Enabling Plausible Deniability in Flash-based Storage through Data Permutation

Weidong Zhu*, Wenxuan Bao†, Vincent Bindschaedler†, Sara Rampazzi†, and Kevin R. B. Butler†
*Florida International University, †University of Florida
*weizhu@fiu.edu,†{wenxuanbao, vbindschaedler, srampazzi, butler}@ufl.edu

***Abstract*—Plausible deniability (PD) allows at-risk users to deny the existence of their sensitive data stored on storage devices. This is critical to protect the privacy and the personal safety of users, as adversaries might force users to decrypt their devices, risking the disclosure of sensitive data that could endanger their lives and liberty.**

In this work, we show how current PD systems built on flash memory fail to obscure distinguishable data layouts created when hidden data is written. This deficiency makes them vulnerable to coercive adversaries who can capture single or multiple data snapshots of storage devices for scrutiny. To defend against this threat, we propose MUTE, a perMUTation-based PD systEm designed for flash memory. Building upon widely-adopted full disk encryption (FDE) mechanisms that provide device-level data encryption, MUTE modifies the distribution of initialization vectors (IVs) for encryption blocks within FDE, translating the hidden data into a permutation derived from the IV. Unlike other PD solutions, MUTE allows for storing hidden data without requiring the reduction of storage capacity. Moreover, it preserves the plausible deniability of the hidden data in a provably secure manner by maintaining the original logic of data operations on the flash memory without changing the data layout. We implement MUTE in the flash translation layer (FTL) of flash-based SSDs using FEMU, a widely-used emulator supporting flash memory research. Our evaluation with various micro-benchmarks and real-world workloads demonstrates that MUTE provides practical write and read throughputs of 23.4 MB/s and 15.7 MB/s and a capacity of 25.3 GB for hidden data in a 512 GB SSD, comparable with existing PD systems. MUTE achieves strong PD guarantees for flash-based devices against coercive adversaries, outperforming current PD systems.

***Index Terms*—Plausible Deniability, Data Layout, Flash-based SSDs**

1. Introduction

With escalating threats from authoritarian regimes and pervasive stringent law enforcement [9], [15], protecting sensitive information for vulnerable individuals and at-risk users, such as human rights activists and journalists, is critical for their safety and privacy. Traditional full disk encryption (FDE) [18] is widely used to protect sensitive data by making it inaccessible without the correct encryption key. However, adversaries may coerce users to surrender their keys, bypassing encryption protections. Consider real-world scenarios where individuals face extreme risks to protect their data. For example, secondary screenings of journalists crossing borders threaten press freedom, as agents conduct warrantless searches of devices like laptops and phones [33]. A videographer in Syria concealed a micro-SD card, which contains evidence of human rights violations, within a wound, smuggling it past hostile forces [12]. In these examples, discovering hidden data could result in life-threatening risks to users.

Plausible deniability (PD) offers a critical protection layer by allowing users to deny the existence of any sensitive data on their storage devices against coercive adversaries. Consider the examples above: the storage devices, rather than being smuggled at great risk to the carrier, could instead be presented to an authority without revealing sensitive data. This is especially crucial to protect high-value targets, such as journalists and whistleblowers. A PD system is typically employed as a standard feature within the FDE module [7], [29] to conceal the "pattern of accesses" [25] for hidden data. It allows users under coercion to reveal a decoy key that decrypts only non-sensitive (public) data while keeping the key for hidden data secret without raising suspicion.

Implementing PD in flash memory, a widely-used storage media in solid-state drives (SSDs) and mobile devices, presents significant challenges. Traditional PD systems [25], [10], [59], [26], [28], [68] are primarily designed for hard disk drives (HDDs), which support secure data deletion and in-place data updates. However, flash memory uses a flash translation layer (FTL) that remaps data to different physical locations, leaving remnants of deleted data and creating distinctive data patterns, which can inadvertently reveal the presence of hidden data [44].

In this paper, we introduce a novel data layout (DL) attack that allows adversaries to identify abnormal data layouts on flash memory. We propose MUTE, a per**MUT**ation-based PD syst**E**m in flash-based storage, to address the DL attack and provide robust PD guarantees, overcoming the following limitations presented in current flash-based PD systems [67], [44], [29], [45], [58]:
***(1)* Suspicious Data Layout Alterations.** The FTL assigns physical pages to incoming data in a predictable manner [47]. Thus, the data layout on flash memory exhibits observable patterns when users store a file on the device. However, current PD systems either modify the public data content [29] or manipulate the data allocation [67], [44], [45] for writing hidden data. This disrupts the normal data layout,

creating a distinctive marker of hidden data's presence.
(2) **Vulnerable to Multi-Snapshot Attacks.** Current PD systems are vulnerable to adversaries capable of obtaining multiple data snapshots over time [25], [26]. A realistic example is when users travel with their devices. Individuals may lose direct control of their devices in various situations, such as during airport security inspection or when devices are left unattended in hotel rooms, making them vulnerable to "hotel maid" attacks [29], [25]. Thus, adversaries can detect hidden information by analyzing changes in random data across multiple snapshots. Unfortunately, current flash-based PD systems either only protect against single-snapshot adversaries [67], [44] or remain vulnerable to the DL attack [26], [29], [58], [45].
(3) **Potential Hidden Data Loss.** Current PD systems require explicit storage space for hidden data, reducing the storage efficiency for public data; for example, PEARL [29] reserves 20% of a device's capacity for hidden data, and MDEFTL [45] stores the hidden data directly in flash memory. However, since the storage is unaware of the hidden data when users operate public data, writing public data can potentially overwrite hidden data, resulting in its loss.

MUTE outperforms current PD systems in the following aspects. *(1)* MUTE provides a provably secure PD protection for hidden data against multi-snapshot and DL attacks, whereas current systems cannot protect PD under these threats. *(2)* MUTE modifies the FDE logic without changing other FTL operations, reducing the system complexity without creating detectable patterns, such as DL attacks, that can compromise PD guarantees. *(3)* MUTE does not consume the storage capacity for hidden data, preventing it from being overwritten when users operate the public data.

The core idea of MUTE arises from the observation that modern implementations of FDE on flash-based storage predominantly leverage XTS-AES [42] for encryption. XTS-AES operates at the granularity of data units (flash pages), where each data unit can be divided into multiple encryption blocks (either 128 bits or 256 bits). Each encryption block within a data unit is assigned a unique integer as an Initialization Vector (IV) for encryption. Thus, the arrangement of IVs results in a permutation of multiple integers. We design MUTE to leverage permutation unranking and ranking operations for data hiding. Permutation unranking converts an integer into a permutation of elements within a set, whereas ranking transforms a permutation back into an integer value. Consequently, MUTE encodes hidden data as an integer to represent the ranking value, which is mapped to the permutation of IVs via ranking and unranking operations. Then, MUTE encrypts public data using the permuted IVs, embedding the hidden data within the standard encryption process without altering the public data's content or storage patterns. Retrieving the hidden data involves reversing this process by extracting the IV's permutation from the encrypted data. We thus make the following contributions:
• We derive a data layout (DL) attack that can compromise state-of-the-art PD systems designed for flash memory.
• We propose MUTE, a novel PD system that offers robust PD guarantees against multi-snapshot and DL attacks without occupying space for public data.
• We generalize plausible deniability with data layout as a security concept for PD systems by framing it within an indistinguishability game. Using this framework, we prove the security of existing PD systems and MUTE.
• We implement MUTE in FEMU [53], a widely used SSD emulator. Our evaluation shows that MUTE achieves 23.4 MB/s write and 15.7 MB/s read bandwidths for hidden data, with a 25.3 GB hidden capacity, comparable to current multi-snapshot PD systems [29], [45].

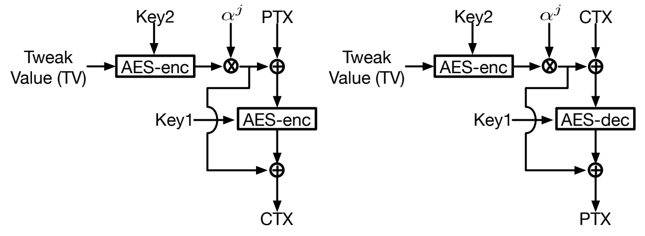
2. Background

2.1. Flash Memory

NAND flash has been widely used in solid-state drives (SSDs) due to its high performance and power efficiency [64]. Flash cells are grouped into pages (e.g., 4KB), which serve as the I/O access granularity. Each flash page is associated with an out-of-bound (OOB) area, which is typically 10% [55] of a flash page and used for metadata storage. Pages are grouped into a flash block containing 32 to 256 pages. Flash memory erases data at block granularity, but block erasure is time-consuming and can degrade the lifetime of the storage medium. Thus, it employs out-of-place writes, retaining old data until it is garbage collected.
Parallelism. Flash-based SSDs contain high parallelism for simultaneous data processing. Incoming I/O requests are distributed to different bus channels (CH). Each channel connects to multiple flash chips, known as logical units (LUNs), which can operate independently. Each flash chip consists of multiple dies, further divided into planes (PLs), each with its own register for plane-level parallelism.
Flash Translation Layer (FTL). FTL provides numerous functions to manage data accesses. *Address translation* converts logical page addresses (LPAs), derived from logical block addresses (LBAs) at the OS, into physical page addresses (PPAs) in flash memory. *Garbage collection* (GC) is designed to reclaim invalidated pages by migrating valid pages in a flash block and erasing the block containing invalidated pages. Since flash memory has a limited number of program/erase cycles [51], FTL also implements *wear-leveling* to distribute erasures evenly across blocks and manages worn-out blocks through *bad block management*.

2.2. Full-disk Encryption

Full-disk encryption (FDE) is crucial to protect the confidentiality of data at rest. IEEE Standard P1619 [42] defines the implementation of the XTS-AES encryption for block-oriented storage devices, which is recommended by NIST [34] and used by modern SSDs to implement FDE [60], [78]. Figure 1 shows the workflow of XTS-AES. To use XTS-AES, data is first divided into fixed-size units corresponding to flash pages. Each data unit is then split into encryption blocks of 128 or 256 bits. A randomly generated tweak value (TV) is assigned to each data unit and shared



(a) Encryption. (b) Decryption.

Figure 1: The workflow of XTS-AES. PTX indicates the plaintext, while CTX is the ciphertext.

among its encryption blocks to prevent duplicate encrypted units. To ensure the uniqueness of encrypted blocks, each encryption block is assigned with a unique j to generate α^j, where α is a primitive element of a finite field $GF(2^{128})$ or $GF(2^{256})$.[1] XTS-AES multiplies the encrypted TV with α^j, and this value is used in XOR operations with the plaintext and ciphertext during encryption and decryption. Since j is incrementally generated for each encryption block to ensure uniqueness, *we define it as the IV of each encryption block*. We later discuss in Section 5 how we leverage the permutation of IVs to store hidden data.

2.3. Data Hiding Techniques

Data hiding techniques have been employed in encryption software, such as TrueCrypt [6] and VeraCrypt [7], to prevent the disclosure of sensitive data on a storage device. A data-hiding scheme creates two encrypted volumes in the storage device. The public volume stores data that users feel free to surrender to adversaries, whereas the hidden volume contains the sensitive data that users want to protect. In these schemes, the storage space is filled with random data, and then hidden data is written such that it also appears as random data without being disclosed. However, flash memory changes assumptions about data hiding. Since flash memory writes data in an out-place fashion, “deleted” hidden data (random) persists in storage. Therefore, these changes result in observable patterns of random data that cannot be plausibly explained [44].

3. Model

3.1. System Model

We assume that users hold a flash-based storage device (e.g., SSD) to store *hidden* or *public* data. Hidden data requires the highest-level protection against adversaries who might use coercion to access sensitive information. In contrast, public data is surrendered without repercussion and without disclosing the hidden data. Both types of data should be encrypted using cryptographically-secure implementations [44], [29]. When coercion happens to the user, they surrender the public data key while denying the existence of the hidden data and keeping the hidden key confidential.

The storage device provides public and hidden modes as in a typical PD system. Users must use a correct password, either a public or hidden password, to initiate the respective PD mode. The public and hidden encryption keys are derived from the corresponding public and private passwords upon user authentication using the securely-created password at boot time. We assume the user can operate the storage device securely within a *session* in which the user is authenticated at the beginning, consistent with prior PD works [25], [67], [44], [29], [45]. Upon the completion of the data operation (public or hidden), the user can inform the storage device to end the session. Finally, we consider a common assumption that the storage device is equipped with a secure random number generator [71], [76], [44], such as using hardware-based randomness sources [70], providing sufficient randomness without being compromised.

