efforts may look into making this scheme fully indistinguishable by removing the bitmap, while not compromising security.

INFUSE [15] was designed to be indistinguishable from YAFFS [1]. However, INFUSE has a limited capacity for hidden data: If too much hidden data is stored, the distribution of cell voltages may become suspicious. Further, INFUSE requires the firmware support which allows precise manipulation on flash cell voltages. However, current NAND flash chips do not have the corresponding interface admitting such manipulations.

A promising direction is work on WOM codes [16] (Sec. 4.4), where information is surreptitiously hidden in the WOM codes of public data. While WOM codes are widely deployed on NAND flash, making it possible to deny the use of a PD solution, PEARL is based

on customized "PD-friendly" WOM codes. Nevertheless, PEARL [16] suggests that WOM codes have great potential for efficient PD constructions. Future research may focus on finding other PD-friendly WOM codes with improved efficiency, and further our understanding of PD-friendly WOM codes, e.g., proving necessary and efficient conditions for WOM codes to be PD-suitable, and lower-bounds on code rates for such codes.

6.3 Explore Adversary Model Changes

As discussed in Sec. 2, designing PD schemes secure against multi-snapshot adversaries is challenging. Existing solutions are still too slow. To design a new PD scheme against multi-snapshot adversaries, one can either come up with a new strategy to hide the number of hidden accesses, or a new strategy to hide which hidden access happens, and then combine it with some of the exiting strategies listed in Sec. 4.5. Finding new strategies for hiding the number of hidden accesses seems more promising as there could be different ways to interpret the disk changes resulting from hidden accesses.

A further promising direction is to design solutions secure against more realistic, bounded adversaries. Examples worth exploring include (lower) bounds on the number of operations that the user needs to perform between adversary-captured snapshots, or the total number of snapshots that an adversary can capture.

We also note however that assumptions A2 and A3 (Sec. 2.3) underestimate the power of realistic adversaries, who can perform attacks that include cold boot attacks, access swap files and core dumps. Real-time access to, e.g., caches, allows inference of some `Read` operations. Unfortunately, existing work ignores caches. Extending deniability to other parts of the system stack represents an interesting future direction. For instance, future work may treat caches and the DRAM as another layer in the storage hierarchy. We note however that a PD solution that is provably secure when `Read` instructions leave traces on the storage device, can be converted to a full ORAM via an analog of Lem. 2, thus will suffer from ORAMs' efficiency lower bound [5, 20, 22, 27, 47]. (Sec. 3.2)

6.4 Synthetic Operations

Existing PD schemes try to match hidden operations to public ones. This makes hidden operations rather passive: to perform a hidden operation, the system has to wait until the occurrence of the related public operation. It also restricts the types of allowed hidden operations.

Instead, an *active* approach is to let the system generate synthetic public operations whenever the user wants to perform hidden operations. Existing AI/ML solutions, e.g., variational autoencoders [26] and generative adversarial networks [21], trained on large sets of real-user operations, may be used to generate synthetic public operations that are difficult to distinguish from real public operations.

7 Conclusion

Plausible deniability can provide strong privacy guarantees that impacts millions of users in a world increasingly encroaching on encryption and personal privacy. Yet, building secure plausibly deniable efficient systems is far from trivial. This work systematizes existing knowledge for researchers and practitioners alike aiming to understand, deploy, or design plausible deniability systems. We believe plausible deniability to be an important property on the cusp of efficient mainstream practicality. This work is meant as a concise yet reasonably-complete guide on this journey.

8 Acknowledgements

We thank the shepherd, Diogo Barradas, and the anonymous reviewers for their feedback. This work has been supported by the National Science Foundation (award 2052951) and the Office of Naval Research (award N000142112407).

References

[1] A robust flash file system since 2002. "https://yaffs.net/".

[2] *TrueCrypt*. "http://truecrypt.sourceforge.net/".

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

[4] 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. ACM, 2014.

