공학석사학위논문

A Design and Implementation of SSDs with Strong Plausible Deniability

2022년 8월

서울대학교 대학원

컴퓨터공학부

조 건 희

A Design and Implementation of SSDs with Strong Plausible Deniability

지도교수 김 지 홍

이 논문을 공학석사 학위논문으로 제출함

2022년 8월

서울대학교 대학원

컴퓨터공학부

조 건 희

조건희의 공학석사 학위 논문을 인준함

2022년 8월

위 원 장: 하 순 회 (인)

부위원장: 김 지 홍 (인)

위 원: 이 창 건 (인)

Abstract

While encryption can hide the contents of private data, it cannot hide the existence of encrypted data. Thus, privacy is compromised from the adversary who coerces the user to disclose the decryption key. To defend against such a coercive adversary, plausible deniability (PD) solution, which supports data hiding have been proposed. To enhance plausibility, it is important to support *strong PD*, which makes it possible to further deny the use of data hiding technique itself. Unfortunately, existing solutions supporting strong PD are not practical to be implemented in modern flash-based storage system due to their reliability penalty. In this paper, we propose a new access-centric data hiding mechanism, Fidelius, which is practical in high-density 3D NAND flash memory while supporting strong PD. By exploiting the on-chip resources, Fidelius supports flash lock/unlock commands (`bHide` and `bExpose`) that can re-configure the block-level data access permission. Because the adversary cannot inspect the data insides locked block, Fidelius can deny the existence of private data by simply hiding it in a locked block. Since the existence of locked blocks can be explained not as a result of data hiding, but as a result of data sanitization, which control the accesses to stale data, Fidelius can supports strong PD. To evaluate the proposed technique, we built SafeSSD, an Fidelius-enabled emulated flash storage system that supports strong PD with multiple-snapshot resistance. Our experimental results show that SafeSSD can support strong PD with a negligible performance overhead and no reliability penalty.

keywords: solid-state drives (SSDs), 3D NAND flash memory, security, privacy, plausible deniability, deniable storage

student number: 2019-21844

Contents

Abstract **i**

Contents **ii**

List of Tables **iv**

List of Figures **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 4

1.3 Thesis Structure 6

2 Background **8**

2.1 Flash Memory Overview 8

2.1.1 Organization of Flash Memory 9

2.1.2 Flash Operations 9

2.1.3 Multi-level Cell Technology 10

2.2 Flash Translation Layer 11

2.2.1 Address Translation 11

2.2.2 Garbage Collection 12

2.2.3 Data Sanitization 12

3 Limitations of Existing PD Solutions **13**

3.1 Lack of Deniability for Data-Hiding Mechanism 13

3.2 Lack of Practicality in High-Density Flash Memory 15

4 Threat Model **18**

4.1 The Capability of Adversary 18

4.2 Assumptions 19

5 Access-Centric Data Hiding Technique **20**

5.1 Approach Overview 20

5.2 Flash Commands for Block Access Control 23

5.2.1 Organizational Overview 23

5.2.2 Implementation 24

6 SafeSSD: Fidelius-based PD Solution **28**

6.1 Overview 28

6.2 Fidelius-Aware FTL 29

6.3 User Interfaces 30

7 Evaluation Results **32**

7.1 Experimental Settings 32

7.2 Performance Evaluation 33

7.3 Usability Evaluation 34

8 Related Work **36**

9 Conclusions **38**

Bibliography **39**

Abstract (In Korean) **48**

List of Tables

1 User-level steps τ_{exp} and τ_{hid}. 30

2 A summary of I/O characteristics of three traces. 32

List of Figures

1 An overview of a NAND flash block. 9

2 V_{th} distributions of 2^m-state NAND flash memory. 11

3 Operational overview of `bHide` and `bExpose`. 22

4 An overview of the Fidelius implementation. 23

5 Device-level experimental results for Fidelius 26

6 An organizational overview of SafeSSD 29

7 Performance of SafeSSD under three different workloads. 33

8 Operation latency of SafeSSD interfaces. 34

Chapter 1

Introduction

1.1 Motivation

As various security-sensitive data are commonly stored to mobile computing devices (e.g., smartphones, tablets, and laptops), ensuring data security and privacy in such systems is one of the most critical design requirements. In order to manage sensitive data in a secure and private fashion, these systems need to support a strong data-protection capability so that sensitive data can be properly concealed from unauthorized accesses. For example, data encryption techniques [1–3] are widely used to protect important private data. Although encryption-based schemes are generally effective in avoiding unauthorized data disclosure, they protect sensitive data with a clear hint that a storage owner *may* hide some data. Such a hint, however, may lead to unwanted disclosure of sensitive data when the storage owner is forced to reveal an encryption key by coercive adversaries.

In order to hide data in an SSD without leaving any trace on data hiding itself, in this paper, we investigate how to build SSDs with *plausible deniability (PD)*. Informally, an SSD S is considered to support PD under a threat model T_S if an owner can claim that no data are hidden in the SSD S when data are actually hidden while an adversary under the threat model T_S cannot contradict this claim. When data are en-

crypted, for example, the adversary can easily disprove the claim of the storage owner that no data is hidden in the SSD by decrypting the encrypted sensitive data. Therefore, an SSD with data encryption does not support PD when the adversary is allowed to acquire the secret key under the threat model.

Most early PD solutions [4–6] for supporting PD have focused on hiding the *existence* of encrypted sensitive data. For example, the encrypted sensitive data were stored in the free space region of an SSD that was pre-filled with randomized dummy data. Since the adversary cannot distinguish the encrypted sensitive data from random dummy data of the free space region, the adversary cannot contradict the claim of the storage owner.

Although those solutions are successful in hiding the existence of *encrypted* sensitive data at a single point of time, they fail to conceal write operations to the encrypted sensitive data if an adversary is allowed to take multiple snapshots of an SSD. For example, even if the adversary could not distinguish encrypted sensitive data from dummy data in the snapshot S_0 captured at time T_0, the adversary can detect some questionable activities by comparing S_0 with the snapshot S_1 at time T_1 (i.e., $T_1 > T_0$) if the sensitive data were modified within the interval $[T_0, T_1]$. In order to solve the multi-snapshot problem, recent solutions [7–12] have proposed various schemes that hide indications of write operations to encrypted sensitive data. A basic approach is to generate extra dummy writes to the free space region so that an adversary cannot identify questionable write operations to the free space region by comparing multiple snapshots.

In order to better understand the pros and cons of existing PD solutions, we classify an SSD with PD support in two categories. When an SSD S is supported by a PD mechanism M_S, we call that the SSD S supports *strong PD*[1] if S can *plausibly deny* the existence of M_S as well as the existence of hidden data. When the SSD S cannot plausibly deny the existence of M_S, S is defined to support *weak PD*. For example,

[1]Strong PD systems are called as invisible PD systems in [13].

most existing SW-centric techniques (such as ECD [11] or MDEFTL [12]) support weak PD because they cannot plausibly deny the existence of its mechanism M_S for supporting PD. For example, it will be difficult to persuade an adversary why seemingly random writes to the free space region are required for a normal SSD. Like the existence of encrypted data in an SSD, the existence of a special PD mechanism (that cannot be plausibly explained) can work as a strong hint to an adversary that some data may be hidden. Therefore, it is essential to support strong PD under a powerful adversary.

In this paper, our goal is to devise a PD mechanism M that can support strong PD in modern high-density SSDs under a powerful threat model. (For a detailed threat model, see Section 4.) Existing strong PD solutions (INFUSE [14] and PEARL [15]), unfortunately, are difficult to put into practice in modern high-density SSDs. The key data hiding techniques used in both INFUSE and PEARL are not applicable in modern high-density SSDs. For example, a voltage-based data-hiding scheme [16] used in INFUSE [14], which relies on the reprogram operation that adjusts the voltage level of the flash cells that were previously programmed, is very difficult to be reliably used in modern high-density flash memory (e.g., TLC NAND flash memory) due to program disturbance [17–19]. Likewise, WOM codes [20] that PEARL [15] depends on also requires reprogram operations, so it shares the same limitation as INFUSE.

In order to support strong PD in modern high-density SSDs, we argue that data hiding should not depend on either data conversions based on secret information (e.g., data encryption with a secret key) or data concealment based on data embedding within other data (e.g., reprogram-based techniques at the flash cell level). Instead, data hiding should focus on disabling accesses to sensitive hidden data. If such access control can be supported by plausibly deniable interface functions, strong PD can be supported without causing data reliability problems in modern high-density flash memory while leaving no hint on the existence of a PD mechanism.

1.2 Contribution

In this paper, we propose Fidelius[2], a new data hiding mechanism for modern SSDs that achieves strong PD by employing an access-centric data hiding approach. Fidelius hides sensitive data by disabling unauthorized access to the sensitive data using on-chip access control. Since access control is managed within a flash chip, hidden data is not accessible even with a very powerful adversary. Fidelius is based on a special flash command, `bLock`, that was proposed to block unauthorized access to deleted files in an SSD [22]. The `bLock` command is used to securely erase (i.e., sanitize) sensitive private data in high-density 3D NAND flash memory. When a block is sanitized by `bLock`, a read request to the sanitized block always returns zero-filled data (i.e., all bits are "0").

In Fidelius, we use a simple variant of `bLock`, which we call `bHide`, that explicitly associates a secret password to a sanitized block. Once the block is locked by `bHide` with a password p, a read request to the block returns a zero-filled page (as with `bLock`). The locked block can be read normally again only after the block was successfully *unlocked* by the `bExpose` command, which is a new flash command proposed in Fidelius. The locked block can be unlocked by `bExpose` if `bExpose` can verify the password p that was used by `bHide` when the block is locked. When data in a block B_s need to be hidden, the block B_s is locked by `bHide` with a secret password. As long as the secret password is unknown to an adversary, data in the block B_s is safely hidden because a read request to B_s will return a zero-filled page, not a hidden page in B_s. The hidden data in B_s can be accessed again only after `bExpose` verifies the secret password of B_s.