3.2. Adversarial Model

We assume a *multi-snapshot* adversary that has multiple opportunities to image the storage device during which data writes can happen between snapshots [29], [25]. Then, we make these additional assumptions:

- We assume a computationally-bounded adversary capable of coercing the user to surrender the storage device and its encryption key. However, the adversary ceases coercion once convinced that no hidden data exists.
- We consider a threat model similar to other works on PD storage [67], [44], [29], [45], [25], [26], [28]. Specifically, we assume a common scenario where a user employs a PD system integrated into the computer system, akin to standard commercial solutions like NTFS Alternate Data Streams employed in Windows OSes [11]. Adversaries can be aware of the PD system's presence but cannot infer the keys/passwords for either the hidden or public modes. Note that the mere presence of a PD system does not serve as a red flag, as it cannot prove the existence of hidden data unless adversaries can find the *patterns of accesses* [25] of hidden data made by users. This is a common assumption in PD research, as PD systems can be explained as standard features in computer systems. Legal risks arise only when the existence of hidden data is confirmed; merely possessing a data-hiding tool does not lead to legal repercussions [13], [14], [16]. Thus, the employment of MUTE does not itself imply the use of hidden data storage.
- Before exiting the secure *session*, users perform a final operation – such as a graceful power-down with secure memory erasure [50] – to ensure that no data remnants remain in volatile memory. Therefore, adversaries cannot access the user's device, including the DRAM, storage device, processor, and OS in this session. This is a common model [29], [44], [67], as users are unlikely to operate hidden data in the presence of adversaries. Monitoring online I/O traffic makes it impossible to provide PD guarantees.

1. The power is the size of the encryption block which is 128 or 256 bits.

• Adversaries can extract the raw data from flash chips[2] [41], [35], [40], [80], including both data page and OOB area. This is practical during coercive storage inspection and consistent with existing PD works [29], [30], [44], [45]. Specifically, adversaries can access raw data by unsoldering flash memory chips and connecting them to a specialized hardware board [75], or even without physically removing the chips at all [61]. Furthermore, modern data forensic tools now support extracting data from more advanced flash chips, such as 3D NAND flash memory [21].

• We assume that adversaries can detect hidden data in a PD system by identifying anomalous data layouts, as insecure PD systems may create suspicious layouts when managing sensitive (hidden) data, differing from those generated in public mode. Thus, data layout patterns can reveal the hidden data existence, compromising PD guarantees.

3.3. Generalizing Plausible Deniability

We generalize the storage device and plausible deniability to verify a PD system's security in a provably secure way. The storage space is divided into public $\mathcal{V}_p$ and hidden $\mathcal{V}_h$ volumes, which are assigned with different passwords $\mathcal{P}_p$ and $\mathcal{P}_h$. Despite the well-known security vulnerabilities of password use, we assume the password can be selected securely for simplicity. Then, the password can be used to derive an encryption key that contains at least s bits of entropy [25]. We define s as a security parameter [48].

A volume consists of logical data blocks lb_i, where each logical block is associated with a physical flash page pg_i, which has the address $addr_i$. Therefore, the data layout of $\mathcal{N}$ physical flash pages $\cup_{i=1}^{\mathcal{N}}\{pg_i\}$ can be represented by the collection of physical page addresses $\cup_{i=1}^{\mathcal{N}}\{addr_i\}$. In addition, the storage device contains k files $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$, where each file f_i is associated with j data blocks $\mathcal{B}_i^j = \{j \text{ logical data blocks in file } f_i\}$.

We create a PD game [25] to formalize the security of the PD guarantee. We define an adversary $\mathcal{A}$ who can ask a challenger $\mathcal{C}$ to execute a sequence of write accesses $\mathcal{O} =< o_1, o_2, \ldots, o_n >$, where each access o_i writes data on either $\mathcal{V}_p$ or $\mathcal{V}_h$. Then, the PD game runs as follows:

1) The challenger $\mathcal{C}$ initializes the public $\mathcal{V}_p$ and hidden $\mathcal{V}_h$ volumes using passwords $\mathcal{P}_p$ and $\mathcal{P}_h$ with the security parameter s, respectively. Moreover, the challenger $\mathcal{C}$ generates a random bit $b \in \{0, 1\}$.
2) The challenger $\mathcal{C}$ provides the public password $\mathcal{P}_p$ to adversary $\mathcal{A}$ without revealing the hidden password $\mathcal{P}_h$.
3) The adversary $\mathcal{A}$ selects two write access sets: $\mathcal{O}_{0,1}$, which includes only writes to the public volume $\mathcal{V}_p$, and $\mathcal{O}_{1,1}$, which contains writes to both volumes $\mathcal{V}_p$ and $\mathcal{V}_h$. Finally, $\mathcal{A}$ sends $\mathcal{O}_{0,1}$ and $\mathcal{O}_{1,1}$ to $\mathcal{C}$.
4) The challenger $\mathcal{C}$ executes $\mathcal{O}_{b,1}$ on $\mathcal{V}_p$ and $\mathcal{V}_h$.
5) The challenger $\mathcal{C}$ returns the storage snapshot, including files $\mathcal{F}$, raw data $\cup_{i=1}^{\mathcal{N}_{b,1}}\{pg_i\}$, and data layout $\cup_{i=1}^{\mathcal{N}_{b,1}}\{addr_i\}$, to the adversary $\mathcal{A}$.
6) Repeat steps 3 to 5 for r rounds (for r at most polynomial in s).
7) The adversary outputs bit b'.

Definition 1. *We define a PD scheme under the restrictions to be secure when a negligible function* negl(s) *in security parameter s exists for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$, such that:*

$$Pr(b' = b) \leq \frac{1}{2} + \text{negl(s)}$$

Restrictions. The adversary $\mathcal{A}$ can only snapshot the storage once users unmount the device after a *session* as discussed in Section 3. While the game rounds are probabilistic polynomial times, it is impractical for users to surrender their devices exponentially many times. Moreover, since $\mathcal{A}$ has password $\mathcal{P}_p$ for public volume $\mathcal{V}_p$, $\mathcal{A}$ can easily identify the hidden volume $\mathcal{V}_h$ if $\mathcal{O}_{0,1}$ and $\mathcal{O}_{1,1}$ contain different amount of writes to $\mathcal{V}_p$. Thus, we restrict that, for any $\mathcal{O}_{0,1}$ and $\mathcal{O}_{1,1}$ chosen by $\mathcal{A}$, the data written to $\mathcal{V}_p$ must be identical.

PD Description. The definition indicates that a secure PD system renders the hidden and public accesses *indistinguishable*. When $b = 0$, all accesses in $\mathcal{O}_{0,1}$ are posed on public volume $\mathcal{V}_p$. When $b = 1$, a secure PD system ensures that any accesses in $\mathcal{O}_{1,1}$ containing writes to the hidden volume $\mathcal{V}_h$ can be plausibly explained by the accesses in $\mathcal{O}_{0,1}$. Thus, $\mathcal{A}$ cannot distinguish between the writes in $\mathcal{O}_{0,1}$ and $\mathcal{O}_{1,1}$.

The access patterns $\mathcal{O}_{0,1}$ and $\mathcal{O}_{1,1}$ impact the storage in terms of data content and layout. When $b = 0$, $\mathcal{O}_{0,1}$ writes public data into $\mathcal{N}_{0,1}$ flash pages $\cup_{i=1}^{\mathcal{N}_{0,1}}\{pg_{0,i}\}$ in $\mathcal{V}_p$ with data layout $\cup_{i=1}^{\mathcal{N}_{0,1}}\{addr_{0,i}\}$, where $pg_{0,i}$ is the data page at address $addr_{0,i}$. When $b = 1$, $\mathcal{O}_{1,1}$ operates on both $\mathcal{V}_p$ and $\mathcal{V}_h$. It writes $\mathcal{N}_{1,1}$ flash pages, denoted as $\cup_{i=1}^{\mathcal{N}_{1,1}}\{pg_{1,i}\}$ with data layout $\cup_{i=1}^{\mathcal{N}_{1,1}}\{addr_{1,i}\}$, where $pg_{1,i}$ is a flash page at address $addr_{1,i}$. Thus, a secure PD system ensures the indistinguishability between $\mathcal{O}_{0,1}$ and $\mathcal{O}_{1,1}$ if, for any access pattern $\mathcal{O}_{1,1}$, there exists $\mathcal{O}_{0,1}$ capable of producing identical sets of $\cup_{i=1}^{\mathcal{N}_{1,1}}\{pg_{1,i}\}$ and $\cup_{i=1}^{\mathcal{N}_{1,1}}\{addr_{1,i}\}$.

We adopt a *proof by counterexample* approach [73] in our PD game to evaluate the PD guarantees provided by existing PD works when using secure cryptography with security parameter s. Definition 1 indicates that the access patterns in $\mathcal{O}_{1,1}$ can always be plausibly explained by accesses in $\mathcal{O}_{0,1}$. If $b = 1$, the adversary $\mathcal{A}$ can be plausibly convinced that $b' = 0$ and $b' \neq b$. This implies, for polynomial rounds of game, $Pr(b' = b) \approx Pr(b' \neq b) \approx \frac{1}{2}$. However, if the adversary $\mathcal{A}$ can choose access patterns $\mathcal{O}_{0,1}$ and $\mathcal{O}_{1,1}$ such that $\mathcal{O}_{1,1}$ cannot be plausibly explained by $\mathcal{O}_{0,1}$, $\mathcal{A}$ can infer that $b' = b = 1$, thereby increasing $Pr(b' = b)$. Thus, $Pr(b' = b) > \frac{1}{2} + \text{negl(s)}$, violating Definition 1, and the PD guarantee is broken.

4. Data Layout Attack

Flash memory uses data allocation algorithms to assign incoming data to physical flash pages, making the data

2. We primarily consider NAND flash in our paper, as it is widely used in storage. NOR flash is less used in data storage due to its lower density. However, our method can also be applied to NOR flash because it has the similar FTL method as NAND flash-based SSD [23].

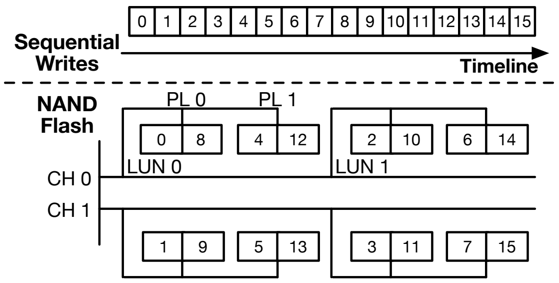


Figure 2: The data layout of data writes on flash memory. Data are written to flash memory consecutively in the timeline. They are distributed with the channel-first algorithm.

layout on flash memory predictable. We use the channel-first algorithm as an exemplar due to its wide adoption in current SSDs [47], although other algorithms can be applied. In Figure 2, the channel-first algorithm writes data by sequentially accessing channels to allocate flash pages – for instance, page 0 and page 1 possess different CH IDs – while other parameters such as LUN and PL remain the same. After a channel traversal finishes, it traverses LUNs within a channel for subsequent page allocations. Finally, it starts the PL traversal.

Since adversaries can physically access the storage device, they can identify the layout of user data and metadata on flash memory, including the address translation mapping tables. This can be accomplished by performing off-the-shelf analysis on flash chips [80], [41], [35], [40], [75] or by using advanced tools to access the controller and read the raw data [21]. As users surrender their devices with only public data keys under coercion, adversaries can access users' files in the OS and identify their data layouts on flash memory.

4.1. DL Attack

We propose and demonstrate a data layout (DL) attack to identify the abnormal data distributions on flash memory that can reveal the existence of hidden data. Three DL abnormalities can be exploited to break the current flash-based PD systems [44], [58], [67], [29], [45], [30]. We provide the detailed PD proof of current PD schemes in Section A.