[5] Elette Boyle and Moni Naor. Is there an oblivious ram lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 357–

368, 2016.

[6] Ran Canetti, Sunoo Park, and Oxana Poburinnaya. Fully deniable interactive encryption. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part I*, volume 12170 of *Lecture Notes in Computer Science*, pages 807–835. Springer, 2020.

[7] Rein Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In *Advances in Cryptology – CRYPTO'97*, pages 90–104. Springer, 1997.

[8] Anrin Chakraborti, Adam J. Aviv, Seung Geol Choi, Travis Mayberry, Daniel S. Roche, and Radu Sion. roram: Efficient range ORAM with o(log2 N) locality. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[9] Anrin Chakraborti, Chen Chen, and Radu Sion. Datalair: Efficient block storage with plausible deniability against multi-snapshot adversaries. *Proceedings on Privacy Enhancing Technologies*, 2017(3):179–197, 2017.

[10] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client ORAM. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[11] Anrin Chakraborti and Radu Sion. Sqoram: Read-optimized sequential write-only oblivious RAM. *Proc. Priv. Enhancing Technol.*, 2020(1):216–234, 2020.

[12] Bing Chang, Zhan Wang, Bo Chen, and Fengwei Zhang. Mobipluto: File system friendly deniable storage for mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, page 381–390, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Bing Chang, Fengwei Zhang, Bo Chen, Yingjiu Li, Wen-Tao Zhu, Yangguang Tian, Zhan Wang, and Albert Ching. MobiCeal: Towards secure and practical plausibly deniable encryption on mobile devices. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 454–465. IEEE, 2018.

[14] Chen Chen, Anrin Chakraborti, and Radu Sion. Pd-dm: An efficient locality-preserving block device mapper with plausible deniability. *Proceedings on Privacy Enhancing Technologies*, 2019(1), 2019.

[15] Chen Chen, Anrin Chakraborti, and Radu Sion. Infuse: Invisible plausibly-deniable file system for nand flash. *Proceedings on Privacy Enhancing Technologies*, 4:239–254, 2020.

[16] Chen Chen, Anrin Chakraborti, and Radu Sion. PEARL: Plausibly deniable flash translation layer using WOM coding. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association.

[17] Alexei Czeskis, David J. St. Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce Schneier. Defeating encrypted and deniable file systems: Truecrypt v5.1a and the case of the tattling os and applications. In *Proceedings of the 3rd Conference on Hot Topics in Security*, HOTSEC'08, pages 7:1–7:7, Berkeley, CA, USA, 2008. USENIX Association.

[18] Fred Douglis and John Ousterhout. Log-structured file systems. In *COMPCON Spring'89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers.*, pages 124–129. IEEE, 1989.

[19] Philippe Godlewski. Wom-codes construits à partir des codes de hamming. *Discrete mathematics*, 65(3):237–243, 1987.

[20] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[21] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[22] Pavel Hubáček, Michal Kouckỳ, Karel Král, and Veronika Slívová. Stronger lower bounds for online oram. In *Theory of Cryptography Conference*, pages 264–284. Springer, 2019.

[23] R. P. Weinmann J. Assange and S. Dreyfus. Rubberhose:cryptographically deniable transparent disk encryption system. "http://marutukku.org".

[24] 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. ACM, 2017.

[25] Gabriela Kennedy. Encryption policies: Codemakers, codebreakers and rulemakers: Dilemmas in current encryption policies. *Computer Law & Security Review*, 16(4):240–247, 2000.

[26] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[27] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *Annual International Cryptology Conference*, pages 523–542. Springer, 2018.

[28] Sungjin Lee, Keonsoo Ha, Kangwon Zhang, Jihong Kim, and Junghwan Kim. Flexfs: A flexible flash file system for mlc nand flash memory. In *USENIX Annual Technical Conference*, pages 1–14, 2009.

[29] 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.