In order to plausibly deny data hiding in Fidelius, locking blocks by `bHide` should not leave any suspicious trace to an adversary. To satisfy this requirement, we assume that a target SSD is based on modern 3D flash NAND memory which requires a de-

[2]Fidelius, which means *more* trustworthy/faithful in Latin, comes from Harry Potter's Fidelius charm that is used to conceal a secret [21].

layed erasure scheme for meeting its data reliability requirement.[3] Under the lazy erasure scheme, a block is not erased when it is selected as a victim block of garbage collection (GC) but its erasure is delayed until data are programmed on the block. Since a block should be lazily erased, the block may be exposed to unauthorized access while its erasure is delayed. To avoid such a security loophole, when a block is selected as a victim of GC, it should be locked by `bHide` so that its data become immediately inaccessible although its erasure is postponed. In fact, all (future) free blocks are maintained as locked blocks by `bHide` because they should be erased right before they are used for programming data. When sensitive data are hidden by `bHide`, therefore, it is not possible to distinguish whether blocks are locked for hiding data or blocks are locked before they are erased. Since a block to hide sensitive data is taken from locked blocks and the block is locked again after sensitive data are programmed, no suspicious trace can be found even when multiple snapshots of an SSD are inspected.

For Fidelius to support strong PD, we need to explain why the `bHide`/`bExpose` commands are used in an SSD. Although the existence of `bHide` can be plausibly explained for supporting data sanitization in SSD, the existence of `bExpose` might be considered suspicious to an adversary because it is not an essential command for data sanitization. In a strong PD scenario, however, we do not admit the existence of `bExpose`. Rather, we claim that our SSD supports data sanitization. Even if an adversary tries brute-force attack to find out the hidden command that potentially support data hiding capability, the result of arbitrary command execution cannot indicate any data hiding. For example, even if an adversary executes `bExpose` by chance during a brute-force attack, when a wrong password is used, following read results will not change.

To demonstrate the effectiveness of Fidelius, we built SafeSSD, a Fidelius-aware

[3] Due to structural characteristics of modern 3D NAND flash memory, the longer the interval between the time the block is erased and the time the data is programmed to the block, the worse the reliability of data stored in the block [23, 24].

SSD emulator that supports strong PD (with multiple-snapshot resistance) using `bHide` and `bExpose`. SafeSSD supports two logical volumes, the public volume and private volume. The private volume, which exists within the locked blocks, is revealed by unlocking all the locked blocks by using a custom command through the existing NVMe admin interface [25]. Using metadata of a file system (e.g., super block) that were hidden in the locked blocks, the file system is mounted on the private volume and hidden files can be accessed using standard file interface functions (e.g., read() and write() system calls). Note that when the private volume needs to be hidden again, the file system is unmounted while saving metadata of the file system to the locked blocks.

We evaluated SafeSSD using various workloads collected from enterprise server. Our experimental results show that SafeSSD supports strong PD with negligible performance overhead over a normal SSD. In our evaluation, the IOPS degradation of SafeSSD was at most 0.35% over common SSDs with no data sanitization. To understand the usability of SafeSSD, we measured the private volume exposing time, which is the time spent for unlocking blocks and mounting an ext4 file system, and private volume hiding time, which is the time spent for unmounting the file system and re-locking blocks. On a PC with 2.1-GHz CPU and 8-GB memory, when a 6.4-GB private volume was supported out of a 128-GB SafeSSD, it took about less than 0.3 seconds for private volume exposing and less than 0.2 seconds for private volume hiding.

1.3 Thesis Structure

The rest of this paper is organized as follows. In Section 2, we review flash-based storage systems. Section 3 briefly introduce the existing PD solutions and reports their limitations. Before describing Fidelius, the threat model is presented in Section 4. We describe the proposed Fidelius with its new flash commands, `bHide` and `bExpose`, in Section 5. An overview of SafeSSD including its FTL and user interfaces is presented

in Section 6. Evaluation results follow in Section 7. Related work is summarized in Section 8, and Section 9 concludes with a summary.

Chapter 2

Background

2.1 Flash Memory Overview

The NAND flash chip is organized by multiple flash cells and peripheral circuits. The flash cells are used to store data and the peripheral circuits are responsible for supporting flash operations. Flash cells that are placed on the same wordline (WL) constitute a flash page and a flash block is a unit composed with hundreds of flash pages (e.g., 768 pages [26]). A typical organization of the flash block is illustrated in the Figure 1. As shown in the Figure 1, m flash cells in each row (e.g., 8K or 16K) are grouped to form a single WL and there are n WLs in a block. In addition, a bitline (BL) is a unit that is formed with n flash cells in each column and the BLs are shared by all flash blocks in a flash chip. Each flash block has two select transistors, source select line (SSL) and ground select line (GSL). We can determine the target block for each flash operation (among flash blocks existing in a flash chip) by applying selecting voltages on the SSL and GSL. BLs are connected to the page buffer, which is used to store data temporarily before transferring data to off-chip through the data-in/out circuit.

As illustrated in the Figure 1, the floating gate of each flash cell traps electrons to represent the threshold voltage (V_{th}) states. In order to adjust the V_{th}, electrons are injected or ejected from the floating gate of the flash cell. Under a given control gate

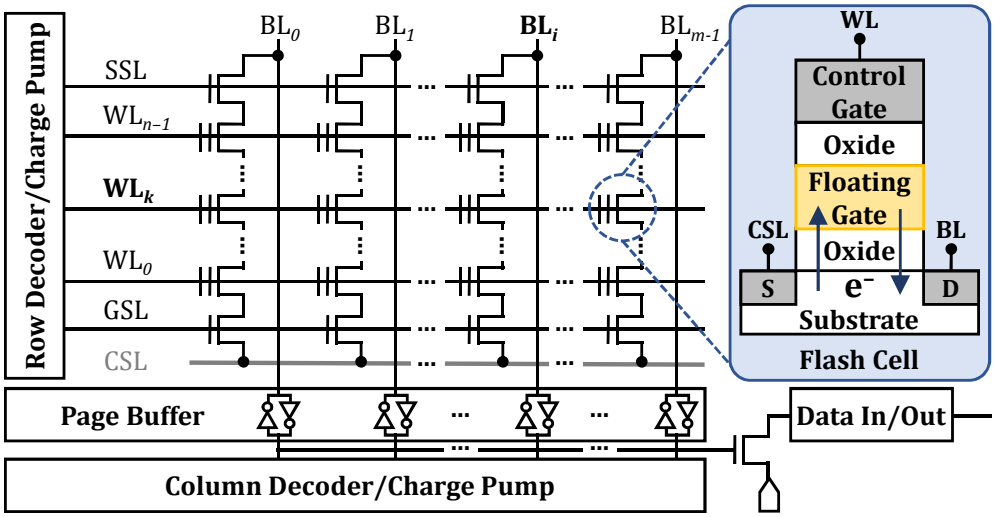


Figure 1: An overview of a NAND flash block.

voltage, the flash cell acts as an off-switch or an on-switch depending on the amount of electrons in the floating gate. For example, we can store '0' bit when the flash cell has a high V_{th} (i.e., off state), and '1' bit when the flash cell has a low V_{th} (i.e., on state).

2.1.1 Organization of Flash Memory

2.1.2 Flash Operations

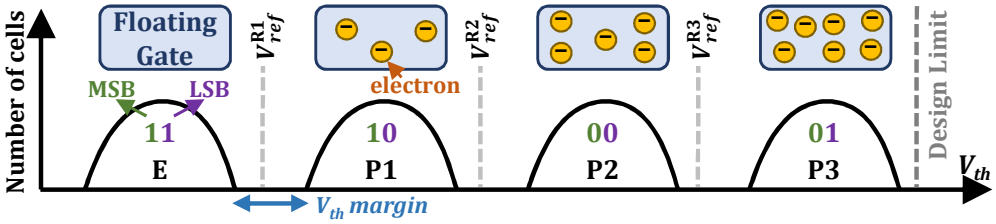
During a program operation, high voltage ($>$ 20V) is applied to control gates to cause FN tunneling [27]. As a result of FN tunneling, the electrons are transferred from the substrate to the floating gates of the selected flash cells. As electrons are injected and captured into the floating gate, V_{th} of the flash cells increases. As flash cells on the same WL (i.e., a page) share the same signal from the row decoder module, they are programmed (or read) together as a unit. Since the program operation can only increase V_{th} of flash cells, to program new data on the page, all the flash cells of a page should be erased (i.e., erase-before-program). In order to erase programmed cells, a high voltage ($>$ 20V) is applied to the substrate (while control gates are set

to 0V) to transfer electrons from floating gates to the substrate. Then, the V_{th} of the flash cell returns to the lowest level, and it becomes a state that can be programmed again later. Since the whole block shares the entire substrate, all the cells constituting a block are erased together.