***(1)* Adjacent Layout Exploitation.** The FTL assigns physical pages consecutively for incoming I/O requests. Assume a single-snapshot adversary, when $b = 0$, $\mathcal{O}_{0,1}$ operates on $\mathcal{V}_p$, writing $\mathcal{N}_{0,1}$ flash pages $\cup_{i=1}^{\mathcal{N}_{0,1}}\{pg_{0,i}\}$ with data layout $\cup_{i=1}^{\mathcal{N}_{0,1}}\{addr_{0,i}\}$, which has an adjacent pattern. When $b = 1$, $\mathcal{O}_{1,1}$ involves both $\mathcal{V}_p$ and $\mathcal{V}_h$, writing $\mathcal{N}_{1,1}$ flash pages to the storage. However, adversary $\mathcal{A}$ may intentionally select an access pattern $\mathcal{O}_{1,1} =< \mathcal{O}_p, \mathcal{O}_h, \mathcal{O}'_p >$, where hidden volume accesses occurs between two sets of public volume accesses. This creates a non-adjacent layout: $\cup_{i=1}^{j-1}\{addr_{1,i}\} \cup \cup_{i=j}^{\mathcal{N}_{1,1}}\{addr_{1,i}\}$, in which $addr_{1,j-1}$ and $addr_{1,j}$ are not consecutive. This data layout cannot be plausibly explained by $\mathcal{O}_{0,1}$. Thus, adversary $\mathcal{A}$ can identify the undecryptable flash pages located among the public data, violating the expected continuous public data layout.

Current flash-based single-snapshot PD systems [44], [67] are vulnerable to this issue. They adopt the traditional data allocation method of assigning flash pages consecutively for both public and hidden data. $\mathcal{O}_{1,1}$ can produce a distinguishable[3] layout that cannot be generated by $\mathcal{O}_{0,1}$.

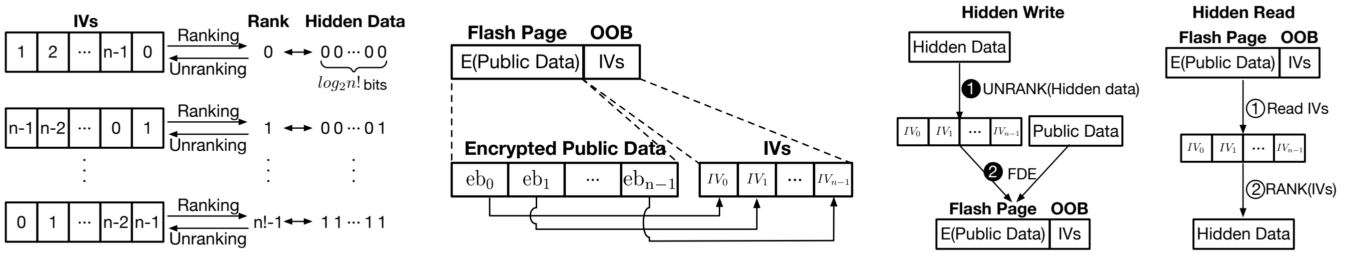
***(2)* Exploiting Random Data Relocation.** Current PD systems [45], [58] designed to defend against multi-snapshot adversaries embed hidden data within previously-filled random data. The GC of flash memory can migrate valid flash pages to other locations for storage efficiency. However, hidden data may need to be migrated during GC, creating distinguishable patterns of random (hidden) data relocation.

Consider a multi-snapshot adversary that runs the PD games for r rounds. When $b = 0$, in round k ($1 \leq k < r$), $\mathcal{O}_{0,k}$ only writes public data to $\mathcal{V}_p$. Since the device is unaware of hidden data in the public mode, GC can directly overwrite the selected $\mathcal{N}_{0,k}$ random data pages $\cup_{i=1}^{\mathcal{N}_{0,k}}\{pg_{0,i}\}$ located at $\cup_{i=1}^{\mathcal{N}_{0,k}}\{addr_{0,i}\}$ by migrating valid public pages to these locations. When $b = 1$, $\mathcal{O}_{1,k}$ writes both public and hidden data. Subsequently, in round $k+1$, $\mathcal{O}_{1,k+1}$ could trigger GC after writing public data. Then, it migrates $\mathcal{N}_{1,k+1}$ valid flash pages to other locations $\cup_{i=1}^{\mathcal{N}_{1,k+1}}\{addr_{1,i}\}$, which might contain hidden data pages $\cup_{i=1}^{\mathcal{N}_{1,k+1}}\{pg_{1,i}\}$. However, to avoid erasing the hidden data, they need to be migrated to other storage space with new data layout $\cup_{i=1}^{\mathcal{N}_{1,k+1}}\{addr'_{1,i}\}$. By comparing storage snapshots taken at rounds k and $k+1$, adversary $\mathcal{A}$ can identify this relocation of data originally marked as random, breaking plausible deniability.

***(3)* Semantic-aware Layout Exploitation.** Current multi-snapshot PD systems [30], [29] also reply on encoding the hidden data within the public data for robust PD guarantees. This requires to altering the public data for the hidden data written. However, modern applications can exhibit specific data access patterns. A well-known phenomenon is data hotness [65], [24], [52], where a small subset of *hot* data serves the most I/O requests while most *cold* data remains unmodified; for example, downloaded files in a web browser typically remain unchanged. Thus, any alterations to cold public data may raise suspicion from adversaries.

When $b = 0$, in round k, $\mathcal{O}_{0,k}$ operates solely on $\mathcal{V}_p$ and writes n files $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$, which corresponds to $\mathcal{N}_{0,1}$ public data pages $\cup_{i=1}^{\mathcal{N}_{0,1}}\{pg_{0,i}\}$. These files and data pages remain unchanged unless explicitly modified or deleted by $\mathcal{O}_{0,k}$. When $b = 1$, $\mathcal{O}_{1,k}$ writes n public files $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ to the storage. However, these files may include cold files $\mathcal{F}' = \{f'_1, f'_2, \ldots, f'_m\}$, where $m < n$ and $\mathcal{F}' \in \mathcal{F}$, which are unlikely to be modified. In round $k+1$, $\mathcal{O}_{1,k+1}$ may cloak the hidden data within the public data associated with $\mathcal{F}'$, resulting in the modification of $\mathcal{F}'$. By analyzing the snapshots taken at rounds k and $k+1$, adversary $\mathcal{A}$ can detect anomalous changes in cold data, thereby compromising plausible deniability.

3. Distinguishable evidence indicates that it contains a data layout generated by $\mathcal{O}_{1,1}$ that cannot be explained by operations in $\mathcal{O}_{0,1}$. Thus, $|Pr(b' = b) - Pr(b' \neq b)| >$ negl(s).



(a) The translation between IV permutations and hidden data.

(b) The data distribution of encrypted public data and IVs in flash memory.

(c) Hidden write and read workflows of MUTE.

Figure 3: The design overview of MUTE, including (a) permutation translations, (b) data distributions, and (c) I/O workflows.

5. MUTE Design

Current PD systems are vulnerable to the DL attack, as they need to explicitly write hidden data to flash memory or alter public data, inevitably changing the data layout.

To solve these limitations, we propose MUTE, which encodes hidden data by permuting the IVs of public data. Each IV permutation corresponds to a rank value that represents the hidden data as shown in Figure 3a. These permuted IVs are stored in the out-of-bound (OOB) area, while the flash page contains the encrypted public data, as illustrated in Figure 3b. The workflow for MUTE's I/O operations in hidden mode is detailed in Figure 3c. For hidden write requests, MUTE first applies the $UNRANK$ function to convert the hidden data into IVs ❶. Then, MUTE uses the generated IVs to encrypt public data during the FDE process before writing the encrypted public data into flash memory, while the IVs are stored in the OOB area ❷. To ensure that the IVs for public data are indistinguishable from those cloaked with hidden data, MUTE randomly shuffles the arrangement of IVs for written public data in public mode.] For hidden read operations, MUTE retrieves the IVs from the OOB area ①. Then, it uses the $RANK$ function to translate the permutation of extracted IVs into the hidden data ②.

5.1. Permutation Primer

MUTE permutes the IV assignments of FDE to write hidden data. A permutation of n elements indicates their arrangement in a specific order. For simplicity, we assume an integer set $S = \{0, 1, \ldots, n-1\}$ with n elements, where each member in S is unique and $n \geq 1$. Then, we define a permutation of S as $\pi = \pi_0, \pi_1, \ldots, \pi_{n-1}$, where $\pi_i \in S$ and $\pi_i \notin \{0, \ldots, \pi_i - 1, \pi_i + 1, \ldots, n-1\}$, $0 \leq i \leq n-1$. Thus, the number of permutations in S is $n! = 1 \cdot 2 \cdots (n-1) \cdot n$.

A bijective ranking $rank$ takes a permutation π as the input to produce an index $rank(\pi)$. Permutation unranking is the reverse function, converting a numeric ranking value v ($0 \leq v \leq n! - 1$) into a permutation $\pi = unrank(v)$. Since the highest rank value is $n! - 1$, the rank value can represent $log_2(n!)$ data bits. A classical algorithm [62] provides $O(n)$ time complexity by simplifying the unranking algorithm. It defines an unranking function and then derives the ranking function from the unranking. A permutation π is initialized as $\pi[i] = i$ for $i = 0, 1, \ldots, n-1$. Thus, the unranking function converts a rank v into a permutation. In addition, the ranking function converts the incoming permutation π into a rank. Before starting the ranking, a permutation π^{-1} is first initialized by iterating $\pi^{-1}[\pi[i]] = i$ for $i = 0, 1, \ldots, n-1$. The unranking and ranking functions are defined as follows.

```
function UNRANK($n$, $\pi$, $v$)
  if n > 0 then
    swap($\pi[n-1]$, $\pi[v \ mod \ n]$)
    unrank($n-1$, $\pi$, $\lfloor v/n \rfloor$)
  end if
  return $\pi$
end function
```

```
function RANK($n$, $\pi$, $\pi^{-1}$)
  if n == 1 then
    return 0
  else
    $mid = \pi[n-1]$
    swap($\pi[n-1]$, $\pi[\pi^{-1}[n-1]]$)
    swap($\pi^{-1}[mid]$, $\pi^{-1}[n-1]$)
    return $mid + n \cdot rank(n-1, \pi, \pi^{-1})$
  end if
end function
```

5.2. Permutation-based Hiding

A flash page serves as an encryption unit containing multiple encryption blocks, each associated with a unique IV, as discussed in Section 2.2. We define a flash page containing n encryption blocks $\{\mathrm{eb}_0, \mathrm{eb}_1, \cdots, \mathrm{eb}_{n-1}\}$, which are assigned with unique IVs $\{\mathrm{IV}_0, \mathrm{IV}_1, \cdots, \mathrm{IV}_{n-1}\}$, where $\mathrm{IV}_i \notin \{\mathrm{IV}_0, \cdots, \mathrm{IV}_{i-1}, \mathrm{IV}_{i+1}, \cdots, \mathrm{IV}_{n-1}\}$, $0 \leq i \leq n-1$. In real-world XTS-AES implementation [42] of flash memory, the IVs of encryption blocks in a data unit are consecutive integers and assigned incrementally. Thus, the IVs of a data unit are defined by equation $IV_i = i$, where $0 \leq i \leq n-1$, and $[\mathrm{IV}_0, \mathrm{IV}_1, \cdots, \mathrm{IV}_{n-1}] = [0, 1, \cdots, \mathrm{n}-1]$.

MUTE controls the allocation of IVs to encryption blocks to store hidden data. Specifically, MUTE defaults the IV set to integer values, where $IV\_SET = \{0, 1, \cdots, n-1\}$, similar to the integer set as discussed in Section 5.1. Then, MUTE assigns IVs to encryption blocks with specific permutations based on the hidden data. In Figure 3a, a permutation of IVs can be translated into a rank value which can be used to represent the hidden data. Thus, MUTE has two main functions for hidden data write and read.