[30] Andrew D McDonald and Markus G Kuhn. StegFS: A steganographic file system for Linux. In *Information Hiding*, pages 463–477. Springer, 1999.

[31] J. Mull. How a syrian refugee risked his life to bear witness to atrocities. toronto Star Online, posted 14-March-2012, 2012.

[32] Adam O'Neill, Chris Peikert, and Brent Waters. Bi-deniable public-key encryption. In *Annual Cryptology Conference*, pages 525–542. Springer, 2011.

[33] HweeHwa Pang, Kian-Lee Tan, and Xuan Zhou. Stegfs: A steganographic file system. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 657–667. IEEE, 2003.

[34] Timothy Peters, Mark Gondree, and Zachary N. J. Peterson. DEFY: A deniable, encrypted file system for log-structured storage. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[35] Fabien AP Petitcolas, Ross J Anderson, and Markus G Kuhn. Information hiding-a survey. *Proceedings of the IEEE*, 87(7):1062–1078, 1999.

[36] Denver Post. Password case reframes fifth amendment rights in context of digital world. "http://www.denverpost.com/news/ci_19669803".

[37] The Register. Youth jailed for not handing over encryption password. 2010.

[38] Ronald L Rivest and Adi Shamir. How to reuse a "write-once memory". *Information and control*, 55(1-3):1–19, 1982.

[39] 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.

[40] Amir Shpilka. New constructions of wom codes using the wozencraft ensemble. *IEEE Transactions on Information Theory*, 59(7):4520–4529, 2013.

[41] Adam Skillen and Mohammad Mannan. Mobiflage: Deniable storage encryptionfor mobile devices. *IEEE Transactions on Dependable and Secure Computing*, 11(3):224–237, 2013.

[42] Adam Skillen and Mohammad Mannan. On implementing deniable storage encryption for mobile devices. 2013.

[43] Toronto Star. How a syrian refugee risked his life to bear witness to atrocities. 2012.

[44] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652*, 2011.

[45] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[46] M Weaver. Developer tortured by raiders with crowbars. 31 October 97.

[47] Mor Weiss and Daniel Wichs. Is there an oblivious ram lower bound for online reads? In *Theory of Cryptography Conference*, pages 603–635. Springer, 2018.

[48] WhisperSystems. Github: Whispersystems/whisperyaffs: Wiki, 2012. "https://github.com/WhisperSystems/WhisperYAFFS/wiki".

[49] Wikipedia. Key disclosure law. "http://en.wikipedia.org/wiki/Key_disclosure_law".

[50] Eitan Yaakobi, Scott Kayser, Paul H Siegel, Alexander Vardy, and Jack Keil Wolf. Codes for write-once memories. *IEEE Transactions on Information Theory*, 58(9):5985–5999, 2012.

[51] 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.

A Unified PD Definition Equivalence

We now show that the unified PD definition in Sec. 3.2 is equivalent to the one in [9], which allows *both* $\mathcal{P}_0$ and $\mathcal{P}_1$ to contain hidden requests.

First, it is easy to see that the definition in [9] is no weaker than the one defined in Sec. 3.2, because allowing hidden requests in *both* $\mathcal{P}_0$ and $\mathcal{P}_1$ only grants the adversary more power in the CAP game. So, the only thing left is to show that the definition in Sec. 3.2 is no weaker than that in in [9]. Roughly speaking, this is true for the following reason. Consider a pair of [9]-type challenge $\mathcal{P}_1 := \mathcal{P}^1_{\text{pub}} \cup \mathcal{P}_{\text{hid}}$ and $\mathcal{P}'_1 := \mathcal{P}^1_{\text{pub}} \cup \mathcal{P}'_{\text{hid}}$, both of which contain hidden requests (but share the same public part). The security guaranteed by Fig. 1 says that a $\mathcal{P}_0 := \mathcal{P}^1_{\text{pub}} \cup \mathcal{P}^2_{\text{pub}}$ should be indistinguishable with $\mathcal{P}_1$, and also with $\mathcal{P}'_1$. Thus, it must the case that $\mathcal{P}_1$ and $\mathcal{P}'_1$ are indistinguishable. In the following, we formalize the above intuition.