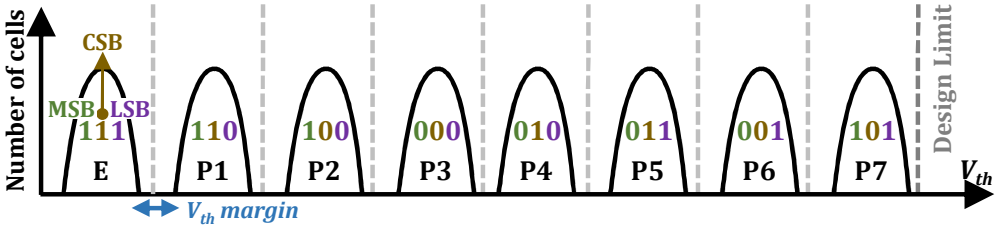
The read operation probe the V_{th} level of the flash cells on the selected WL by using a read reference voltage V_{ref}. If V_{th} of the i-th flash cell in WL_k is higher than V_{ref} (signal from the row decoder module), the i-th flash cell act as an off-switch, then the cell current of BL_i is blocked. On the other hand, if the V_{th} of the i-th flash cell is lower than V_{ref}, the i-th flash cell acts as an on-switch, then the cell current can flow through BL_i. By sensing the current of BLs, the stored data (in WL_k) can be identified and read out to the page buffer. Note that if we can disconnect the data transfer path (between the page buffer and data-in/out circuit), even if the data can be staged into the page buffer, the data transfer to the off-chip can be disabled. On the other hand, if we can re-connect the data transfer path, it will be possible to control (i.e., block or unblock) the accesses to data in the block *intentionally*.

2.1.3 Multi-level Cell Technology

A multi-leveling techniques [28, 29], that stores multiple bits in one flash cell, have been developed to implement a large capacity flash memory in cost-effective way. Multi-level cell (MLC) technology is extended to triple-level cell (TLC) which supports storing 3 bits per cell, and even further developed to quad-level cell (QLC) that supports storing 4 bits per cell. Figure 2 shows the V_{th} distributions of flash cells for 2^m-state NAND flash memory, which stores m bits within a single flash cell (i.e., m is 2 and 3 for MLC and TLC, respectively). As m increases, more V_{th} states should be put into the limited V_{th} window. In consequences as shown in the Figure 2(a) and 2(b), each V_{th} state should become finer and a V_{th} margin, a gap between two neighboring V_{th} states, becomes narrower. Due to the smaller V_{th} margin, various noise conditions can result the two neighboring V_{th} states to be more likely to be overlapped



(a) m=2 : MLC flash memory.



(b) m=3 : TLC flash memory.

Figure 2: V_{th} distributions of 2^m-state NAND flash memory.

which results degraded flash reliability. For example, the number of program and erase (P/E) cycles of the TLC flash memory is only allowed for about 1,000 counts whereas MLC flash memory can endure until 3,000 P/E cycles. Due to the degraded reliability, various effective optimization strategies that were introduced for smaller m's can no longer be effective or even cannot be applied as m increases.

2.2 Flash Translation Layer

2.2.1 Address Translation

To hide unique characteristics of flash memory and emulate it as a traditional block-based storage device (e.g., HDDs), a flash translation layer (FTL), is adopted to flash-based storage systems. Owing to the erase-before-program characteristics of the flash memory, FTL uses out-of-place update mechanism which requires a logical-to-physical (L2P) address mapping table, that translates from a logical sector address (of the host system) to a physical page address (of the flash memory).

2.2.2 Garbage Collection

Out-of-place update mechanism results in lots of invalid pages, which contain the stale data (i.e., updated or deleted). When invalid pages consume more than a certain amount of space, the FTL triggers a a garbage collection (GC) process to prepare some free blocks for future host write. During GC process, FTL selects a victim block, then invalidates all the valid pages in the victim block after copying them to another block. The related L2P mapping is also changed properly. The invalid block, which consists of only invalid pages, are maintained in the free block pool until they are erased. When there are no free pages to service host writes, FTL erases the block taken from the free block pool.

2.2.3 Data Sanitization

Since the erasure of the block is triggered in an on-demand manner, there is a time gap between when the victim block is logically invalidated and when the block is physically erased. As a result, there are always some amount of invalid blocks (that are not yet erased) are maintained in the free block pool. However, remained invalid pages provide an opportunity for data recovery, which can be maliciously performed. To address such a security loophole in a flash-based storage system, studies on data sanitization (i.e., make deleted/stale data irrecoverable) have been actively conducted [22, 30–38].

Chapter 3

Limitations of Existing PD Solutions

In this section, we briefly review existing PD solutions focusing on their key ideas and limitations.

3.1 Lack of Deniability for Data-Hiding Mechanism

In general, a PD solution aims to hide the existence of hidden data from adversary by making the system state (or behavior) with hidden data indistinguishable from usual ones without any hidden data. There are many possible ways to achieve this goal, such as leveraging ciphertext indistinguishability [39], exploiting device-specific properties (e.g., threshold-voltage level of a flash cell [16], aging characteristics of SRAM cells [40]), and using special encoding scheme (e.g., a write-once memory (WOM) code re-purposed to hide data [15]).

While existing PD solutions target various storage media (e.g., SRAM, magnetic disk, and flash memory), many of them are designed to target NAND flash-based storage systems to enjoy their ubiquity and high capacity. Considering the limited resources of flash-memory controllers or mobile devices, most are based on ciphertext indistinguishability which has low implementation complexity and minimum computational burden. For example, one of the flash-based PD solutions exploiting cipher-

text indistinguishability, DEFTL [41] hides encrypted private data among the random noise (e.g., data generated by a cryptographically secure pseudorandom number generator [42]) pre-filled on the disk's free space region. Since the ciphertext is indistinguishable from random noise, the adversary cannot tell if hidden data even exist, so the device owner can plausibly deny the existence of the hidden data.

However, as such solutions cannot preserve PD if the adversary can examine the device at multiple different time points. Since hidden data writes leave unexplainable changes on random noise, which is expected to remain static over time, the adversary can detect indications of hidden data writes by comparing disk contents (i.e., snapshots) before and after hidden data writes. To preserve PD against such a multiple-snapshot adversary, for example, an advanced version of DEFTL, called MDEFTL [12], occasionally (e.g., every minute) performs extra dummy writes to obfuscate the hidden data writes. Thus, the device owner can attribute the questionable changes in random noise to part of normal system behavior, making the adversary cannot tell such changes are the results of hidden data writes.

Unfortunately, they result in suspicious system states or behaviors that are difficult to plausibly justify, making the adversary suspect that the device owner has an intention to hide some data. For example, random noise needed to hide encrypted private data consumes considerable amount of disk space, but there is no plausible reason for random noise to be existed if the storage system is really normal one (with no data-hiding mechanism). Moreover, the random noise changes observed across multiple snapshot also makes a clear difference from normal storage systems (whether that changes are due to dummy data writes or hidden data writes). Such differences makes the adversary suspicious of the use of the special mechanism which potentially allows the user to hide data. Even if the attacker cannot prove the existence of hidden data through the disk snapshot investigation, the exposure of the existence of data-hiding mechanism can lead the device owner to be placed in a dangerous situation (e.g., rubberhose attack).

To mitigate this problem, it is highly desirable to be able to plausibly deny not only the existence of hidden data, but also the *existence of the data-hiding mechanism*. In other words, even if the data-hiding mechanism is combined, system states (or behaviors) should be not different from that of the normal storage system with no data-hiding mechanism. We will refer to such a data-hiding mechanism, which can also deny the use of data-hiding mechanism itself, as a *strong PD* mechanism. Contrariwise, a data-hiding mechanism which lacks of PD for data-hiding mechanism itself will be referred to as a *weak PD* mechanism.

3.2 Lack of Practicality in High-Density Flash Memory

To our knowledge, there are two promising PD solutions, which have potential to provide strong PD. We will review their key data hiding mechanisms and discuss their limitations in this subsection. The first solution, called INFUSE [14], uses a voltage-based data-hiding technique called VT-HI [16] as a building block to provide strong PD with multiple-snapshot resistance. VT-HI leverages threshold voltage side channel to seamlessly hide private bits in flash cells. The key idea of VT-HI is to hide private bits by transparently increasing the densities (i.e., the number of bits stored inside) of already programmed flash cells without leaving detectable deviation in the overall V_{th} distribution. For example, to hide a private bit '0' on a single-level cell, VT-HI slightly increase the V_{th} of the target flash cell (while preserving the public bit value). On the other hand, to hide a private bit '1', leave the V_{th} of the target flash cell as it is. The private bits can be extracted later using MLC-style read reference voltage by the user with the secret key (which determines the locations of hidden private bits). As there is an enough variation in the range of V_{th} in flash cells, the V_{th} changes of some flash cells can be obscured, the adversary cannot distinguish the storage system (with some private bits hidden by VT-HI) from the typical SLC flash-based storage system.

The second solution, called PEARL [15], exploits properties of write-once mem-

ory (WOM) codes [20]. PEARL designs an WOM code (tailored for data hiding) that makes a string of public bits with hidden private bits indistinguishable from a string of only public bits. Like other WOM codes, PEARL reuses the invalidated flash page (which is already programmed with the 1st codeword) by overwriting (i.e., re-programming) it with the 2nd codeword. The difference is PEARL has two (rather than one) 2nd codewords choices and it chooses the 2nd codeword according to the value of the private bit to hide. For example, in PEARL, '11001' or '10110' can be used as the 2nd codeword for representing the logical public bits '001'. Suppose we need to re-program the flash cells of the target invalidated page to store the logical public bits '001'. If we re-program the cells with the first option (i.e., '11001'), the cells logically represent public bits '001' and private bit '0'. Conversely, if we re-program the cells with the second option (i.e., '10110'), the cells logically represent same public bits, but different private bit (i.e., '1'). (When reusing the target flash page without hiding the private bit, PEARL randomly choose one of the 2nd codewords.) As a result, since the data-hiding mechanism is seamlessly combined to WOM code, seemingly it has no difference with normal storage system using WOM code, thus make it possible to deny the existence of data-hiding mechanism.