Hidden Write. Hiden data is cloaked within the public data page by permuting IVs. However, the number of encryption

Algorithm 1 MUTE_Write.

```
Require: $H$ = Hidden data, $n$ = Number of encryption blocks in a flash page
1: $l = \log_2(n!)$    /*Number of data bits that a permutation can represent.*/
2: $N_h = \text{len}(H)$    /*Number of data bits in the hidden data.*/
3: $N_{pg} = \lceil N_h/l \rceil$    /*Number of public pages needed for the hidden data.*/
4: $IV_{ini} = \{0, 1, \cdots, n-1\}$    /*Initialize IVs to $\{0, 1, \cdots, n-1\}$.*/
5: for $h_i$ in $H$, where $i = 0, 1, \cdots, N_{pg} - 1$ do /*Divide $H$ into $N_{pg}$ data batches.*/
6:     $rank = h_i$    /*Take the hidden data $h_i$ as a rank value.*/
7:     $IVs = \text{UNRANK}(n, IV_{ini}, rank)$    /*Get IVs via the unrank function.*/
8:     $PG = \text{gc\_get\_pg}()$    /*Get a migrated public page during GC.*/
9:     $PG_{new} = \text{enc}(TV, IVs, PG)$    /*Re-encryption with new $IVs$.*/
10:    $\text{flush}(PG_{new})$    /*Store the $PG_{new}$ to a free flash page.*/
11: end for
```

blocks per flash page is limited, determined by the encryption granularity (128 or 256 bits) and the flash page size. For simplicity, we define the number of encryption blocks in a flash page as n. Algorithm 1 shows the procedure of hidden data write. MUTE first gets the number of hidden bits (i.e., l) that a flash page can represent. This number is determined by $l = \lfloor \log_2(n!) \rfloor$ (see Section 5.1). Afterward, the number of required public data pages N_{pg} for the hidden data is calculated by using l to divide the total bits of hidden data (i.e., N_h). Thus, MUTE splits the hidden data into N_{pg} data batches, each used to generate a permutation of IVs via $UNRANK$ function. Then, MUTE triggers GC to select a valid public data page for migration and re-encrypts it using new IVs. Finally, the re-encrypted public data page is written to a free flash page. More details on MUTE's design for hidden write operations are provided in Section 5.6.

Hidden Read. MUTE handles the hidden read operation as the reversed operation of hidden write. Algorithm 2 shows how the stored IVs can be interpreted into the hidden data. MUTE first locates the flash page PG containing the requested hidden data batch, described in Section 5.6, then reads the IVs of the flash page from its OOB area. Finally, MUTE ranks the permutation of fetched IVs to get the rank value, which is the hidden data.

5.3. Randomizing Public Data Layout

Since hidden data is encrypted before cloaking into the IVs of public data, the generated IVs containing hidden data have a randomized look. However, IVs of the encrypted block are typically generated incrementally. This makes the hidden data distinguishable because the public data without hidden data attached has incremental IVs, unlike the randomized arrangement of IVs that contains the hidden data.

To overcome this limitation, MUTE shuffles IVs before writing public data in public mode. MUTE first generates a cryptography-secure random number as a rank value $rank$. Then, it uses $rank$ in the $UNRANK$ function to obtain a permutation of IVs. Finally, MUTE encrypts the public data page with the generated IVs and writes the encrypted public data into a flash page. MUTE ensures that public data

Algorithm 2 MUTE_READ.

```
Require: $batch_{num}$ = The batch number of the hidden data, $n$ = Number of encryption blocks in a flash page
1: $PG = \text{locate\_pg}(batch_{num})$    /*Locate the page containing the hidden data.*/
2: $IVs = \text{read\_oob}(PG)$    /*Read the OOB area of a page $PG$ to get IVs.*/
3: for $i$ in $0, 1, \cdots, n-1$ do    /*Initialize $IVs^-$ as discussed in Section 5.1.*/
4:     $IVs^-[IVs[i]] = i$
5: end for
6: $rank = \text{RANK}(n, IVs, IVs^-)$    /*Translate $IVs$ to a rank value.*/
7: $h = rank$    /*Take the rank value as hidden data.*/
8: Return $h$    /*Return the hidden data.*/
```

tied with IVs derived from hidden data is indistinguishable from public data encrypted using randomly shuffled IVs. This is because both types of IV permutations are derived through cryptographically secure processes, making them indistinguishable from each other.

5.4. Storage Modes

In public mode, hidden-related metadata should not appear in the DRAM of the storage device to prevent potential side-channel attacks such as the cold boot attack [38] to recover DRAM data. Thus, the FTL should not be aware of hidden data existence in public mode. However, since the GC is triggered when the free space reaches a threshold (e.g., 20% of full capacity), the hidden data can be reclaimed in public mode, leading to the hidden data loss. To overcome this risk, MUTE introduces three PD modes described as follows.

Hidden mode. This is the same as a traditional hidden mode in previous PD systems and can be initiated only with the hidden password. Incoming data is considered hidden and is processed using MUTE's permutation-based hiding method.

Public-hidden mode. This mode is reserved for trusted users to operate on public data and requires both public and hidden passwords to initiate. The public-hidden mode allows the FTL to be aware of the hidden data, thus the FTL can use the metadata of the hidden data stored in the DRAM to avoid the erasure of hidden data during GC.

Public-only mode. This mode is enabled to serve public data while the DRAM of the storage device does not store any hidden-related metadata. Note that this mode is only dedicated to adversaries for decoy purposes.

5.5. Metadata Management

MUTE contains the following metadata of public and hidden data to handle I/O and GC operations, as shown in Figure 4.

Metadata of public data. MUTE uses the current FTL design [46], [37] to manage the metadata of public data. It leverages an address mapping table (AMT) to maintain the translation from LPA to PPA. However, the DRAM size is too small to store the entire AMT. Thus, MUTE creates a cached mapping table (CMT) to cache frequent-accessed

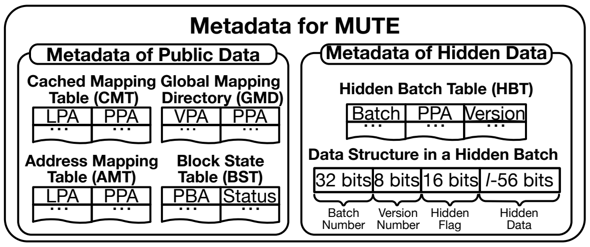


Figure 4: Metadata required by MUTE to serve public and hidden I/Os. In the public metadata, MUTE leverages CMT, GMD and AMT to manage the address mapping, and it uses BST for guiding GC. In the hidden metadata, MUTE creates an HBT to record the mapping from the batch number to the physical hidden data address and version number. Moreover, each hidden batch contains metadata for managing purposes.

mapping entries in the DRAM. When cache misses happen in CMT, MUTE uses a global mapping directory (GMD) to track the mapping entries in the AMT, which is stored in the flash memory. Finally, MUTE introduces a block state table (BST) to record the invalidation status of the pages in a flash block for GC. Note that the invalidation status includes the number of invalidated data pages of a flash block.

Metadata of hidden data. Since each flash page contains l bits of hidden space as discussed in Section 5.2, MUTE treats the l bits of data as a hidden batch. Thus, MUTE creates a hidden batch table (HBT), an in-memory data structure, to maintain the mapping between the batch number of hidden data and the PPA of a flash page, which can locate the IVs stored in its OOB. The HBT contains the version number of each batch. Since hidden data can be overwritten, the version number tracks the newest hidden batch. MUTE reserves 56 bits in a hidden batch as the metadata, including the batch number, version number, and a hidden flag. The batch number serves as an identifier to track the address of hidden data, while the hidden flag indicates the presence of hidden data in the batch. In addition, the hidden flag is truncated from the hidden data's MAC value, which is calculated using the BLAKE2 hash algorithm [1].

5.6. I/O Operations

Public write. Upon the arrival of a public write request, MUTE allocates a free flash page for the incoming data, where the page allocation is consistent with the traditional algorithm as discussed in Section 4. In addition, the written data needs to be encrypted before flushing into the flash page to achieve FDE. Thus, MUTE leverages the encryption method described in Section 5.3. Finally, MUTE updates the AMT and CMT for the LPA with the new PPA.

Public read. MUTE queries the mapping tables to locate the PPA of the read LPA requested by the public read operation. If the LPA hits the CMT, MUTE reads the data page from flash memory using the PPA cached in the CMT. Otherwise, MUTE queries the GMD to pinpoint the mapping entry in the AMT and then reads it to get the PPA. With the PPA, the data page and its IVs can be read from flash memory. Finally, MUTE decrypts the read data using its IVs and returns the decrypted data to complete the read request.

Public-hidden write and read. The public-hidden write and read have the same workflow as the public write and read operations. However, the public-hidden mode differs from the public-only mode in that it enables the FTL to be aware of the hidden metadata. Therefore, MUTE can avoid the erasure of hidden data when the public write operation incurs garbage collection, and we will give more details in the following garbage collection discussion.

Storage warm-up before initiating hidden mode. Since hidden data are cloaked within the public data, the device needs to have sufficient public data for the hidden data. Before activating the hidden mode, users should write adequate public data into the storage device as in previous work [29].

Hidden write. MUTE triggers GC for writing hidden data. The flash block with the most invalidated pages is selected for reclamation during GC. For an incoming hidden write request, MUTE encrypts the hidden data using the hidden key, and the encrypted hidden data can be translated into IVs using Algorithm 1. Afterward, MUTE selects a valid page, which has no hidden data associated, from the chosen flash block and decrypts the page using its IVs, which are stored within the OOB area. Then, MUTE encrypts the valid public data (decrypted) with the IVs, which are generated by the hidden data, and the newly encrypted data is written into a new flash page. Note that the new IVs should be written into the OOB area of the new flash page.

Hidden metadata should be updated accordingly. MUTE first checks the existence of the incoming batch number in HBT. If it exists, MUTE updates the PPA of the entry in the HBT to the new PPA and increments the version number with 1. Otherwise, MUTE inserts a new mapping entry with the batch number and PPA, setting its version number to 1. Finally, before encrypting the hidden data, MUTE attaches the associated metadata to the hidden data payload.

Hidden read. MUTE queries the HBT to locate the physical page address of the hidden data. Then, the IVs in the OOB area of the flash page associated with the searched PPA can be fetched. Therefore, MUTE leverages Algorithm 2 to translate the IVs into the encrypted hidden data, which is further decrypted with the hidden key. Finally, the hidden data can be returned to finish the read request.

Garbage collection. In public-only and public-hidden modes, MUTE selects the flash block with the most invalid pages for reclamation, while migrating valid pages to other free blocks. First, MUTE decrypts the valid public data page using the public key and the old IVs. Afterward, MUTE generates new IVs via Section 5.3 to encrypt the public data before migrating it to a new flash page. However, the selected public data page may include hidden data. In public-hidden mode, MUTE uses Algorithm 2 to rank the old IVs into the encrypted hidden data, which are re-encrypted with the hidden key to alter the original encrypted content, preventing potential side-channel attacks (see Section 6). Finally, MUTE executes Algorithm 1 to cloak the re-encrypted hidden data within a new public data page.

Updating public and hidden data generates stale data. This brings two challenges. *(1)* Invalidated public data pages may contain hidden data, which should be preserved during GC. *(2)* Hidden data cloaked within a public data page might be outdated and should not be migrated. For challenge *(1)*, MUTE inspects the invalidated flash pages before the flash block erasure. Specifically, MUTE verifies the presence of hidden data by checking the hidden flag. If hidden data exists, MUTE re-encrypts it and attaches it to a valid public page for migration during GC. Otherwise, the flash page is erased directly. For challenge *(2)*, before migrating hidden data, MUTE verifies the version number of the hidden data using the HBT to preserve the newest hidden data by attaching it to a new public flash page. If the hidden data is not the latest version, MUTE does not migrate it.

6. PD Analysis For MUTE

We verify the security of MUTE using the PD game described in Section 3.3. The PD game is conducted over r rounds. In round k ($1 \leq k \leq r$), the PD game has access set $\mathcal{O}_{0,k}$ on public volume and set $\mathcal{O}_{1,k}$ on both public and hidden volumes. Based on Definition 1, accesses in $\mathcal{O}_{1,k}$ can be plausibly explained by accesses in $\mathcal{O}_{0,i}$, $1 \leq i \leq k$. Additionally, we apply a restriction to the PD game: both $\mathcal{O}_{0,k}$ and $\mathcal{O}_{1,k}$ can generate sufficient public data to the device that can trigger GC. This assumption is reasonable, as users can warmup the device before writing hidden data securely during the operating $session$ as discussed in Section 3.1.

When $b = 0$, $\mathcal{O}_{0,k}$ accesses the public volume $\mathcal{V}_p$, captured after the $(k-1)$th round finished. MUTE allocates a free flash page for the public data and generates random IVs to encrypt it using the public key K_p. When the GC is triggered by $\mathcal{O}_{0,k}$, MUTE selects a flash block with the most invalidated public data pages. Then, it migrates the valid public pages to other free blocks by decrypting the encrypted pages, re-generating new random IVs, and re-encrypting the valid pages with the new IVs. For the public pages that include hidden data, MUTE decodes the IVs of them to get the hidden data using the ranking operation, re-encrypts the hidden data to get a new rank value, and generates new IVs using the unranking function for migrated public pages.