To prove it formally, we need to show that if a PPT adversary $\mathcal{A}$ can win the CPA game defined in [9] with non-negligible probability, then it can be efficiently converted into another PPT $\mathcal{A}'$ that wins the CPA game defined in Fig. 1 with non-negligible probability. We construct $\mathcal{A}'$ as follows. $\mathcal{A}'$ begins by picking a random bit $b' \xleftarrow{\$} \{0,1\}$ and then runs $\mathcal{A}$ internally. In the i-th ($i \in \{1,\ldots,r\}$) round, $\mathcal{A}$ will send a pair of challenge requests $\mathcal{P}_0$ and $\mathcal{P}_1$ (we emphasize that both $\mathcal{P}_0$ and $\mathcal{P}_1$ contain hidden requests). When this happens, $\mathcal{A}'$ sets $\mathcal{P}'_1 := \mathcal{P}_{b'}$; and $\mathcal{A}'$ sets $\mathcal{P}'_0$ to the public part of $\mathcal{P}_0$ (or equivalently, the public part of $\mathcal{P}_1$). $\mathcal{A}'$ uses $\mathcal{P}'_0$ and $\mathcal{P}'_1$ as its i^{th}-round challenge requests for its own CPA game (i.e., the game defined Fig. 1), and forwards the response from its challenger to the internal $\mathcal{A}$. At the end, if $\mathcal{A}$ guesses $\mathcal{A}'$'s b' correctly, $\mathcal{A}'$ will output 1; otherwise, $\mathcal{A}'$ outputs 0.

It is easy to see that if the b picked by $\mathcal{A}'$'s challenger (in the game specified in Fig. 1) equals 1, then the view of the internal $\mathcal{A}$ is identical to the case when it is participating in the CPA game in [9]. Since $\Pr[b = 1] = 1/2$, it follows that with probability $1/2$, the internal $\mathcal{A}$ will "think" that it is participating in the CPA game from [9]. Recall that we assume that $\mathcal{A}$ wins the [9] CPA game with some non-negligible probability p. Therefore, $\mathcal{A}'$ will win its own Fig. 1 game with probability $p/2$, which is also non-negligible.

B Write-Only ORAMs from Trace-Oriented PDs

B.1 Write-Only ORAMs

Notations. A *data request* is a tuple $(\mathsf{op}, \mathsf{addr}, \mathsf{d})$, where $\mathsf{op} \in \{\mathtt{Read}, \mathtt{Write}\}$ denotes a $\mathtt{Read}(\mathsf{addr})$ or a $\mathtt{Write}(\mathsf{addr}, \mathsf{d})$ operation, addr denotes the identifier of the block being read or written, and d denotes the data being written. For an ORAM scheme Π and a sequence $\vec{y} = \{r_1, \ldots, r_n\}$ of data requests, let $\mathsf{PhysicalAcc}^{\Pi}(\vec{y})$ denote the the physical access pattern that is produced by executing Π on $\vec{y}$.

Definition 3 (Write-Only ORAMs [4, 29, 39]). *An ORAM scheme is* write-only oblivious *if for any two sequences of data requests $\vec{y}_0$ and $\vec{y}_1$ containing the same number of* `Write` *requests, it holds that*

$$\mathsf{WOnly}(\mathsf{PhysicalAcc}^{\Pi}(\vec{y}_0)) \overset{c}{\approx} \mathsf{WOnly}(\mathsf{PhysicalAcc}^{\Pi}(\vec{y}_1)),$$

where $\mathsf{WOnly}(\cdot)$ *filters out the read physical accesses, and* $\overset{c}{\approx}$ *denotes* computational indistinguishability.