However, it is unclear whether they can be applied to high-density flash chips that are mainstream in modern computing devices. To be deployed in practice, the re-program operation, which is necessary to implement their data-hiding mechanism, should be performed reliably. However, it is difficult to implement re-program operations in a reliable way. As multi-level cell technology advances, it becomes more difficult to precisely shift the V_{th} of already programmed cells. Even if the V_{th} of the target cells are shifted precisely, the V_{th} of another cells sharing the same wordline are unintentionally increased due to the high program-voltage stress. Such an undesired phenomenon is called program disturbance [17–19] and can cause uncorrectable errors in valid pages that must be stored reliably [22]. Furthermore, even assuming that

the reliability issues of re-programs are handled[4], they each additionally have their own problems. In the case of VT-HI, it is difficult to deploy in real-world due to its low usability. VT-HI has an encoding throughput of 4.3KB/s. For example, it will takes about 4 minutes to hide a 1MB image, which is not practically usable. (The decoding throughput is 0.3MB/s and is relatively high compared to the encoding throughput, but it is still not practically usable.) In the case of PEARL, since WOM codes are rarely (if ever) used in flash-based storage system, they can be viewed as the use of the special mechanism by the adversary, which could potentially lead to weakening of the deniability of the existence of a PD mechanism.

[4]For example, instead of encoding hidden bits into already programmed cells using re-programming operations, combine hidden bits with public bits in a buffer and then program them on free cells at once.

Chapter 4

Threat Model

4.1 The Capability of Adversary

We consider computationally-bounded realistic adversary who can investigate snapshots of an SSD multiple times (i.e., multiple-snapshot adversary). For example, whenever the user crosses the border of a country, the officer collects snapshots of the SSD and tries to detect the possibility of carrying sensitive data by analyzing and comparing collected snapshots. All the suspicious system states (or behaviors) observed by examining snapshots can be seen as the possibility of carrying data and the adversary will continue to suspect and maintain coercion. For example, if an adversary observes that a mysterious dummy writes are being performed to the free space (through comparison between snapshots), it can be suspected as an evidence of the data-hiding mechanism and the adversary can perform further investigation or tortures the device owner to admit the existence of hidden data. It is possible to obtain logical snapshots through the host interfaces and it is also possible to obtain physical snapshots by de-soldering flash chips and accessing the raw data through the known flash commands.

4.2 Assumptions

We establish the adversary based on the following assumptions:

- The adversary only stop coercing when there is no suspicious thing (e.g., unexplainable system states or behavior) is observed.
- The adversary is an on-event adversary [7], who can access the SSD only after the data-hiding process is done. Thus, the adversary cannot capture the run-time state of the device during the data-hiding process.
- The SSD firmware cannot be reverse-engineered. As in modern SSDs, the firmware is properly encrypted, so that the adversary cannot directly reason data-hiding mechanism through firmware investigation.
- The vendor-specific flash commands are not disclosed, and the adversary cannot find any vendor-specific flash commands for the flash chip. Since a flash chip becomes a black box whose internal behavior cannot be observed after being packaged, it is impossible to find out the existence of vendor-specific commands.
- It is impossible to inspect stored data by probing raw flash cells due to the structural characteristics of 3D NAND flash memory and technical limitations of probing tools [22].
- All the system software including the operating system, bootloader, SSD firmware are malware-free.

Chapter 5

Access-Centric Data Hiding Technique

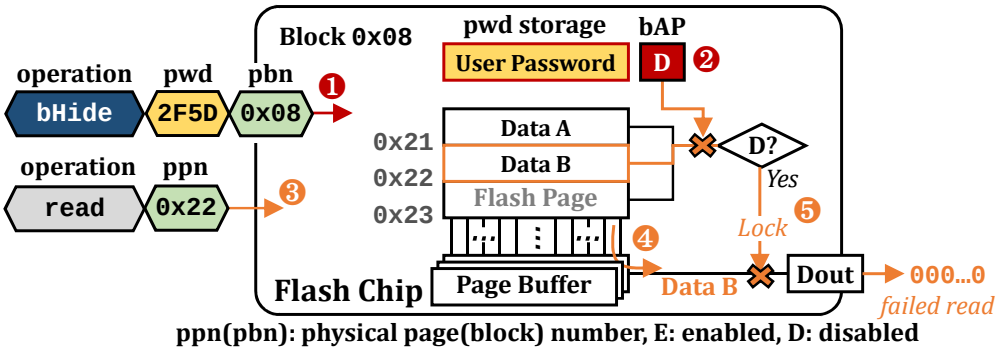
5.1 Approach Overview

We take an access-centric data hiding approach, which is based on the following two insights. First, if we can support access control mechanism, which can set access rights to private data, we can simply hide *the existence of the private data* by disabling accesses to private data. It would be also possible to retrieve the hidden private data whenever the user wants it back. Since the private data do not interfere with other public data (e.g., change the V_{th} of public data cells), there will be no data reliability issues. Moreover, since access to private data is blocked, the result of writing hidden data is invisible. It leads to a lightweight multiple-snapshot resistant PD solution as no exhausting system behavior (e.g., periodic dummy writes) is required to obfuscate hidden data writes. Second, if we can plausibly explain the existence of inaccessible data, we can also deny *the existence of a data hiding mechanism*. In other words, strong PD can be supported if we can explain that such inaccessible data are not the result of intentional access control to hide private data.

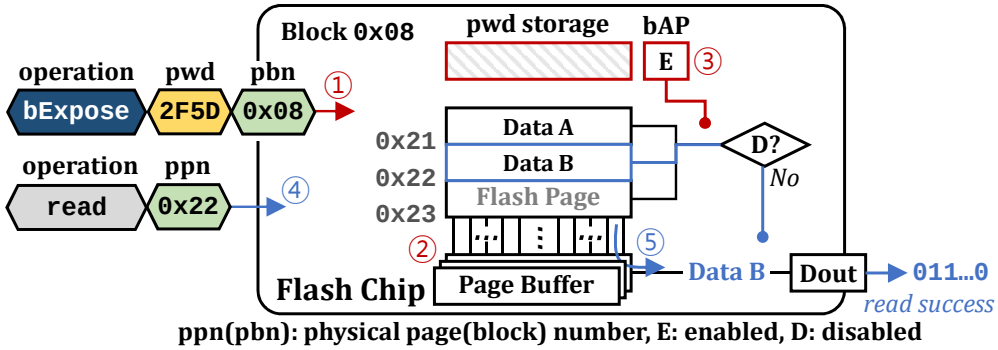
For plausible explanation of the existence of the inaccessible data, we assume that a target SSD is based on modern 3D flash memory which requires a delayed erasure scheme for meeting its data reliability requirements [23, 24]. Under the lazy erasure

scheme, a block is not erased when it is selected as a victim block of GC but its erasure is delayed until data are programmed on the block. Since a block should be lazily erased, the block may be exposed to unauthorized access while its erasure is delayed. To avoid such a security loophole, it is reasonable to employ data sanitization techniques to prevent unauthorized information access. Among the various sanitization techniques [22, 30–38], the lock-based sanitization technique [22] has been successfully demonstrated that disabling access to data (not physically destroy data) through on-chip access control can achieve the same effect with data sanitization. In Fidelius, we adopt same data sanitization approach, thus we can attribute (i.e., plausibly explain) the existence of inaccessible data to a result of data sanitization, not the result of intentional access control to hide private data.

In Fidelius, access control mechanism inside a flash chip is supported by two new flash commands, blockHide (`bHide`) and blockExpose (`bExpose`). As with lock-based sanitization, Fidelius disables access (i.e. lock) to data by controlling the on-chip access permission (AP) flag. The difference is, Fidelius supports only block-level AP (bAP) flag, not page-level. In addition, on-chip password storage is required to future on-chip password-based authentication for re-enabling access (i.e. unlock). The `bHide<pwd><pbn>` command associates a password `<pwd>` to the physical block number `<pbn>` by setting the `<pwd>` to on-chip password storage. It also lock the block by setting the bAP flag of `<pbn>` to the disabled state. On the other hand, The `bExpose <pwd><pbn>` command checks whether `<pwd>` matches a password set in the on-chip password storage. If the password matching fails, the `bExpose` command would be aborted, thus the bAP flag remains in the disable state. Conversely, if matching is successful, the block is unlocked by resetting bAP flag to the enabled state. Note that if we set the on-chip password storage to a one-time password (which is randomly generated) and forget the password, the data in the block becomes inaccessible



(a) bHide.



(b) bExpose.

Figure 3: Operational overview of bHide and bExpose.

until the entire block is erased, so it can have the same effect as data sanitization[5].

Figures 3(a) and 3(b) illustrate an operational overview of bHide and bExpose, respectively. To lock physical block 0x08 (denoted as PB#0x08), the bHide <2F5D><0x08> command (❶) sets the on-chip password storage to <2F5D> and bAP flag to disabled (❷). After the bAP flag is disabled, reads to any pages (e.g., reads to physical page 0x22 in ❸) in PB#0x08 fail, because Fidelius-enabled logic inside the flash chip checks if the bAP flag is enabled (❹) before sending the page data out of the flash chip (❺). On the other hand, as shown in Figure 7(b), to unlock PB#0x08, the bExpose

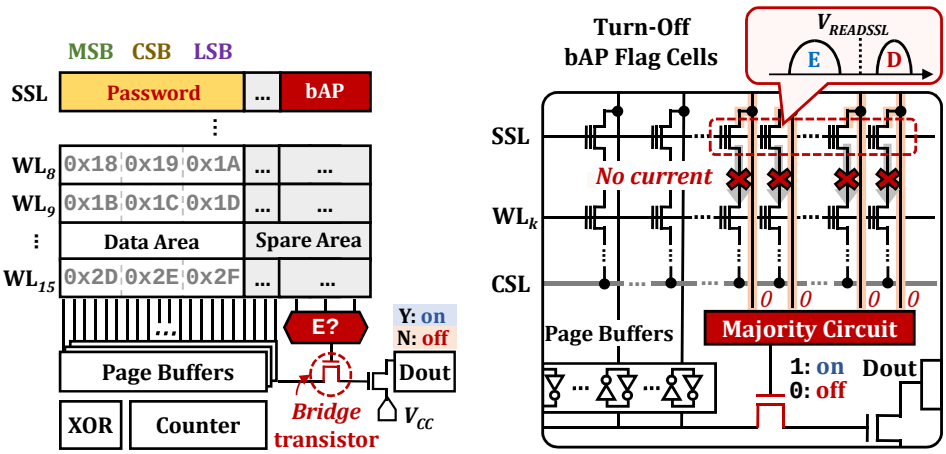
[5]As private data cannot be recovered without the password, it has the same level of security as the encryption-based sanitization techniques [36–38] that sanitizes data by forgetting (i.e., deleting) the encryption key of encrypted private data.