When $b = 1$, $\mathcal{O}_{1,k}$ operates both public and hidden data on public $\mathcal{V}_p$ and hidden $\mathcal{V}_h$ volumes. MUTE triggers the GC to write hidden data. Then, MUTE leverages the unranking function to translate the hidden data into IVs. To avoid the erasure of hidden data, MUTE traverses the IVs of each flash page in a selected flash block to identify the existence of hidden data in the IVs of each flash page. For the detected hidden data, MUTE re-encrypts it to alter its original data content. This ensures the randomness of the IV generation, consistent with the operation in public mode.

Lemma 1. *After $\mathcal{O}_{1,k}$ in kth round, for the content of snapshoted n public flash pages $\cup_{i=1}^{n}\{pg_i\}$, it has $pg_j \notin \cup_{i=1}^{j-1}\{pg_i\} \cup \cup_{i=j+1}^{n}\{pg_i\}$ and $IVs_j \notin \cup_{i=1}^{j-1}\{IVs_i\} \cup \cup_{i=j+1}^{n}\{IVs_i\}$, where $1 \leq j \leq n$.*

Proof. Lemma 1 fails if the adversary $\mathcal{A}$ can find an $\mathcal{O}_{1,k}$ that results in duplicated public flash pages or duplicated IVs. However, MUTE uses secure cryptography with the secure parameter s, as defined in Section 3.3. The encrypted hidden data, which serves as IVs for public data encryption, exhibits a cryptographically-secure randomness, indistinguishable from the random number used to shuffle IVs in public mode. Thus, the public data pages exhibit no replication. In addition, during GC, hidden data is re-encrypted when moved to free pages. Thus, MUTE ensures that IVs assigned to public data pages are not duplicated. □

Theorem 1. *After $\mathcal{O}_{1,k}$, the snapshot DS_k has the data layout $\cup_{i=1}^{n}\{addr_i\}$ with n public flash pages $\cup_{i=1}^{n}\{pg_i\}$. Then, it can be replayed by applying $\mathcal{O}_{0,k}$ on the prior data snapshot DS_{k-1} which is generated by $\mathcal{O}_{0,k-1}$ or $\mathcal{O}_{1,k-1}$.*

Proof. Assume that $\mathcal{O}_{1,k}$ contains the access pattern $\mathcal{O}_p$ on $\mathcal{V}_p$, writing $\mathcal{N}_p$ public data pages. Under the restriction discussed in Section 3.3, $\mathcal{O}_{0,k}$ writes the same amount of public data (i.e., $\mathcal{N}_p$) as in $\mathcal{O}_{1,k}$. If $\mathcal{O}_{1,k}$ contains no access to the hidden volume $\mathcal{V}_h$, DS_k can be easily replayed by choosing $\mathcal{O}_{0,k} = \mathcal{O}_p$, which accesses only the public volume $\mathcal{V}_p$ as the access patterns of $\mathcal{O}_{1,k}$. When $\mathcal{O}_{1,k}$ includes accesses to $\mathcal{V}_h$, $\mathcal{O}_p$ can trigger GC to ensure that hidden data is cloaked within the IVs of the migrated public data pages. Since the fundamental GC logic of MUTE remains consistent in both public and hidden modes, and only the public data written affects the GC process, the adversary $\mathcal{A}$ can select $\mathcal{O}_p$ as $\mathcal{O}_{0,k}$ to replicate the same GC behavior as in $\mathcal{O}_{1,k}$. Based on Lamma 1, $\mathcal{A}$ cannot identify the IV and data content changes caused by hidden data operations. Thus, DS_k can be plausibly explained by executing $\mathcal{O}_{0,k}$. □

Theorem 1 demonstrates that, with MUTE, the execution of $\mathcal{O}_{1,k}$ can always be plausibly explained by the accesses in $\mathcal{O}_{0,k}$, thereby satisfying Definition 1 as discussed in Section 3.3. Thus, MUTE provides strong PD guarantees.

7. Implementation

We implement MUTE in FEMU [53], a prevalent SSD emulator adopted by the system community for SSD-related research [54], [79], [80], [43]. FEMU is built in QEMU [4], a full-stack virtual machine. The emulated SSD contains 512GB flash memory with parameters detailed in Appendix B.1. We implement an FDE module using XTS-AES-128 from the QEMU crypto library, performing encryption and decryption operations before data is written to or read from the flash memory. Thus, an IV of an encryption data block is 16 bytes, and a flash page has 256 encryption blocks corresponding to 256 IVs, where each IV consumes 1 Byte. In addition, the TV and IVs of a flash page are stored in its out-of-bound (OOB) area, which is 409 B, 10% of the flash page [55]. Since the permutation's arithmetic operation involves the integer with hundreds of data bytes, we use libGMP [5] for large integer calculation.

8. Evaluation

This section answers two research questions. *RQ1:* Does MUTE provide sufficient performance? *RQ2:* How large is

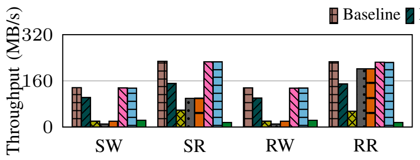


Figure 5: The bandwidth of Baseline, PEARL, MDEFTL and MUTE when testing FIO benchmarks.

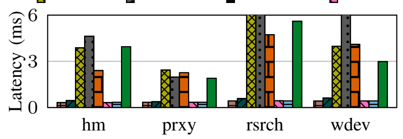


Figure 6: The average latency of Baseline, PEARL, MDEFTL and MUTE when evaluating MSR workloads.

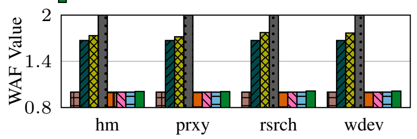


Figure 7: The average WAF of Baseline, PEARL, MDEFTL and MUTE when evaluating MSR workloads.

the MUTE's hidden volume? We address *RQ1* in Section 8.2 by comparing the performance of MUTE with current SSDs and PD schemes. Then, we characterize the capacity of hidden volume in Section 8.3 to answer *RQ2*.

8.1. Experimental Setup

We use an Intel Xeon E3-1245 v5 @ 3.50GHZ 8-core processor with 64GB DRAM, while installed with Ubuntu 22.04.5 with kernel 5.13.4 as the host OS. Then, we build a guest system VM for FEMU by allocating a 50GB QCOW2 image file and installing Ubuntu 18.04 along with kernel 4.15.0 on the VM; the guest system is allocated with 4GB DRAM and four vCPUs.

Workloads. We evaluate MUTE using Flexible I/O Tester (FIO) [3] and real-world workloads from Microsoft (MSR [63]). FIO is a standard I/O benchmark for block-based storage systems, while MSR workloads are collected from server racks at Microsoft Research detailed in Appendix B.1.

Comparison Selection. We evaluate MUTE in three modes: public-only (MUTE-PO), public-hidden (MUTE-PH), and hidden (MUTE-H). Then, we use an unmodified FEMU SSD with FDE as the baseline, maintaining identical SSD parameters with MUTE. Additionally, we re-implemented PEARL [29] and MDEFTL [45], which are state-of-the-art PD schemes against multi-snapshot attacks. Further implementation details are provided in the Appendix B.2.

SSD Initialization. In public mode, experiments begin with an empty SSD. However, both MUTE and PEARL require sufficient public data written for the hidden mode. Thus, we use an FIO sequential write workload to fill the SSD with public data, ensuring GC can be triggered in hidden mode.

8.2. Performance and Lifetime Testing

Bandwidth of MUTE vs Baseline. We evaluate throughput using FIO workloads: sequential write (SW), sequential read (SR), random write (RW), and random read (RR), each with a 4 KB request size. Figure 5 shows that the average bandwidths of Baseline, MUTE-PO, MUTE-PH, and MUTE-H are 181.5 MB/s, 180.3 MB/s, 180 MB/s, and 19.5 MB/s, respectively. In public mode, MUTE-PO and MUTE-PH reduce bandwidth over the Baseline by 0.7% and 0.9%. MUTE introduces small performance overhead, as each write request only requires one unranking operation, which is small relative to the flash access latency. In hidden mode, MUTE-H significantly degrades the performance, as

TABLE 1: Latency percentage of MUTE-H per operations.

Trace	FDE	PERM	FLASH	FTL
hm	0.5%	0.2%	93.9%	5.4%
prxy	0.7%	0.2%	94.8%	4.2%
rsrch	0.4%	0.1%	94.5%	5.0%
wdev	0.5%	0.2%	92.9%	6.4%

MUTE writes hidden data during GC. Figure 6 shows the latency under real-world workloads. MUTE-PO and MUTE-PH increase the latency over the Baseline by 1.3% and 2%, whereas MUTE-H results in 9.6x increase in latency. We further break down MUTE-H's access latency, revealing that its primary performance penalty (94% of total I/O latency) stems from flash media access. A detailed latency breakdown is provided in Appendix C.2.

While the throughput of MUTE in the hidden mode drops significantly, MUTE still provides HDD-level bandwidth for hidden data operation. For example, a Seagate Barracuda HDD [2] offers 22.7 MB/s for 4 KB sequential write and 18 MB/s for 4KB sequential read, whereas MUTE-H provides 23.4 MB/s and 15.7 MB/s, respectively, outperforming HDD-based PD systems [25], [26], [68].

Bandwidth of MUTE vs PEARL and MDEFTL. In public mode, Figure 5 shows that MUTE achieves 43%-124% higher bandwidth than PEARL and MDEFTL. MUTE maintains normal I/O bandwidth in public mode, significantly outperforming PEARL and MDEFTL. In hidden mode, MUTE provides approximately 15% better write bandwidth but reduces read bandwidth by 71%-92%. The better write bandwidth of MUTE is due to avoiding PEARL's time-consuming WOM operations and MDEFTL's extra dummy writes, while the reduced read performance results from MUTE's ranking operations. As discussed in Section 3.1, users access the hidden data within a secure "session", indicating a secure operation environment without time stress. Thus, users have sufficient time to manage their hidden data, making the reduced bandwidth acceptable. More discussion about the hidden read overhead is provided in Appendix C.2. In real-world workloads (Figure 6), MUTE decreases the average latency over PEARL in the hidden mode by 12.5% and increases it over MDEFTL by 15.3%. These results show that MUTE offers comparable I/O performance over current flash-based PD solutions.

Latency Breakdown. We break down the access latency of MUTE-H into specific SSD operations in Table 1, where *FDE* indicates data encryption, *PERM* is the computational overhead of permutation operations, *FLASH* is the latency

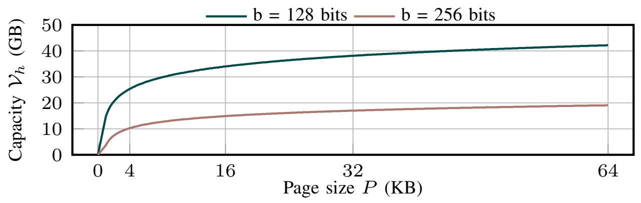


Figure 8: Hidden volume capacity $\mathcal{V}_h$ for different page size P when total capacity C is 512GB and M is 56 bits

for accessing flash memory, and *FTL* refers to other computational overhead in the FTL. The primary performance penalty of MUTE arises from flash media access, which accounts for 94% of the total I/O latency on average. This is because MUTE-H must trigger GC for hidden data writes, and the associated data migrations and flash block erasure significantly slow down hidden data operations.

Metadata Recovery Overhead. When switching the device from public to hidden mode, MUTE needs to recover the hidden data metadata (HBT). This process needs to read the OOB area and perform permutation computations. Our evaluation shows that reconstructing the HBT in DRAM takes approximately 4.2 minutes, with 83 seconds on reading the OOB area and 168 seconds on computational operations.

Lifetime Testing. The write amplification factor (WAF) measures the ratio of internally written data to user-written data, where a higher WAF indicates a shorter storage lifespan. Figure 7 shows that MUTE does not impact storage lifetime in public mode, whereas PEARL-P and MDEFTL-P increase the WAF value significantly by 67% and 100%, respectively. This is because MUTE does not modify the GC logic of the FTL, while other schemes incur additional writes. In hidden mode, MUTE-H incurs trivial lifetime overhead with a 1% WAF increase over Baseline and MDEFTL, whereas PEARL greatly increases the WAF value by 74.4%. Since MUTE initiates GC without extra data migrations, it imposes minimal overhead on storage lifetime.