Remark 1. *In Def. 3 $\vec{y}_0$ and $\vec{y}_0$ may have different length*[10]*; they are only required to contain the same number of* `Write` *requests. This stipulates that the execution of* `Read` *requests does not incur any physical writes: otherwise two sequences with different number of* `Read` *requests might be easily distinguished by checking the number of resulted physical writes.*

B.2 Write-Only ORAMs from Trace-Oriented PD

The High-Level Idea. In the security game of trace-oriented PDs, it is guaranteed that the writing traces resulted from two adversarially chosen access patterns $\mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}_{\mathsf{hid}}$ and $\mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}^2_{\mathsf{pub}}$ are computationally indistinguishable. In particular, this implies the existence of two "universal" public patterns $\mathcal{P}^1_{\mathsf{pub}}$ and $\mathcal{P}^2_{\mathsf{pub}}$ with the following property: for any hidden patterns $\mathcal{P}_{\mathsf{hid}}$, the `Write` traces resulted from $\mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}_{\mathsf{hid}}$ are indistinguishable with that from $\mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}^2_{\mathsf{pub}}$ Given a PD scheme under the above restriction, a wORAM can be implemented as follows: to perform a target operation $\alpha = (\mathsf{op}, \mathsf{addr}, \mathsf{d})$, it first loads α into $\mathcal{P}_{\mathsf{hid}}$, and then executes the PDS access algorithm on $\mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}_{\mathsf{hid}}$, where $\mathcal{P}^1_{\mathsf{pub}}$ is the aforementioned universal public pattern. Thanks to the security of the PDS, the `Write` traces of the execution of $\mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}_{\mathsf{hid}}$ are indistinguishable from those of the execution of $\mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}^2_{\mathsf{pub}}$, whichever α is hidden inside $\mathcal{P}_{\mathsf{hid}}$. This provides the hiding of `Write` operations as required by wORAMs. This idea is formalized in Alg. 1.

Alg. 1: Write-Only ORAM from Trace-Oriented PDS

```
1:  procedure ORAM.Setup(1^λ)
2:      K_pub, K_hid, T_init ← PDS.Setup(1^λ)
3:      Initialize the device/memory blocks by executing T_init
4:  end procedure

5:  procedure ORAM.Access(op, addr, d)
6:      P_hid := (r_1, ..., r_n) ← HiddenGen(n, op, addr, d)
7:      if op == Read then ▷ if this is a Read request
8:          P := {R_dummy} ∪ P_hid
9:      else                ▷ if this is a Write request
10:         P := P^1_pub ∪ P_hid
11:     end if
12:     T ← PDS.Oper(K_pub, K_hid, P)
13:     return T
14: end procedure

15: procedure HiddenGen(n, op, addr, d)
16:     r_1 = (op, addr, d)
17:     for i = 1 to n do
18:         r_i = R_dummy   ▷ Pad the access pattern with dummy requests
19:     end for
20:     return (r_1, ..., r_n)
21: end procedure
```

ORAM Setup. The setup procedure (Line 1) simply runs the PDS.Setup to get the keys for public and hidden PD requests, and a sequence of commands $\mathcal{T}_{\mathsf{init}}$ that is meant to initialize the PD scheme. Once the commands in $\mathcal{T}_{\mathsf{init}}$ are executed on the underlying device/memory blocks, the ORAM system is ready to work.

ORAM Access. On input a request $\alpha = (\mathsf{op}, \mathsf{addr}, \mathsf{d})$, the ORAM.Access procedure first invokes a sub-procedure called HiddenGen (Line 15), which pads α with $n-1$ (same) dummy request R_{dummy}. This "padding" is necessary for the following reasons. Recall that the construction wishes to execute α by loading it in the hidden part of some input pattern to PDS.Oper. To leverage the se-

10 This is in contrast to standard ORAMs, which considers $\vec{y}_0$ and $\vec{y}_1$ of equal length, and requires the indistinguishabilty between the execution results without applying $\mathsf{WOnly}(\cdot)$.

curity of PDS, the hidden part must have length n. This is exactly the purpose of HiddenGen. Now, the procedure can create a pattern $\mathcal{P} = \mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}_{\mathsf{hid}}$ by concatenating the "universal" $\mathcal{P}^1_{\mathsf{pub}}$ with output $\mathcal{P}_{\mathsf{hid}}$ of HiddenGen; the writing traces for $\mathcal{P}$ will be indistinguishable with that for $\mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}^2_{\mathsf{pub}}$, due to the security of PDS.