<2F5D><0x08> command (①) reads password of the block (from the on-chip password storage) to check if it matches <2F5D> (②). When a match fails, the command is aborted and no on-chip states (e.g., bAP flag, on-chip password storage) are changed. When the match it successful, Fidelius-enhanced logic clear the on-chip password storage and reset bAP flag to enabled (③). The read request to page 0x22 (④) is then successfully performed and the data can be transferred out from a flash chip (⑤).

5.2 Flash Commands for Block Access Control

5.2.1 Organizational Overview

Figures 4 shows an organizational overview of our proposed bHide and bExpose implementations. As shown in Figures 4 (a), Per-block on-chip password storage and bAP flag are allocated on the main data area and spare area of SSL, respectively. Note that it is possible to program (and erase) the SSL of a block as a normal WL because SSL transistors are implemented with normal flash cells in 3D flash organization [28]. If the bAP flag is disabled, the bridge transistor is disconnected. Thus, even if data



(a) Organizational overview.

(b) A bAP implementation.

Figure 4: An overview of the Fidelius implementation.

(placed in the main data area) can be read into the page buffer, it cannot be transferred out of the flash chip, instead, all-zero data are sent out of a flash chip. To implement password matching logic inside the flash block, we use existing on-chip resources, XOR gate and bit counter. In general, flash chips are equipped with an XOR gate and a bit counter to control the V_{th} distribution of the flash cell [6]. Thus, we can compute the number of different bits between the input password (e.g., `<2F5D>` in Figures 3(b)) and the password set in the on-chip password storage by using an XOR gate and a bit counter. If password matching is successful, bAP flag is reset to enabled state and on-chip password storage is cleared by erasing the SSL. Then, the requested data can be transferred from the page buffer to the data out circuit.

5.2.2 Implementation

To implement `bHide` and `bExpose` with an organization shown in Figures 4 (a), two requirements should be guaranteed. First, the bAP flag must reliably perform the role of an off switch under various error sources (e.g., retention times). For example, if the bAP flag value is mistakenly turned to enabled state after a long retention time, the data in locked block can be accessed again and the hidden private data can be accidentally exposed. Second, the on-chip password storage must stably store the user password under various error sources, so that no false positive or false negative are occurred during the password matching process. For example, if some password bits (stored in the on-chip password storage) are flipped, `bExpose` *with wrong password* may succeed and private data may be accidentally exposed, or `bExpose` *with correct password* may fail and private data may be accidentally lost.

We first distinguish between enabled and disabled states according to the V_{th} of flash cells for the bAP flag, and employ a majority circuit to satisfy the first require-

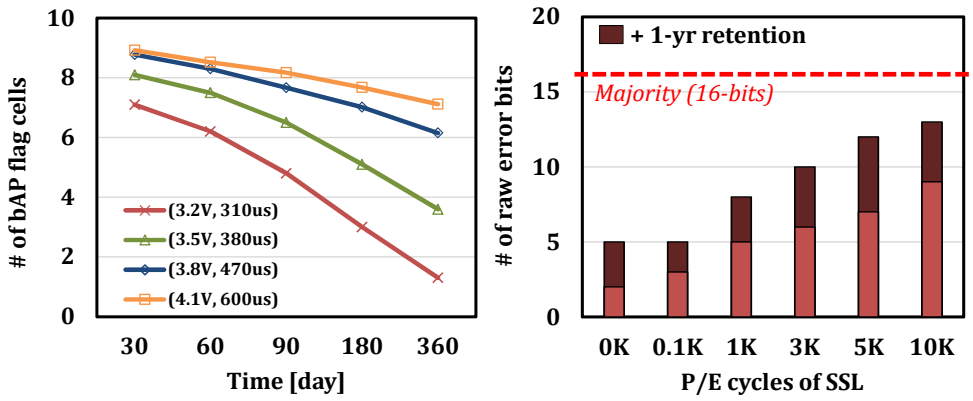
[6] To program a flash cell to a target state, the incremental step pulse programming (ISPP) scheme [43], which gradually increases a program voltage until all the flash cells in a page are shifted to their target V_{th} range, is commonly used. In ISPP, after each program step, flash cells are checked if they have been correctly programmed using XOR gate and bit counter.

ment. If the V_{th} of bAP flag cells are shifted higher than $V_{READSSL}$ (i.e., disabled), voltage applied to SSL to activate the corresponding block during page read operations, no current can flow through BLs and the page buffer is disconnected from the data out circuit. As a result, it is possible to block all read requests to the block. As with a lock-based data sanitization technique [22], we allocate 9 flash cells for bAP flag and employ a 9-bit majority circuit to implement error-tolerant bAP flag. When we program bAP flag cells with a sufficiently high V_{th} ($> V_{READSSL}$), the bAP flag can reliably turned off for a sufficient period of time (e.g., 1-year) with the help of the majority circuit.

In order to determine the design parameter of the bAP implementation, we conducted reliability evaluations using a total of 160 state-of-the-art 3D flash chips. We evenly selected 64 test blocks from each flash chip at different physical block locations, and tested the SSL in each selected block. Using an in-house custom test board, we evaluated reliability metrics while varying the number of P/E cycles (from 0 to 10K) and data retention requirements (up to 1-year). Our results are measured under worst-case reliability condition[7]. As shown in Figures 5 (a), we evaluated how the number of retention errors changes under various target V_{th} of bAP flag cells (under 10K P/E cycles). For example, the pair (3.2V, 310us) means that the target V_{th} of bAP flag cells is 3.2V, and the program time required to move their V_{th} to 3.2V is 310us. While (3.8V, 470us) pair and (4.1V, 600us) pair satisfy the bAP flag implementation requirements, we conservatively select (4.1V, 600us) pair as our `bHide` design parameter.

It is more challenging to implement reliable on-chip password storage. The on-chip password storage is also made up of flash cells and potentially there can be unavoidable bit errors. However, since there is no ECC module inside the flash chip, password matching logic cannot be helped by ECC, and simply employing a majority circuit (like implementation of bAP flag) is also not feasible due to space (and cost) overhead.

[7]Our test procedure followed the JEDEC standard [44] recommended for commercial-grade flash products



(a) Retention error bits of the bAP flag.

(b) Raw bit errors of the password cells.

Figure 5: Device-level experimental results for Fidelius

For example, for a password of 256-bits length, if we assign 9-redundancy for each password bit, we need 256 9-bit majority circuits, which is 256 times the bAP flag implementation overhead.

We satisfy the second requirement with no additional hardware resources by implementing simple *majority checking* mechanism using only on-chip resources (e.g., page buffers, XOR gate, bit counter). Using an XOR gate and a bit counter, we can find how many bits are different between the expected value (stored in the page buffer) and the actually value (in the flash cells of on-chip password storage). We allocate k flash cells for each bit of the password when performing `bHide` command and check that the majority of each bit of password matches the expected value (i.e., input password from `bExpose`) by exploiting on-chip resources when performing `bExpose` command. For each bit of a password (programmed in k flash cells), if majority (i.e., $> k/2$) are equal to the expected bit value, password matching logic determines that the input password is correct. We set the password length to 256-bits, which is equivalent to the longest key length supported by the advanced encryption algorithm [45]. To determine k, we measured the changes in the maximum number of raw bit errors (per 1-KB in SSL) as shown in Figures 5 (b). Since the maximum raw bit error is 13-bits

under high P/E cycles with 1-year retention time, if the password (with redundancy) fits within 1-KB and the majority of each bit of the password exceeds 13, we can guarantee error-tolerant on-chip password storage for 1-year. We set k to be 32, so that the 256-bits password (with 32 redundancy for each bit) fits into 1-KB while the majority of each bit of the password (i.e., > 16) is greater than the maximum bit errors (i.e., 13). Since the size of the main data area of SSL is usually 8K or 16K and only 1K is dedicated for on-chip password storage, there is a remaining capacity (e.g., 7K or 15K). The remaining main data area of SSL will be used to store metadata for hidden data management (see Section 6). Note that metadata is programmed together with password and bAP when `bHide` is performed.

Chapter 6

SafeSSD: Fidelius-based PD Solution

6.1 Overview

We design a Fidelius-enabled flash-based storage system, called SafeSSD, a strong PD solution with multiple-snapshot resistance. SafeSSD operates the same as a normal SSD, except that it supports data sanitization. SafeSSD supports two logical volumes, the public volume, which is placed across the entire blocks, and the private volume, which is placed within the subset of free blocks. The free blocks constituting the private volume are locked with the `bHide` set with the user's secret password. Multiple private volumes can exist in a SafeSSD, each is set with a different secret password, and can be managed separately using the password as an identifier.

The user can manage the private volume through the existing NVMe admin interface [25]. As NVMe admin interface provides passthru [46, 47] feature, which allows users to submit an arbitrary command, the user can pass hidden commands with password. When receiving the hidden command, SafeSSD succeed the command only if the password matches, and if the password is wrong, it reacts as if it does not support such a command (e.g., return invalid command opcode). Therefore, without the password, the adversary cannot detect the existence of an interface for the PD.