8.3. Characterizing Hidden Volume Capacity

Assume the total capacity for the entire storage device is C, the page size is P and the encryption block size is b. The metadata size of a hidden batch is M which is a constant. Then, a page has $q = \frac{P}{b}$ encryption blocks. Thus, we can get the capacity of the hidden volume (in bits) $\mathcal{V}_h$ as follows based on the Stirling formula [72]:

$$\mathcal{V}_h > \frac{C}{b} \cdot (\log_2(q) - \eta) \tag{1}$$

where $\eta = 2 + \frac{M}{q}$ which approaches to 2 when q grows. We show the full derivation in Appendix C.3.

When C and b are constants, the hidden capacity expands logarithmically with respect to page size P. Figure 8 shows the hidden capacity as a function of the page size P when total capacity C is 512 GB and M is 56 bits. Our proof-of-concept implementation sets P to 4 KB. Thus, MUTE provides 25.3GB hidden volume when $b = 128$, while 10.2 GB is provided when $b = 256$, comparable to 12 GB of PEARL and hundreds of megabytes of MDEFTL.

Page sizes typically vary from 4 KB to 16 KB [56] and cannot be indefinitely enlarged. In this case, our MUTE implementation can only provide 33.9 GB or 14.8 GB capacity for the hidden data at maximum – P is 16 KB – when the encryption block size b is 128 bits or 256 bits. To overcome this limitation, a potential solution is to aggregate several flash pages into a flash group for larger q and subsequently permute the IVs of encryption blocks within this group.

9. Discussion

Risk of randomized IVs. MUTE ensures PD by randomizing the assignment of IVs to public data during FDE. However, the traditional XTS-AES implementation generates IVs sequentially for a data unit. This could raise the question: *why does your device employ such an unconventional implementation?* Such type of concern exists in all current PD systems, as they often alter public data layouts or use ad-hoc software that could signal the use of PD strategies to adversaries. For example, current FDE systems typically implement standard encryption by filling the storage device with random data [22], [10], [26], [28], [7]. This practice seems to contradict the conventional usage of storage devices wherein users write only the necessary data and reveal the PD presence. However, the sole presence of a PD system cannot serve as evidence of the existence of hidden data as discussed in Section 3. Thus, randomizing IVs does not compromise the PD guarantee.

Secure crypto implementation. Public and hidden data are encrypted before being written into the flash pages and IVs, requiring a secure crypto implementation. MUTE employs XTS-AES for encryption and ensures security by focusing on two key aspects: *(1)* secure random number generation (RNG) and *(2)* secure re-encryption of hidden data. RNG is dedicated to producing TV and randomized IVs, which are stored in the flash page's OOB area. Typically, flash memory RNGs use noise sources, such as thermal and random telegraph noises in flash cells, to generate high-quality random values [70]. Thus, MUTE leverages the current secure RNG implementation for FDE. For *(2)*, the hidden data are encrypted with the new TV to generate different IVs after its relocation during GC. This obfuscates discernible evidence of hidden data existence as it has the identical pattern of the IV generation in public mode.

IV collisions. The IVs within a flash page do not have collisions, as they are permuted using incremental integers. In addition, the permutation of IVs is created by unranking the cryptographically secure random data encrypted hidden data. Since MUTE leverages XTS-AES, which is IND-CPA, with a cryptographically secure implementation to encrypt hidden data, the generated permutations have no collision. Thus, IV collisions do not occur in MUTE.

DoS attacks. MUTE creates a public-only mode as a decoy for adversaries. Upon coercion, users disclose only the public key, granting adversaries access to public-only mode and denying the existence of hidden data. When adversaries solely capture snapshots of the raw data without initiating writes, GC is unlikely to occur, thereby preventing hidden data loss. However, adversaries might execute a denial of service (DoS) attack [29] by overwhelming the device with excessive data writes. Since operating hidden data in the presence of adversaries could reveal its existence, a secure PD system must make the device completely unaware of hidden data under such conditions. Thus, the DoS attack can erase all data from the storage, yet they do not reveal the presence of hidden data, thereby preserving user safety. Current PD methods [67], [44], [29], [45], [25], [26], [28], including MUTE, primarily consider protecting the deniability of hidden data without a solution to defend against DoS attack. We leave this as a promising direction in our future work.

Hidden data loss risk. Previous PD systems write hidden data directly into flash pages, occupying the space of public data. This decreases usability, as normal public data operations from legitimate users can inadvertently erase hidden data. A potential remedy is to restrict the available storage capacity in public mode. However, this creates a noticeable discrepancy between the actual and available storage capacities, alerting adversaries. In contrast, MUTE requires users to switch to public-hidden mode when accessing public data. By embedding hidden data within the permutation of IVs, it avoids using extra storage space for hidden data. Even if users fill the storage with public data, the hidden data remains intact without being overwritten.

Performance side channel. MUTE reduces SSD performance in the hidden mode. However, adversaries cannot use this to infer the hidden data, as they lack access to the device during hidden mode operations, as discussed in Section 3.

Amount of public data written. MUTE requires users to write sufficient public data to accommodate hidden data as discussed in Section 5.6. Moreover, the public data volume should be sufficient to nearly trigger GC; otherwise, adversaries may become suspicious if GC occurs on a storage device with minimal public data. In our implementation aligned with FEMU [53], GC initiates when 95% of storage space is utilized and stops when usage drops below 75%. In MUTE, users must write at least 75% of the device's total storage capacity with public data. Note that this threshold for public data may vary across different FTL implementations.

Unexpected Session Termination. If a private session is interrupted accidently (e.g., power loss), data (e.g., hidden data's metadata) in DRAM may be at risk. However, modern SSDs are typically equipped with power loss prevention (PLP) by employing a capacitor- [69] or battery-backed [49] DRAM. The data in the DRAM can be securely written back to flash memory upon an accidental event.

TEE-based PD Systems. Recent PD solutions [58], [31], [57] leverage trusted execution environments (TEEs) to securely implement plausible deniability and prevent reverse engineering the executable files containing the PD logic. However, these works overlook the suspicious data layout patterns on flash memory resulting from PD operations. Thus, adversaries capable of inspecting raw flash layouts can still employ a DL attack and compromise the PD guarantees provided by TEE-based solutions.

10. Related Work

Steganographic File System. Previous works [22], [59], [22], [66] incorporate data hiding into a file system, obfuscating the hidden data storage. For instance, Han et al. [39] introduce DRSteg to provide files' plausible deniability in a multi-user computing environment. However, these methods only apply to traditional HDDs. With the arise of flash memory, these methodologies become vulnerable to adversaries who can perform off-the-shelf analysis on flash chips.

Plausibly Deniable Block Storage. FDE tools like TrueCrypt [6], VeraCrypt [7], and Rubberhose [10] provide PD guarantees at the host OS but are vulnerable to multi-snapshot adversaries. HIVE [25] was introduced to counteract multi-snapshot attacks through write-only Oblivious RAM (ORAM). Change et al. developed MobiCeal [28] to avoid the poor performance of ORAM-based solutions [25], [26] via dummy writes. Nonetheless, it fails to account for the unique property of flash memory, compromising its PD guarantees. INVISILINE [68] proposed an invisible PD by embedding hidden data within the metadata of dm-crypt [17]. However, it assumes using traditional HDDs and is incompatible with flash memory. Moreover, INVISILINE results in impractical storage capacities for hidden data. Further details on INVISILINE are provided in Appendix A.3.

Plausibly Deniable Flash-based Storage. Multiple PD systems were proposed for flash memory. DEFY [67] proposed a PD system on YAFFS, allowing users to deny the hidden data existence. Jia et al. [44] identified suspicious data distribution and proposed DEFTL to ensure a strong PD guarantee. However, neither DEFY nor DEFTL can defend against the multi-snapshot adversary. Thus, Chen et al. proposed PEARL [29] against multi-snapshot attackers by leveraging the WOM coding to ensure a strong PD guarantee. INFUSE [30] hides sensitive data by manipulating the voltage threshold of flash cells. MDEFTL [45] and FSPDE [58] are methods that conceal hidden data within dummy writes, which are generated during regular write operations. However, these solutions occupy explicit storage space for hidden data and remain vulnerable to DL attacks.

11. Conclusion

This paper identifies that the data layout on flash memory can lead to the compromise of PD guarantees. To counteract this threat, we propose MUTE to provide strong PD protection for the hidden data, ensuring the privacy and safety of users. Our proof-of-concept evaluation shows that MUTE assures PD protection while providing sufficient throughput and capacity for hidden data operations.

Acknowledgement

We sincerely thank anonymous reviewers for their valuable comments, helping us to improve our work. This work was supported in part by the National Science Foundation under grants CNS-2055123 and CNS-2055014.

References

[1] Blake2: simpler, smaller, fast as md5. https://www.blake2.net/blake2.pdf.

[2] Desktop HDD Product Manual. https://www.seagate.com/www-content/product-content/desktop-hdd-fam/en-us/docs/100768625b.pdf.

[3] Flexible I/O Tester. https://github.com/axboe/fio.

[4] QEMU. https://www.qemu.org/.

[5] The GNU MP Bignum Library. https://gmplib.org/.

[6] TrueCrypt. https://truecrypt.sourceforge.net/.

[7] VeraCrypt. https://www.veracrypt.fr/en/Home.html.

[8] What is a USB security key, and how do you use it? https://www.tomsguide.com/news/usb-security-key.

[9] Regulation of Investigatory Powers Act 2000/Part III. https://wiki.openrightsgroup.org/wiki/Regulation_of_Investigatory_Powers_Act_2000/Part_III, 2000.

[10] Rubberhose. https://web.archive.org/web/20100915130330/http://iq.org/~proff/rubberhose.org/, 2001.

[11] Alternate Data Streams Overview. https://www.sans.org/blog/alternate-data-streams-overview/, 2008.

[12] How a Syrian refugee risked his life to bear witness to atrocities. https://www.thestar.com/news/world/how-a-syrian-refugee-risked-his-life-to-bear-witness-to-atrocities/article_1321cb97-7845-5591-9343-f8cf96d62200.html, 2012.

[13] Cybersecurity Law of the People's Republic of China. https://digichina.stanford.edu/work/translation-cybersecurity-law-of-the-peoples-republic-of-china-effective-june-1-2017/, 2017.

[14] General Data Protection Regulation. https://gdpr-info.eu/, 2018.

[15] Australia's New Anti-Encryption Law Is Unprecedented and Undermines Global Privacy. https://fee.org/articles/australia-s-unprecedented-encryption-law-is-a-threat-to-global-privacy/, 2019.

[16] Anti-Extremism Policies in Russia and How they Work in Practice. https://www.wilsoncenter.org/publication/anti-extremism-policies-russia-and-how-they-work-practice, 2024.

[17] Dm-crypt. https://wiki.archlinux.org/title/dm-crypt, 2024.

[18] Full-disk encryption (FDE). https://www.techtarget.com/whatis/definition/full-disk-encryption-FDE, 2024.

[19] SSD firmware software download. https://www.micron.com/products/storage/ssd/micron-ssd-firmware, 2024.

[20] Update the firmware of your Samsung SSD. https://www.samsung.com/ca/support/memory-storage/update-the-firmware-of-your-samsung-ssd/, 2024.

[21] AceLab. PC-3000 SSD Systems. The List of Supported SSDs (regularly updated, ver. 3.4.12). https://blog.acelab.eu.com/pc-3000-ssd-list-of-supported-ssd-drives-regularly-updated.html, 2024.

[22] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *International Workshop on Information Hiding*, 1998.

[23] A. Aravindan. Flash 101: NAND Flash vs NOR Flash. https://www.embedded.com/flash-101-nand-flash-vs-nor-flash/, 2018.

[24] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, et al. The CacheLib caching engine: Design and experiences at scale. In *OSDI*, 2020.

[25] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *CCS*, 2014.

[26] A. Chakraborti, C. Chen, and R. Sion. Datalair: Efficient block storage with plausible deniability against multi-snapshot adversaries. In *PETS*, 2017.

[27] B. Chang, Z. Wang, B. Chen, and F. Zhang. Mobipluto: File system friendly deniable storage for mobile devices. In *ACSAC*, 2015.