As mentioned in Rem. 1, write-only ORAMs inherently require that Read request should not lead to physical writings. However, this condition may not be satisfied by the underlying PDS. To see that, consider a Read request α. Following the above strategy, the procedure will load α into $\mathcal{P}_{\mathsf{hid}}$ and set $\mathcal{P} = \mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}_{\mathsf{hid}}$. Since $\mathcal{P}_{\mathsf{hid}}$ contains only α and some dummy requests, we can assume that $\mathcal{P}_{\mathsf{hid}}$ does not cause any physical writings. However, the $\mathcal{P}^1_{\mathsf{pub}}$ part may contain some requests which incur physical writes. To resolve this issue, replace $\mathcal{P}^1_{\mathsf{pub}}$ with (the sequence of) a single dummy operation (Line 8), if α is a Read request. Since a dummy operation does not cause any writes, α can be executed without incurring physical writes.

Lemma 2. *If* $\mathsf{PDS} = (\mathsf{Setup}, \mathsf{Op})$ *is a secure PD scheme, then Alg. 1 is a secure write-only ORAM.*

Proof. Let $\vec{y}_0$ and $\vec{y}_1$ be two arbitrary data request sequences that contain the same number of Write operations. Note that it is possible that $|\vec{y}_0| \neq |\vec{y}_1|$. For $b \in \{0,1\}$, let Out_b denote the sequence of traces resulted from executing Alg. 1 sequentially on each requests in $\vec{y}_b$. The following shows that $\mathsf{WOnly}(\mathsf{Out}_0)$ and $\mathsf{WOnly}(\mathsf{Out}_1)$ are computationally indistinguishable.

Let m denote the number of write operations in $\vec{y}_0$ (or $\vec{y}_1$). Note that Out_0 and Out_1 may have different length, because the length of Out_b depends on $\vec{y}_b$. But it is clear that $|\mathsf{WOnly}(\mathsf{Out}_0)| = |\mathsf{WOnly}(\mathsf{Out}_1)| = m$ due to the following two facts:

1. by construction (specifically, Line 7 and Line 8), Read requests do not cause any writing traces;
2. both $\vec{y}_0$ and $\vec{y}_1$ contain exactly m Write requests.

Moreover, by the security of PDS, running Alg. 1 on any Write request has the same effect of executing $\mathsf{PDS.Oper}(\mathsf{K}_{\mathsf{pub}}, \mathsf{K}_{\mathsf{hid}}, \mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}^2_{\mathsf{pub}})$. Therefore, it follows that for any $b \in \{0,1\}$,

$$\underbrace{\big(\mathsf{WOnly}(\mathcal{T}^*), \ldots, \mathsf{WOnly}(\mathcal{T}^*)\big)}_{\text{repeat } m \text{ times}} \overset{\mathrm{c}}{\approx} \mathsf{WOnly}(\mathsf{Out}_b), \qquad (1)$$

where $\mathcal{T}^*$ denotes the output of the following operation:

$$\mathsf{PDS.Oper}(\mathsf{K}_{\mathsf{pub}}, \mathsf{K}_{\mathsf{hid}}, \mathcal{P}^1_{\mathsf{pub}} \cup \mathcal{P}^2_{\mathsf{pub}}).$$

It then follows immediately from Equation (1) that $\mathsf{WOnly}(\mathsf{Out}_0) \overset{\mathrm{c}}{\approx} \mathsf{WOnly}(\mathsf{Out}_1)$. □