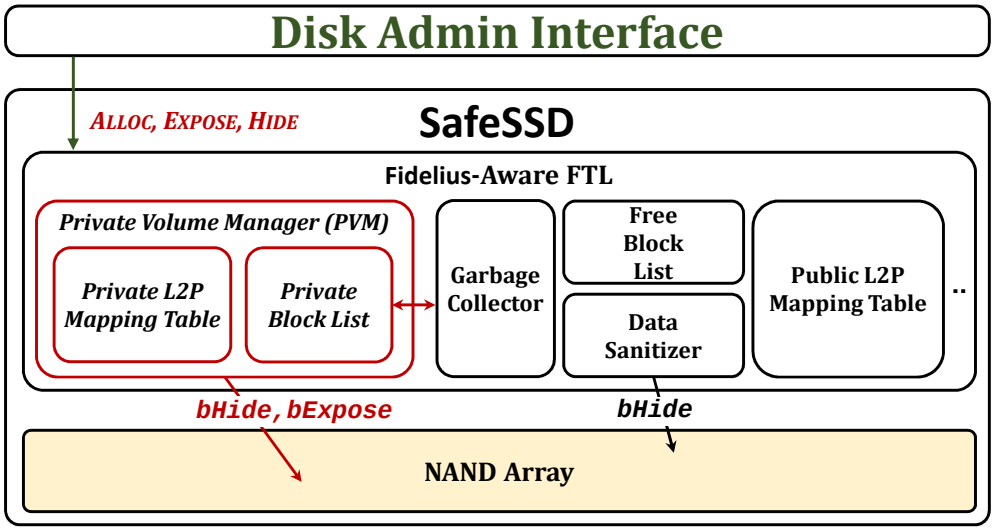


Figure 6: An organizational overview of SafeSSD

6.2 Fidelius-Aware FTL

Figure 6 depicts an overall organization of SafeSSD. The private volume manager (PVM) consists of a list of private blocks constituting a private volume and a private L2P mapping table that links private data's logical page address to physical page address. All the private blocks are taken from the free blocks, which is garbage collected and sanitized (i.e., locked with randomly generated one-time password) by garbage collector (GC) and data sanitizer, respectively. Since private blocks are also managed as free blocks, they can be erased to service writes to the public volume. When the GC needs to erase a free block to service public writes, it checks the private block list of PVM. If the free block is not part of the private volume, GC simply erases the block and the block is used for servicing public writes. On the other hand, if the free block is a private block constituting the private volume, the private data stored in the block is migrated to a new private block (which is selected from the free block list). Note that PVM manages only private volume in *exposed* state, and if no private volume is exposed, that is, if all private volumes are in *hidden* state, private block list of PVM

Table 1: User-level steps τ_{exp} and τ_{hid}.

Tasks	User-level steps
Exposing (τ_{exp})	EXPOSE(p) $\rightarrow$ re-scan $\rightarrow$ create a partition $\rightarrow$ mount
Hiding (τ_{hid})	umount $\rightarrow$ delete a partition $\rightarrow$ HIDE(p) $\rightarrow$ re-scan

is empty. The device owner can avoid unintentional erasure of private data by leaving the private volume in an exposed state, while hiding the existence of private data by changing the private volume to *hidden* state under the attacks.

6.3 User Interfaces

SafeSSD supports several hidden commands to managing private volumes. PVM receives hidden commands from the host system and directly perform flash operations on the blocks. Assume that a private volume σ set with user's secret password p exists on SafeSSD. For 1) ALLOC(p, c), PVM selects and erases blocks as much as capacity c (in MB) from free (i.e. locked) blocks. Then those blocks are allocated to σ and construct private L2P mapping table entries. For 2) EXPOSE(p), PVM tries `bExpose` with p on all locked blocks and makes a list of unlocked blocks. For unlocked blocks, PVM retrieves the L2P mapping table of σ, which was stored within the SSL, and reconstruct it. The PVM then increases the usable disk capacity of the SafeSSD by the capacity of σ, making the σ visible to the host system as an *additional disk space* (i.e., the increment of the number of sectors). Lastly, for 3) HIDE(p), PVM store the σ's L2P mapping table and the user password p on the SSL by performing `bHide` on all blocks constituting σ. Then PVM decrease SafeSSD's usable disk capacity by the capacity of σ, making the σ invisible to the host system again.

In addition to the help of the NVMe admin interface, the host system exploits the existing disk utilities (e.g., fdisk [48], mkfs [49], etc.) to access private volume through

common file interface. After the EXPOSE(p) command is executed successfully, the host system can detect the increased usable disk capacity through the existing disk re-scan feature [50,51]. After the private volume is visible to the host system as additional sectors, the host system can create a partition and mount the file system through the existing disk utilities. (If it is first private volume use, ALLOC(p, c) command should be performed to allocate blocks for the private volume and a file system should be formatted on private volume.) When hiding the private volume, it is necessary to make the private volume invisible again. The host system must unmount the file system mounted on private volume, delete its partition, and re-lock the unlocked blocks by passing the HIDE(p) command to SafeSSD. Lastly, disk re-scan should be performed to restore the original SafeSSD capacity. We divide the above steps to be performed in the host system into *exposing* task (τ_{exp}) and *hiding* task (τ_{hid}) and summarized them in Table 1.

Chapter 7

Evaluation Results

7.1 Experimental Settings

We implement SafeSSD on FEMU [52], a state-of-the-art SSD emulator, with an Fidelius-enabled emulated flash model. In the performance evaluation, we limited the SSD capacity to 16-GB for fast evaluation while we set the SSD capacity to a maximum of 128-GB to reflect the actual user-perceived latency in usability test. We set operation timing parameters of flash commands for t_{READ}, t_{PROG} and t_{BERS} to 70μs, 660μs and 3.5ms, respectively. In addition, based on our device-level experiment results (in Section 5), we set operation timing parameters of t_{BHIDE} and $t_{BEXPOSE}$ to 600μs and 3.5ms, respectively.

For the performance evaluation, we use three different benchmark traces, Var-

Table 2: A summary of I/O characteristics of three traces.

Benchmark	read:write	File access pattern	Write size
Varmail	3:7	create/append/delete e-mails	16–32 KB
OLTP	2:8	overwrite data files and log files	4–256 KB
Fileserver	4:6	create/append/delete files	16–128 KB

mail, OLTP and Fileserver, from the Filebench benchmark tool [53]. Table 2 summarized I/O characteristics of the benchmarks: the read to write ratio, the access pattern, and the write size. For a stable evaluation result, each trace initially fills 75% of the SafeSSD capacity and continues I/O until 4 drive writes are performed. We evaluated our SafeSSD against a conventional SSD, which does not support no data sanitization. Since the public volume and the private volume are managed in the same way, only the capacity is different, there is no difference in performance. Thus, we will only show the performance of the public volume.

For the usability test, we measured the time spent performing 1) τ_{exp} and 2) τ_{hid}. Since the time consumed by SafeSSD's hidden command is correlated with the capacity of the disk and private volume, we vary the entire disk capacity and the capacity of the private volume. We set the disk capacity to 64-GB and 128-GB, and measure the latency when the private volume capacity was 2.5% and 5% of the disk capacity.

7.2 Performance Evaluation

To evaluate the performance of SafeSSD, we measure IOPS under the three workloads. SafeSSD's IOPS is normalized over one from an SSD which does not support data sanitization. Figure 7 shows normalized IOPS of SafeSSD under each workload.

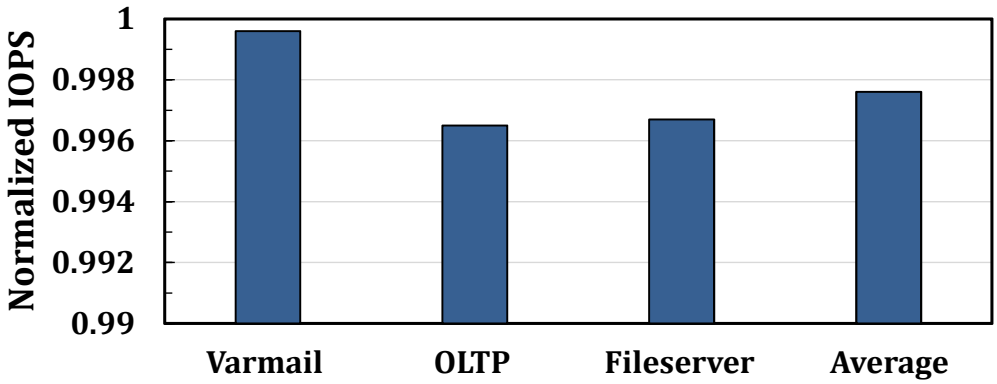


Figure 7: Performance of SafeSSD under three different workloads.

Since most of the time taken for GC is spent copying valid pages within a victim block, data sanitization (i.e., performing `bHide`) impose only negligible performance overhead. As a result, SafeSSD shows almost the same performance as a normal SSD. Note that SafeSSD supports only block-level data sanitization (which is the minimum requirement for strong PD), not page-level data sanitization. If SafeSSD additionally performs page-level data sanitization, the performance overhead will increase slightly, but still be comparable with a normal SSD [22].

7.3 Usability Evaluation

In order to evaluate the usability of the private volume, we evaluated the operation latency of each task in Table 1 as shown in Figure 8. Because some steps composing each task are affected by the size of the private volume, the operation latency of tasks increases as the capacity of the private volume increases. For example, in the case of the HIDE(p, c) command, the larger the size of the private volume is requested, the more blocks that need to be locked and it takes longer time. In both τ_{exp} and τ_{hid}, each operation latency does not exceed 0.3 seconds even in the case of (5%, 128-GB). Therefore, considering the trend of increasing operation latency compared to capacity increase, SafeSSD can achieve high usability even the disk capacity is large.