[28] B. Chang, F. Zhang, B. Chen, Y. Li, W.-T. Zhu, Y. Tian, Z. Wang, and A. Ching. Mobiceal: Towards secure and practical plausibly deniable encryption on mobile devices. In *IEEE/IFIP DSN*, 2018.

[29] C. Chen, A. Chakrabort, and R. Sion. PEARL: Plausibly deniable flash translation layer using WOM coding. In *USENIX Security*, 2021.

[30] C. Chen, A. Chakraborti, and R. Sion. Infuse: Invisible plausibly-deniable file system for nand flash. In *PETS*, 2020.

[31] N. Chen and B. Chen. Hipds: A storage hardware-independent plausibly deniable storage system. *IEEE Transactions on Information Forensics and Security*, 2024.

[32] N. Chen, B. Chen, and W. Shi. A cross-layer plausibly deniable encryption system for mobile devices. In *International Conference on Security and Privacy in Communication Systems*, 2022.

[33] CPJ. Nothing to declare: Why U.S. border agency's vast stop and search powers undermine press freedom. https://cpj.org/reports/2018/10/nothing-to-declare-us-border-search-phone-press-freedom-cbp/, 2018.

[34] M. Dworkin. Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices. *NIST Special Publication (SP)*, 2010.

[35] M. Fan, R. Ranjit, A. Thurber, and D. Engelhard. High resolution profiles of 3D NAND pillars using x-ray scattering metrology. In *Metrology, Inspection, and Process Control for Semiconductor Manufacturing XXXV*, 2021.

[36] W. Feng, C. Liu, Z. Guo, T. Baker, G. Wang, M. Wang, B. Cheng, and J. Chen. Mobigyges: A mobile hidden volume for preventing data loss, improving storage utilization, and avoiding device reboot. *Future Generation Computer Systems*, 2020.

[37] A. Gupta, Y. Kim, and B. Urgaonkar. Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS*, 2009.

[38] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, et al. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 2009.

[39] J. Han, M. Pan, D. Gao, and H. Pang. A multi-user steganographic file system on untrusted shared storage. In *ACSAC*, 2010.

[40] M. M. Hasan and B. Ray. Data recovery from "scrubbed" NAND flash storage: Need for analog sanitization. In *29th USENIX Security Symposium (USENIX Security)*, 2020.

[41] J. Hirota, K. Yamasue, and Y. Cho. Profiling of carriers in a 3d flash memory cell with nanometer-level resolution using scanning nonlinear dielectric microscopy. *Microelectronics Reliability*, 2020.

[42] IEEE. Cryptographic protection of data on block-oriented storage devices. *IEEE Std 1619-2018*, 2019.

[43] S. Jaffer, K. Mahdaviani, and B. Schroeder. Improving the reliability of next generation SSDs using WOM-v codes. In *USENIX FAST)*, 2022.

[44] S. Jia, L. Xia, B. Chen, and P. Liu. Deftl: Implementing plausibly deniable encryption in flash translation layer. In *ACM CCS*, 2017.

[45] S. Jia, Q. Zhang, L. Xia, J. Jing, and P. Liu. Mdeftl: Incorporating multi-snapshot plausible deniability into flash translation layer. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[46] S. Jiang, Lei Zhang, XinHao Yuan, Hao Hu, and Yu Chen. S-FTL: An efficient address translation for flash memory by exploiting spatial locality. In *MSST*, 2011.

[47] M. Jung and M. Kandemir. An evaluation of different page allocation strategies on high-speed ssds. In *USENIX HotStorage*, 2012.

[48] B. Kaliski. Password-Based Cryptography Specification Version 2.0. https://www.ietf.org/rfc/rfc2898.txt, 2000.

[49] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger. Viyojit: Decoupling battery and dram capacities for battery-backed dram. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[50] Kicksecure. Cold Boot Attack Defense. https://www.kicksecure.com/wiki/Cold_Boot_Attack_Defense#cite_note-wired_coldboot-9, 2025.

[51] J. Kim, J. Lee, J. Choi, D. Lee, and S. H. Noh. Improving SSD reliability with RAID via Elastic Striping and Anywhere Parity. In *IEEE/IFIP DSN*, 2013.

[52] B. Li, C. Deng, J. Yang, D. Lilja, B. Yuan, and D. Du. Haml-ssd: A hardware accelerated hotness-aware machine learning based ssd management. In *IEEE/ACM ICCAD*, 2019.

[53] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *USENIX FAST*, 2018.

[54] H. Li, M. L. Putra, R. Shi, X. Lin, G. R. Ganger, and H. S. Gunawi. Ioda: A host/device co-design for strong predictability contract on modern flash storage. In *ACM SOSP*, 2021.

[55] Q. Li, M. Ye, Y. Cui, L. Shi, X. Li, T.-W. Kuo, and C. J. Xue. Shaving retries with sentinels for fast read over high-density 3d flash. In *IEEE/ACM MICRO*, 2020.

[56] S. Liang, Z. Qiao, S. Tang, J. Hochstetler, S. Fu, W. Shi, and H.-B. Chen. An empirical study of quad-level cell (qlc) nand flash ssds for big data applications. In *IEEE BigData*, 2019.

[57] J. Liao, B. Chen, and W. Shi. Trustzone enhanced plausibly deniable encryption system for mobile devices. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, 2021.

[58] J. Liao, N. Chen, L. Xia, B. Chen, and W. Shi. Fspde: A full stack plausibly deniable encryption system for mobile devices. In *ACM CODASPY*, 2024.

[59] A. D. McDonald and M. G. Kuhn. Stegfs: A steganographic file system for linux. In *Intl. Worskhop Info. Hiding*, 2000.

[60] C. Meijer and B. van Gastel. Self-encrypting deception: Weaknesses in the encryption of solid state drives. In *IEEE S&P*, 2019.

[61] Mike Szczys. Reading NAND Flash Chips Without Removing Them. https://hackaday.com/2010/12/24/reading-nand-flash-chips-without-removing-them/, 2010.

[62] W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 2001.

[63] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *FAST*, 2008.

[64] Nexstor. HDD vs Flash: The Differences Between Flash Storage and HDD. https://nexstor.com/hdd-vs-flash-storage/, 2018.

[65] U. of Michigan. Power laws, Pareto distributions and Zipf's law. https://public.websites.umich.edu/~mejn/courses/2006/cmplxsys899/powerlaws.pdf, 2006.

[66] H. Pang, K.-L. Tan, and X. Zhou. Steganographic schemes for file system and b-tree. *IEEE Transactions on Knowledge and Data Engineering*, 2004.

[67] T. M. Peters, M. A. Gondree, and Z. N. Peterson. Defy: A deniable, encrypted file system for log-structured storage. In *NDSS*, 2015.

[68] S. K. Pinjala, B. Carbunar, A. Chakraborti, and R. Sion. INVISILINE: Invisible Plausibly-Deniable Storage. In *IEEE S&P*, 2024.

[69] T. Pott and I. Thomson. Supercapacitors have the power to save you from data loss. https://www.theregister.com/2014/09/24/storage_supercapacitors/, 2023.

[70] B. Ray and A. Milenković. True random number generation using read noise of flash memory cells. *IEEE Transactions on Electron Devices*, 2018.

[71] J. Reardon, S. Capkun, and D. Basin. Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory. In *21st USENIX Security Symposium (USENIX Security)*, 2012.

[72] H. Robbins. A remark on stirling's formula. *The American mathematical monthly*, 62(1):26–29, 1955.

[73] L. Shorser. Proof By Counterexample. https://www.math.utoronto.ca/writing/Counterexample.pdf, 2025.

[74] A. Skillen and M. Mannan. On implementing deniable storage encryption for mobile devices. *20th Network and Distributed System Security Symposium (NDSS)*, 2013.

[75] A. Team. How to Recover Data from the NAND Flash Drives with the COB (Chip on Board) Memory Chips. https://blog.acelab.eu.com/pc-3000-flash-how-to-read-cob-chip-on-board-memory.html, 2020.

[76] J. Voris, N. Saxena, and T. Halevi. Accelerometers and randomness: perfect together. In *ACM WiSec*, 2011.

[77] X. Yu, B. Chen, Z. Wang, B. Chang, W. T. Zhu, and J. Jing. Mobihydra: Pragmatic and multi-level plausibly deniable encryption storage for mobile devices. In *Info. Sec. Conf. (ISC)*.

[78] Q. Zhang, S. Jia, J. He, X. Zhao, L. Xia, Y. Niu, and J. Jing. Ensuring data confidentiality with a secure xts-aes design in flash translation layer. In *2020 IEEE 5th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*, 2020.

[79] Y. Zhou, Q. Wu, F. Wu, H. Jiang, J. Zhou, and C. Xie. Remap-SSD: Safely and efficiently exploiting SSD address remapping to eliminate duplicate writes. In *USENIX FAST*, 2021.

[80] W. Zhu and K. R. B. Butler. Nasa: Nvm-assisted secure deletion for flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

Appendix A. Additional PD Proofs and Discussions

A.1. Compromising Single-snapshot PD

Current flash-based single-snapshot PD systems [44], [67] ensure plausible deniability by acting the hidden data as random bits. However, they are vulnerable to the DL attack.

Theorem 2. *Current single-snapshot PD systems for flash memory are vulnerable, as they leave distinguishable[4] patterns within the data layout generated by $\mathcal{O}_{0,1}$ and $\mathcal{O}_{1,1}$.*

Proof. The PD game runs for one round (steps 3 to 5) for a single-snapshot adversary. For example, DEFY [67] allocates flash pages for the incoming data using the inherent channel-first method. When $b = 0$, the accesses in $\mathcal{O}_{0,1}$ write public data in $\mathcal{V}_p$, generating a <public, random> data layout. When $b = 1$, the accesses in $\mathcal{O}_{1,1}$ contain both public and hidden writes to $\mathcal{V}_p$ and $\mathcal{V}_h$. However, DEFY adopts the original page allocation algorithm consistent with the method when accessing $\mathcal{V}_p$. This can lead to a <public, hidden, public> data layout, incurred by access patterns $\mathcal{O}_{1,1} =< \mathcal{O}_p, \mathcal{O}_h, \mathcal{O}'_p >$, where $\mathcal{O}_p$ and $\mathcal{O}'_p$ are accesses of public data, and $\mathcal{O}_h$ accesses hidden data. Since challenger $\mathcal{C}$ only reveals the public password, $\mathcal{A}$ interprets the hidden data as random, observing a <public, random, public> data layout, which is distinguishable from the <public, random> layout generated by $\mathcal{O}_{0,1}$.

Another example is DEFTL [44], which proposes a new flash page allocation by creating a flash block pool that contains all free blocks, which have a continuous data layout in flash memory. Then, DEFTL assigns the head blocks for the public data and the tail blocks for the hidden data. When $b = 1$, $\mathcal{O}_{1,1}$ might contain accesses $\mathcal{O}_h$ including hidden writes. However, $\mathcal{O}_h$ can overwrite the hidden data. When the free space is insufficient, DEFTL triggers GC to reclaim invalidated flash pages, which are placed in the head of the flash block pool. Assume $\mathcal{O}_{1,1}$ contain accesses $< \mathcal{O}_p, \mathcal{O}_h, \mathcal{O}'_p >$, $\mathcal{O}_h$ incurs GC, and the reclaimed invalidated hidden blocks are inserted into the head of flash block pool. Then, $\mathcal{O}'_p$ can generate an inexplicable data layout. Some written writes might be serviced by the flash blocks that were reclaimed from the hidden data, while other public data are assigned with the original free blocks. Thus, it generates a <public, random, public> data layout that is not consistent with the <public, random> data layout of $\mathcal{O}_{1,1}$.

Based on Definition 1, these methods lose the PD game as they have discernible access patterns in $\mathcal{O}_{0,1}$ and $\mathcal{O}_{1,1}$. □

A.2. Compromising Multi-snapshot PD

Current flash-based PD systems [30], [58], [29], [45] designed against multi-snapshot attacks rely on special encoding or dummy write methods to hide sensitive data. We verify their PD guarantees via the following proof.