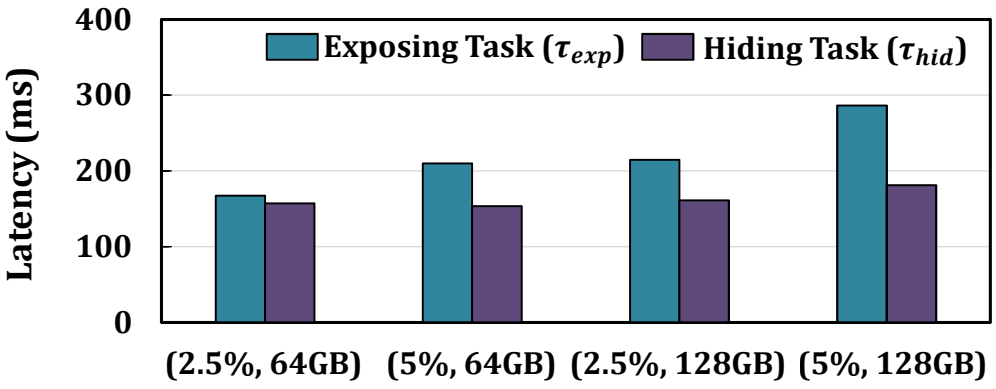


Figure 8: Operation latency of SafeSSD interfaces.

For example, in the case of (5%, 1-TB), we expect that τ_{exp} takes less than 0.8 seconds and τ_{hid} takes less than 0.4 seconds.

Chapter 8

Related Work

In order to protect private data, studies have been actively conducted to hide the existence of private data. We briefly discuss closely related prior work that aims to provide plausible deniability for hidden private data.

Plausibly Deniable Encryption Plausibly deniable encryption (PDE) allows a given ciphertext to be decrypted as original private data (using secret key) or plausible public data (using decoy key). The user can hide the existence of private data by disclosing only the decoy key under the coercion. However, it is hard to implement special encryption scheme in which ciphertext can be decrypted into a number of plausible plaintext, and existing implementation results in an increase in ciphertext size, potentially allowing an adversary to notice that PDE is in use [54].

Steganography Classical steganography conceals private data by embedding it in other cover data [55–58]. As cover data, media data such as image, audio, and video files are mainly used, and private data is hidden by applying unobtrusive distortion to the cover data. However, since media data is generally expected to be immutable, it is difficult to hide data from the multiple-snapshot adversary when the contents of private data are updated. On-chip steganography is similar to classical steganography, with the difference that private data is embedded in the analog domain of storage medium (e.g., flash cell's programming time [59] or threshold voltage level [16]) rather than digital

domain (such as media data). They are suitable for providing strong PD because it is almost impossible to detect the existence of data hiding mechanism. Unfortunately, however, they are difficult to be put into practical use because they degrade the lifetime of the flash memory, or degrade the reliability of the data stored in the flash memory. In addition, like classical steganography, it is difficult to defend against multiple-snapshot adversary.

Deniable Storage System The deniable storage system is the PD solution for the storage system. Many of them exploit ciphertext indistinguishability [39] to hide encrypted private data between random data [4–6]. However, as random data is located in an area where the file system is inaccessible (e.g., free space of the disk), if changes occurs, it can be seen as potentially signs of data-hiding. Thus, to defend against multiple-snapshot adversary, several studies have employed the dummy writes as part of the storage system's normal behavior [7–12] to obscure the cause of random data changes. However, as discussed in Section 3, such a mysterious behavior can be potential sign of data-hiding, so the PD is weakened. Recently, new PD solutions have been proposed that can hide the fact that the storage system is supporting data hiding. Chen *et al*. proposed a strong PD solution by exploiting on-chip steganography technique [16] as a building block [14] and a deniable storage with re-purposed WOM code that supports data hiding [15]. However, as both were discussed in Section 3, the applicability to modern high-density 3D NAND flash memory is unclear due to data reliability issues.

Chapter 9

Conclusions

We have presented Fidelius, a plausibly deniable data hiding technique for modern flash-based storage systems. Fidelius supports per-block access control with two new flash commands, `bHide` and `bExpose`, that disable accesses to blocks and re-enable accesses to blocks, respectively. By exploiting existing lock-based data sanitization as a cloak for access control, Fidelius can plausibly deny the existence of the data hiding mechanism. Using state-of-the-art 3D flash chips, we validate that `bHide` can guarantee that accesses to a target block are disabled and `bExpose` can reliably re-enable the accesses to the block. We designed a Fidelius-enabled flash storage system, SafeSSD, that abstracts the management of private volume and provides high-level private volume management interfaces to a host system for ease of use. Our experimental results show that SafeSSD can provide strong plausible deniability with negligible performance overhead and high usability.

Bibliography

[1] Apple. Filevault. https://support.apple.com/en-us/HT204837, 2015.

[2] Source. Android full disk encryption. https://source.android.com/security/encryption/, 2016.

[3] Microsoft. Bitlocker. https://technet.microsoft.com/en-us/library/hh831713, 2013.

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

[5] TrueCrypt. Free open source on-the-fly disk encryption software. http://www.truecrypt.org/, 2012.

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

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

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

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

[10] Chen Chen, Anrin Chakraborti, and Radu Sion. Pd-dm: an efficient locality-preserving block device mapper with plausible deniability. *Proceedings on Privacy Enhancing Technologies*, 2019(1):153–171, 2019.

[11] Aviad Zuck, Udi Shriki, Donald E. Porter, and Dan Tsafrir. Preserving hidden data with an ever-changing disk. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 50–55, New York, NY, USA, 2017. ACM.

[12] Shijie Jia, Qionglu Zhang, Luning Xia, Jiwu Jing, and Peng Liu. Mdeftl: Incorporating multi-snapshot plausible deniability into flash translation layer. *IEEE Transactions on Dependable and Secure Computing*, 2021.

[13] Chen Chen, Xiao Liang, Bogdan Carbunar, and Radu Sion. Sok: Plausibly deniable storage. *arXiv preprint arXiv:2111.12809*, 2021.

[14] Chen Chen, Anrin Chakraborti, and Radu Sion. Infuse: Invisible plausibly-deniable file system for nand flash. *Proc. Priv. Enhancing Technol.*, 2020(4):239–254, 2020.

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

[16] Aviad Zuck, Yue Li, Jehoshua Bruck, Donald E. Porter, and Dan Tsafrir. Stash in a flash. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 169–188, Oakland, CA, February 2018. USENIX Association.

[17] Rino Micheloni, Luca Crippa, and Alessia Marelli. *Inside NAND flash memories*. Springer Science & Business Media, 2010.

[18] Seiichi Aritome. *NAND flash memory technologies*. John Wiley & Sons, 2015.

[19] Alessandro Torsi, Yijie Zhao, Haitao Liu, Toru Tanzawa, Akira Goda, Pranav Kalavade, and Krishna Parat. A program disturb model and channel leakage current study for sub-20 nm nand flash cells. *IEEE Transactions on Electron Devices*, 58(1):11–16, 2010.

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

[21] Fidelius charm. https://harrypotter.fandom.com/wiki/Fidelius_Charm#.

[22] Myungsuk Kim, Jisung Park, Genhee Cho, Yoona Kim, Lois Orosa, Onur Mutlu, and Jihong Kim. Evanesco: Architectural support for efficient data sanitization in modern flash-based storage systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1311–1326, New York, NY, USA, 2020. ACM.

[23] Christian Monzio Compagnoni, Andrea Ghetti, Michele Ghidotti, Alessandro S. Spinelli, and Angelo Visconti. Data retention and program/erase sensitivity to the array background pattern in deca-nanometer nand flash memories. *IEEE Transactions on Electron Devices*, 57(1):321–327, 2009.

[24] Chulbum Kim, Doo-Hyun Kim, Woopyo Jeong, Hyun-Jin Kim, Il Han Park, Hyun-Wook Park, JongHoon Lee, JiYoon Park, Yang-Lo Ahn, Ji Young Lee, Seung-Bum Kim, Hyunjun Yoon, Jae Doeg Yu, Nayoung Choi, NaHyun Kim, Hwajun Jang, JongHoon Park, Seunghwan Song, YongHa Park, Jinbae Bang, Sanggi Hong, Youngdon Choi, Moo-Sung Kim, Hyunggon Kim, Pansuk Kwak, Jeong-Don Ihm, Dae Seok Byeon, Jin-Yub Lee, Ki-Tae Park, and Kye-Hyun Kyung. A 512-gb 3-b/cell 64-stacked wl 3-d-nand flash memory. *IEEE Journal of Solid-State Circuits*, 53(1):124–133, 2017.

[25] Linux-Nvme. linux-nvme/nvme-cli: Nvme management command line interface. https://github.com/linux-nvme/nvme-cli.

[26] Ryuji Yamashita, Sagar Magia, Tsutomu Higuchi, Kazuhide Yoneya, Toshio Yamamura, Hiroyuki Mizukoshi, Shingo Zaitsu, et al. A 512gb 3b/cell flash memory on 64-word-linelayer bics technology. In *Proceedings of International Solid-State Circuits Conference*, 2017.

[27] J. Maserjian and N. Zamani. Behavior of the si/sio2 interface observed by fowler-nordheim tunneling. *Journal of Applied Physics*, 53(1):559–567, 1982.