Theorem 3. *For current multi-snapshot PD systems in flash memory, access sets $\mathcal{O}_{0,1}$ and $\mathcal{O}_{1,1}$ can generate distinguishable data layouts, breaking their PD guarantees.*

Proof. Considering multi-snapshot adversaries, the PD game runs for r rounds. When $b = 0$, the access set $\mathcal{O}_{0,k}$ in round k ($1 \leq k < r$) operates public data in $\mathcal{V}_p$. When $b = 1$, $\mathcal{O}_{1,k}$ is performed on both $\mathcal{V}_p$ and $\mathcal{V}_h$. Both INFUSE [30] and PEARL [29] encode the hidden data within the public data. However, the adversary $\mathcal{A}$ can compromise them by leveraging the semantic-aware layout exploitation to make $\mathcal{O}_{1,k}$ distinguishable from $\mathcal{O}_{0,k}$. The hidden data written alters the storage state of the original public data on flash memory. However, it might select the cold public data, which are associated with cold files and are unlikely to be modified. Reprogramming cold public data for hidden written raises suspicious data layout, as they should never be invalidated or updated. Thus, encoding-based methods cannot ensure PD guarantees.

MDEFTL [45] and FSPDE [58] hide the data via dummy writes. In round k, $\mathcal{O}_{1,k}$ writes both public and hidden data. In round $k + 1$, $\mathcal{O}_{1,k+1}$ contains accesses $\mathcal{O}_p$ to public data, which trigger GC. However, during GC, a flash block containing valid hidden data may be selected, requiring the relocation of it to prevent its erasure. Thus, multi-snapshot adversaries can identify the relocation of "random data" (hidden data). Since random data are generated as dummy writes reclaimed during GC, the relocation of random data leads to observable data layout changes of $\mathcal{O}_{1,k+1}$. Thus, they cannot ensure plausible deniability. □

A.3. Analyzing other PD Systems

Analysis of INVISILINE. INVISILINE is designed for HDD-based storage device without considering the unique characteristics of flash memory. INVISILINE requires users to write sufficient public data to the storage device before storing hidden data, then reads the public data, encrypts it using a tweak value (TV) derived from the hidden data, and overwrites the original public data with the encrypted version. However, since flash memory uses out-of-place writes, the original public data remains accessible. An adversary can thus detect duplicated public data by examining the raw flash contents and undermine INVISILINE's PD guarantees.

INVISILINE modified the encryption module of the dm-crypt in the OS by correlating each encryption unit to its encryption metadata, which is represented by the hidden data. Given that the size of this encryption metadata is 16 bytes [68], each encryption unit can be associated with 16 bytes of hidden data. In the OS, the encryption unit matches the size of the OS-level data block, which is 512 bytes, whereas a data unit typically aligns with the page size in flash memory. When integrating INVISILINE into flash memory, it can only provide a hidden capacity of 2 GB or 0.5 GB if the page size is 4 KB or 16 KB, respectively. This hidden capacity is insufficient for the hidden data and

4. Distinguishable evidence indicates that it contains a data layout generated by $\mathcal{O}_{1,1}$ that cannot be explained by operations in $\mathcal{O}_{0,1}$. Thus, $|Pr(b' = b) - Pr(b' \neq b)| >$ negl(s).

TABLE 2: Parameters of the implemented SSD.

Parameters	Value	Parameters	Value
SSD Capacity	512GB	Channels	4
Page Size	4KB	OOB Area	409B
Pages/Block	256	OOB Read	0.02ms
Blocks/Plane	16384	Page Read	0.04ms
Planes/Chip	1	Page Write	0.2ms
Chips/Channel	8	Block Erase	2ms

TABLE 3: The characteristics of MSR workloads.

Workload	Write Ratio	Avg. Req. Size
hm	73.7%	8.3KB
prxy	96.9%	2.5KB
rsrch	90.7%	8.7KB
wdev	79.9%	9.4KB

is significantly smaller than that of current PD systems [67], [44], [29], [45]. In contrast, MUTE can provide 25.3 GB or 33.9 GB of hidden capacity, outperforming INVISILINE by 12.7x or 67.8x, under the same conditions.

Applying DL Attacks to Other PD Systems. In Section 4, we only analyzed the PD systems specifically designed for flash memory [44], [58], [67], [29], [45], [30] because they introduce distinct data processing patterns and layouts tailored for flash memory. Other PD methods [27], [28], [32], [36], [74], [77] either use flash memory as a standard block device – ignoring its unique physical properties – or do not offer new approaches to hidden data storage beyond those already analyzed. Consequently, these methods also remain susceptible to the DL attack.

A.4. MUTE Depolyment Considerations

Password selection. The public and hidden keys are derived from the public and hidden passwords. The compromise of passwords can lead to the failure of the PD guarantee as they can be used to uncover hidden data. Thus, we assume the use of robust passwords, such as randomly-generated high-entropy passwords [48] and security keys [8].

Firmware integration. Integrating MUTE into the firmware of current SSDs is practical. With a bandwidth degradation of only 0.9% at maximum in public mode, the degradation is minimal and will not hinder its adoption. Moreover, MUTE can be incorporated into current SSDs through firmware updates, which is standard practice for maintaining modern SSDs [20], [19].

Appendix B. Implementation and Evaluation Details

B.1. SSD Parameters and MSR Traces

The detailed SSD parameters are listed in Table 2. Moreover, we show the details of evaluated MSR traces in Table 3.

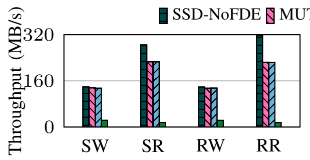


Figure 9: The bandwidth of SSD-NoFDE and MUTE.

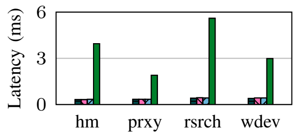


Figure 10: The average latency of SSD-NoFDE and MUTE.

B.2. Implementation Details of PEARL and MDEFTL

We re-implemented PEARL [29] and MDEFTL [45] in FEMU using the same SSD parameters as MUTE, as described in Table 2. Since their source code is not publicly available, we followed the designs outlined in their papers.

PEARL Implementation. We adopt WOM (3,5), encoding 3 bits of original data into 5 bits of WOM codewords, consistent with the PEARL's implementation. Then, we implement WOM functions and embed them into flash memory operations, similar to prior WOM works in flash-based SSDs [43]. In public mode, I/O requests are handled normally as operated by regular WOM SSDs. To write hidden data, our implementation forces GC to migrate valid public data while cloaking the hidden data within the migrated public data, consistent with PEARL's approach. For hidden read operations, each 5-bit read data is decoded using the WOM codeword mapping to retrieve the hidden data.

MDEFTL Implementation. We implement a DW-1 setting in our MDEFTL implementation, which performs one dummy write for each write request. This setting ensures the best performance of MDEFTL as described in its paper for a fair comparison. In addition, MDEFTL adopts a random block allocation method, by randomly selecting a free flash block for the incoming data instead of the traditional page-level channel-first allocation used in FEMU. Thus, we implement the page allocation used by MDEFTL into FEMU to make it align with MDEFTL's implementation. When writing public data, a dummy write is attached to each write request, whereas hidden writes proceed without dummy writes, consistent with MDEFTL's design.

Appendix C. Additional Experimental Results

C.1. Performance Comparison of SSD-NoFDE and MUTE

We also evaluate the performance of the baseline SSD without equipping FDE (SSD-NoFDE). In public mode, Figure 9 indicates that MUTE-PO and MUTE-PH introduce bandwidth degradation over SSD-NoFDE by 18.2% on average. Additionally, Figure 10 shows that MUTE-PO and MUTE-PH increase the average latency over SSD-NoFDE by 4.1% and 4.8%, respectively. The root cause of this performance degradation is that the encryption operations

TABLE 4: Symbols

Symbol	Meaning	Where
$\mathcal{V}_p$	Public volumes	Section 3.3
$\mathcal{V}_h$	Hidden volumes	Section 3.3
$\mathcal{P}_p$	Password for public data	Section 3.3
$\mathcal{P}_h$	Password for hidden data	Section 3.3
$\mathcal{N}$	Total number of pages in storage device	Section 3.3
$\cup_{i=1}^{\mathcal{N}}\{pg_i\}$	A set of physical flash pages	Section 3.3
add_i	Physical address	Section 3.3
$\cup_{i=1}^{\mathcal{N}}\{add_i\}$	A set of physical addresses	Section 3.3
$\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$	A set of files	Section 3.3
$\mathcal{B}$	Logical data block	Section 3.3
$\mathcal{A}$	Adversary	Section 3.3
$\mathcal{C}$	Challenger	Section 3.3
$\mathcal{O} =< O_1, O_2, \ldots, O_n >$	A sequence of write accesses	Section 3.3
b	Random bit	Section 3.3
negl(s)	Negligible function	Section 3.3
$Pr()$	Probability	Section 3.3
s	Security parameter	Section 3.3
$\mathcal{O}_h$	Accesses the hidden data	Section A.1
$\mathcal{O}_p$	Accesses the public data	Section A.1
$S = \{0, 1, \ldots, n-1\}$	Integer set of n samples	Section 5.1
π	Permutation	Section 5.1
$rank()$	Rank of permutation	Section 5.1
v	Numeric ranking value	Section 5.1
N_{pg}	Number of public data pages	Section 5.2
N_h	Number of hidden data pages	Section 5.2
PG	Public data page	Section 5.2
P	Page size	Section 8.3
C	Total capacity of device	Section 8.3
b	Encryption block size	Section 8.3
M	Metadata size of a hidden batch	Section 8.3
q	Number of encryption blocks per page	Section 8.3

of FDE in MUTE are located in the I/O critical path whereas SSD-NoFDE does not include FDE. In hidden mode, MUTE-H decreases 91.1% bandwidth over SSD-FDE and introduces 9.8x of the SSD-FDE latency. This degradation is mainly due to the GC operations triggered by MUTE-H when writing hidden data, as discussed in Section 8.2.

C.2. Hidden Read Performance Degradation

MUTE exhibits lower read bandwidth for hidden data compared to PEARL and MDEFTL. However, this does not impede the normal operation of the PD system in real-world scenarios. As discussed in Section 3.2, users access their hidden volumes within secure environments without time stress rather than in high-risk situations. Therefore, the reduced read bandwidth is acceptable, as users have sufficient time to manage their hidden data.

Current PD research prioritizes robust plausible deniability guarantees over high I/O performance. For example, PEARL provides only 10%–20% of a regular SSD's bandwidth for hidden data, and HIVE achieves just 0.99 MB/s read bandwidth, reducing the storage device's throughput by over 99%. Despite these limitations, such methods are still adopted as PD is a special use case that emphasizes security over performance. Since MUTE achieves HDD-level read throughput (discussed in Section 8.2), it offers practical performance for hidden data operations.

C.3. Capacity Mathematical Analysis Supplement

The capacity of hidden data in a flash page (in bits) is based on permutations of encryption blocks, which in our case is $\lfloor \log_2(q!) \rfloor - M$. Then, the capacity of the hidden volume (in bits) $\mathcal{V}_h$ is:

$$\mathcal{V}_h = \frac{C}{P} \cdot (\lfloor \log_2(q!) \rfloor - M)$$

Since the hidden volume ($\mathcal{V}_h$) depends on the factorial of the number of encryption blocks per page ($q = \frac{P}{b}$), we determine its lower bound using the Stirling formula [72]:

$$\mathcal{V}_h \geq \frac{C}{P} \cdot \left(\left\lfloor \log_2(e \cdot (\frac{q}{e})^q) \right\rfloor - M \right) \tag{2}$$

According to Stirling's formula, we have:

$$\mathcal{V}_h \geq \frac{C}{P} \cdot \left(\left\lfloor \log_2(e \cdot (\frac{q}{e})^q) \right\rfloor - M \right)$$

Then simplify it, we can get:

$$\begin{aligned} \mathcal{V}_h \geq \frac{C}{P} \cdot \left(\left\lfloor \log_2(e) + q \log_2(\frac{q}{e}) \right\rfloor - M \right) > \\ \frac{C}{P} \cdot (1 + q(\log_2(q) - 2) - M) > \\ \frac{C}{P} \cdot (q(\log_2(q) - 2) - M) \geq \\ \frac{C}{b} \cdot \left(\log_2(q) - (2 + \frac{M}{q}) \right) \end{aligned}$$

Then we can get Equation 1:

$$\mathcal{V}_h > \frac{C}{b} \cdot (\log_2(q) - \eta)$$

where $\eta = 2 + \frac{M}{q}$ which approaches to 2 when q grows.

Appendix D. Symbol Table

We summarize the symbols we used in Table 4.