[28] Kazushige Kanda, Noboru Shibata, Toshiki Hisada, Katsuaki Isobe, Manabu Sato, Yui Shimizu, Takahiro Shimizu, Takahiro Sugimoto, Tomohiro Kobayashi, Naoaki Kanagawa, Yasuyuki Kajitani, Takeshi Ogawa, Kiyoaki Iwasa, Masatsugu Kojima, Toshihiro Suzuki, Yuya Suzuki, Shintaro Sakai, Tomofumi Fujimura, Yuko Utsunomiya, Toshifumi Hashimoto, Naoki Kobayashi, Yuuki Matsumoto, Satoshi Inoue, Yoshinao Suzuki, Yasuhiko Honda, Yosuke Kato, Shingo Zaitsu, Hardwell Chibvongodze, Mitsuyuki Watanabe, Hong Ding, Naoki Ookuma, and Ryuji Yamashita. A 19 nm 112.8 mm^2 64 gb multi-level flash memory with 400 mbit/sec/pin 1.8 v toggle mode interface. *IEEE Journal of Solid-State Circuits*, 48(1):159–167, 2012.

[29] Ki-Tae Park, Sangwan Nam, Daehan Kim, Pansuk Kwak, Doosub Lee, Yoon-He Choi, Myung-Hoon Choi, Dong-Hun Kwak, Doo-Hyun Kim, Min-Su Kim, Hyun-Wook Park, Sang-Won Shim, Kyung-Min Kang, Sang-Won Park, Kangbin Lee, Hyun-Jun Yoon, Kuihan Ko, Dong-Kyo Shim, Yang-Lo Ahn, Jinho Ryu, Donghyun Kim, Kyunghwa Yun, Joonsoo Kwon, Seunghoon Shin, Dae-Seok Byeon, Kihwan Choi, Jin-Man Han, Kye-Hyun Kyung, Jeong-Hyuk Choi, and Kinam Kim. Three-dimensional 128 gb mlc vertical nand flash memory with 24-wl stacked layers and 50 mb/s high-speed programming. *IEEE Journal of Solid-State Circuits*, 50(1):204–213, 2014.

[30] Sarah Diesburg, Christopher Meyers, Mark Stanovich, Michael Mitchell, Justin Marshall, Julia Gould, and An-I Andy Wang. Trueerase: Per-file secure deletion for the storage data path. In *Proceedings of the Annual Computer Security Applications Conference*, 2012.

[31] Kyungmoon Sun, Jongmoo Choi, and Sam H. Noh. Models and design of an adaptive hybrid scheme for secure deletion of data in consumer electronics. *IEEE Transactions on Consumer Electronics*, 54(1):100–104, 2008.

[32] Wei-Chen Wang, Chien-Chung Ho, Yuan-Hao Chang, Tei-Wei Kim, Kuo, and Ping-Hsien Lin. Scrubbing-aware secure deletion for 3-d NAND flash. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2790–2801, 2018.

[33] Michael Yung Chung Wei, Laura M. Grupp, Frederick E. Spada, and Steven Swanson. Reliably erasing data from flash-based solid state drives. In *Proceedings of the USENIX Conference on File and Storage Technologies*, 2011.

[34] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Nfps: Adding undetectable secure deletion to flash translation layer. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security*, 2016.

[35] Ping-Hesien Lin, Yu-Ming Chang, Yung-Chun Li, Chien-Chung Ho, and Yuan-Hao Chang. Achieving fast sanitization with zero live data copy for MLC flash memory. In *Proceedings of the International Conference on Computer-Aided Design*, 2018.

[36] Junghee Lee, Kalidas Ganesh, Hyuk-Jun Lee, and Youngjae Kim. Fessd: A fast encrypted ssd employing on-chip access-control memory. *IEEE Computer Architecture Letters*, 16(2):115–118, 2017.

[37] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 333–348, 2012.

[38] Jaeheung Lee, Junyoung Heo, Yookun Cho, Jiman Hong, and Sung Y Shin. Secure deletion for nand flash file system. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 1710–1714, 2008.

[39] Ciphertext indistinguishability. https://en.wikipedia.org/wiki/Ciphertext_indistinguishability#Indistinguishable_from_random_noise.

[40] Jubayer Mahmod and Matthew Hicks. Invisible bits: hiding secret messages in sram's analog domain. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1086–1098, 2022.

[41] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Deftl: Implementing plausibly deniable encryption in flash translation layer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2217–2229, New York, NY, USA, 2017. ACM.

[42] Yinglei Wang, Wing-kei Yu, Shuo Wu, Greg Malysa, G Edward Suh, and Edwin C Kan. Flash memory for ubiquitous hardware security functions: True

random number generation and device fingerprints. In *2012 IEEE Symposium on Security and Privacy*, pages 33–47. IEEE, 2012.

[43] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, et al. A 3.3 v 32 mb nand flash memory with incremental step pulse programming scheme. *IEEE Journal of Solid-State Circuits*, 30(11):1149–1156, 1995.

[44] JEDEC. JEDEC Solid State Technology Assn., Method for Developing Acceleration Models for Electronic Component Failure Mechanisms. https://www.jedec.org, 2003. [JESD91A].

[45] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[46] Nvme-io-passthru(1). https://manpages.debian.org/testing/nvme-cli/nvme-io-passthru.1.en.html. Accessed: 2022-01-24.

[47] Nvme-io-passthru(1). https://manpages.debian.org/testing/nvme-cli/nvme-admin-passthru.1.en.html. Accessed: 2022-01-24.

[48] fdisk(8) — linux manual page. https://man7.org/linux/man-pages/man8/fdisk.8.html. Accessed: 2022-01-25.

[49] mkfs(8) — linux manual page. https://man7.org/linux/man-pages/man8/mkfs.8.html. Accessed: 2022-02-26.

[50] [resend v1 0/5] add support for block disk resize notification. https://lwn.net/ml/linux-kernel/20200102075315.22652-1-sblbir@amazon.com/. Accessed: 2022-01-24.

[51] detect online disk resize. https://lwn.net/Articles/296401/. Accessed: 2022-01-24.

[52] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. The case of femu: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies*, pages 83–90, 2018.

[53] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *login: The USENIX Magazine*, 41(1):6–12, 2016.

[54] Rein Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In *Annual International Cryptology Conference*, pages 90–104. Springer, 1997.

[55] Shikha Sharma and Devendra Somwanshi. A dwt based attack resistant video steganography. In *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, pages 1–5, 2016.

[56] Xintao Duan, Liu Nao, Gou Mengxiao, Dongli Yue, Zimei Xie, Yuanyuan Ma, and Chuan Qin. High-capacity image steganography based on improved fc-densenet. *IEEE Access*, 8:170174–170182, 2020.

[57] Dipti Watni and Sonal Chawla. A comparative evaluation of jpeg steganography. In *2019 5th International Conference on Signal Processing, Computing and Control (ISPCC)*, pages 36–40. IEEE, 2019.

[58] Yunzhao Yang, Yuntao Wang, Xiaowei Yi, Xianfeng Zhao, and Yi Ma. Defining joint embedding distortion for adaptive mp3 steganography. In *Proceedings of the ACM Workshop on Information Hiding and Multimedia Security*, pages 14–24, 2019.

[59] Yinglei Wang, Wing-kei Yu, Sarah Q Xu, Edwin Kan, and G Edward Suh. Hiding information in flash memory. In *2013 IEEE Symposium on Security and Privacy*, pages 271–285. IEEE, 2013.

초 록

암호화는 개인 데이터의 내용을 숨길 수 있으나, 그 존재 자체를 숨길 수는 없다. 따라서 사용자에게 암호 해독 키를 공개하도록 강요하는 공격자로부터 프라이버시가 손상된다. 이러한 강압적 공격자에 대항하기 위해, 데이터 숨김 기능을 지원하는 합리적 부정기능 솔루션이 제안되어왔다. 합리성을 향상시키기 위해, 데이터 숨김 기능 자체의 사용까지 부정할 수 있게끔 강력한 합리적 부정기능을 지원하는 것이 중요하다. 불행하게도, 기존에 강력한 합리적 부정기능을 지원하는 솔루션들은 신뢰성 문제로 인하여 현대의 플래시 기반 저장장치 시스템에 구현되기에 실용적이지 못하다. 본 연구에서는, 강력한 합리적 부정기능을 지원하면서도 고집적 3D 낸드 플래시 메모리에서 실용가능한, 새로운 접근 중점적 데이터 숨김 메커니즘인 Fidelius 를 제안한다. Fidelius 는 칩 내부 자원을 활용하여, 블록 수준에서의 데이터 접근 권한을 재설정할 수 있는 플래시 잠금 (`bHide`) 및 잠금해제 (`bExpose`) 커맨드를 지원한다. 공격자는 잠긴 블록 내부를 조사할 수 없으므로, Fidelius 는 단순히 잠긴 블록에 개인 데이터를 저장함으로써 그 존재를 부정할 수 있다. 잠긴 블록들의 존재는 데이터 숨김 기능의 결과가 아닌, 과거 데이터에 대한 접근을 제어하는 데이터 세니타이제이션의 결과로써 설명될 수 있기에 Fidelius 는 강력한 합리적 부정기능을 제공할 수 있다. 제안한 기술을 평가하기 위해, 에뮬레이터를 기반으로, (멀티스냅샷 저항력과 더불어 강력한 합리적 부정기능을 지원하는) Fidelius 를 적용한 플래시 스토리지 시스템인 SafeSSD 를 구축하였다. 평가 결과를 통해, SafeSSD 는 신뢰성 문제 없이, 경미한 성능 오버헤드만을 가지면서 강력한 합리적 부정기능을 지원할 수 있음을 보였다.

주요어: 솔리드 스테이트 드라이브, 3D 낸드 플래시 메모리, 보안, 프라이버시, 합리적 부정기능, 부정가능 저장장치
학번: 2019-21844