Chen Chen*, Anrin Chakraborti, and Radu Sion

INFUSE: Invisible plausibly-deniable file system for NAND flash

Abstract: Protecting sensitive data stored on local storage devices e.g., laptops, tablets etc. is essential for privacy. When adversaries are powerful enough to coerce users to reveal encryption keys/passwords, encryption alone becomes insufficient for data protection. Additional mechanisms are required to hide the very presence of sensitive data.

Plausibly deniable storage systems (PDS) are designed to defend against such powerful adversaries. Plausible deniability allows a user to deny the existence of certain stored data even when an adversary has access to the storage medium. However, existing plausible deniability solutions leave users at the mercy of adversaries suspicious of their very use. Indeed, it may be difficult to justify the use of a plausible deniability system while claiming that no sensitive data is being hidden.

This work introduces INFUSE, a plausibly-deniable file system that hides not only contents but also the evidence that a particular system is being used to hide data. INFUSE is "invisible" (identical layout with standard file system), provides redundancy, handles overwrites, survives data loss, and is secure in the presence of multi-snapshot adversaries. INFUSE is efficient. Public data operations are orders of magnitude faster than existing multi-snapshot resilient PD systems, and only 15% slower than a standard non-PD baseline, and hidden data operations perform comparably to existing systems.

DOI 10.2478/popets-2020-0071
Received 2020-02-29; revised 2020-06-15; accepted 2020-06-16.

1 Introduction

Storage technology has advanced rapidly in recent years. This has resulted in users storing significant amounts of data, including sensitive information, e.g., credit card details, travel documents etc. on personal storage devices. It is essential to protect this information from unauthorized disclosure. There are numerous examples where stolen unprotected devices have led to privacy breaches of catastrophic proportions [5, 6].

Full Disk Encryption (FDE) is usually the first line of defense. However, encryption is not enough to protect against powerful adversaries that can coerce users to hand over their decryption keys, passwords [9].

In practice, protecting against such powerful adversaries is essential due to increasing instances of intrusion by unfriendly powerful nation state adversaries [15, 21]. This is sometimes a matter of life and death [23] as documented in numerous cases where information had to be transferred through checkpoints manned by hostile adversaries. In a notable example, the human rights group Network for Human Rights Documentation-Burma (ND-Burma) carried data proving hundreds of thousands of human rights violations out of the country on mobile devices, risking exposure at checkpoints and border crossings [23]. Similarly in 2012, a videographer smuggled evidence of human rights violations out of Syria by hiding a micro-SD card in a wound on his arm [21], again risking his life.

Plausibly-deniable storage systems (PDS) allow users to to plausibly deny the existence of certain stored data even when an adversary has access to the storage medium. This is a key tool in the fight against powerful coercive adversaries and is vital for human rights activists, whistleblowers in oppressive regimes, government agents, law-enforcement officials etc.

An example of a PDS is the successful, yet unfortunately now-defunct TrueCrypt [8]. TrueCrypt divides a disk into multiple "password-protected" volumes and allows some of these volumes to be "hidden" in order to store sensitive data. Password-derived encryption keys are used to encrypt each such volume. Upon coercion, a user can plausibly deny the existence of a hidden volume by simply providing a valid password for one of the non-hidden ones, thus showing a plausible use for the disk without revealing the hidden data.

However, TrueCrypt is not secure against adversaries that can access the user's disk at multiple points in time ("multi-snapshot"). Most realistic adversaries

***Corresponding Author: Chen Chen:** Stony Brook University, E-mail: chen18@cs.stonybrook.edu
Anrin Chakraborti: Stony Brook University, E-mail: anchakrabort@cs.stonybrook.edu
Radu Sion: Stony Brook University, E-mail: sion@cs.stonybrook.edu

are ultimately multi-snapshot. Crossing a border twice, or having an oppressive government collude with a hotel maid and subsequently a border guard, provides easy and cheap multi-snapshot capabilities to any adversary. It is obvious that the security of a plausible deniable system should be resilient to and not break down completely in the persence of such realistic externalities.

To protect against multi-snapshot adversaries, several recent systems [12, 14, 15, 23] have proposed mechanisms that plausibly "explain" all device state changes via public data operations, in effect protecting access patterns to hidden data.

Yet, all existing PDS feature a critical shortcoming that renders them insecure in practice and may subject their users to "rubberhose" attacks: *the system is unable to hide the fact that a special storage mechanism is being used which potentially allows the user to hide data.* In particular, the presence of system-specific metadata e.g, special indexing structures stored on disk to track hidden information [12, 14, 15] or other design artifacts such as non-standard on-disk data layout [23], ultimately reveal to the adversary that the user is using a plausible deniable storage system. However, the use of a PDS can in itself raise suspicion – since why would one use such a storage system especially if it is slower than existing standard systems – unless there is something to be hidden.

Motivation. To mitigate this obvious drawback, we explore a new class of plausibly-deniable storage systems, called *invisible* PDS, which not only hides data but also all artifacts and telltale signs of the plausible deniability mechanism – in effect this does not allow the adversary to learn that a plausible deniability mechanism is in use for potentially hiding data.

This paper presents the design and implementation of INFUSE, a full-fledged invisible plausibly-deniable file system for flash devices. INFUSE enables a user to store public data and a small amount of hidden data in the same storage disk. Importantly, the on-disk data storage and access mechanisms in INFUSE do not reveal to a multi-snapshot adversary that a PDS is being used to potentially hide data. This is achieved by ensuring that INFUSE-related on-disk data structures, metadata and I/O look identical to a standard file system without plausible deniability – on coercion the user can simply claim that a standard file system is being used to store public data.

INFUSE is based on several key insights –

Hiding Information in Hardware Side-Channels. The first and foremost requirement of a PDS is to ensure that all changes to the disk between snapshots can be attributed to only public data state changes. Storing and performing operations on hidden data while only making observable changes to public data between snapshots is challenging, since it requires completely hiding the presence data in information channels that are not observable to the adversary. From an information theoretic point of view, one way to achieve this is to store hidden information in an out-of-band information channel.

To this end, in INFUSE, hidden data and metadata are completely stored in flash-based hardware side channels [25, 26]. These side channels operate by manipulating parameters such as *program time*, *threshold voltage* etc. of a user-chosen group of cells to store additional (hidden) information. In INFUSE we leverage the threshold voltage side channel [26], which exploits indistinguishability between operating a flash cell in single cell (SLC) mode and multi-cell (MLC) mode.

Securely Managing Co-resident Public and Hidden Data. Hiding data in out-of-band channels is not enough – if the presence of hidden data data affects public data, metadata etc. stored on-disk, or the data access mechanisms and bookkeeping operations e.g., garbage collection appear non-standard to the adversary, then this would clearly be a telltale sign. To achieve this INFUSE smartly re-designs mechanisms for typical file system functionalities e..g., page allocation, garbage collection etc. while ensuring that hidden data management does not affect on-disk public data:

1. Public data and metadata is not affected by the presence of hidden data. This also entails ensuring that storing and operating on hidden data does not compromise integrity of public data.
2. The public data access mechanisms are not affected by the use of the side channel.
3. Hidden data in the side channel is not destroyed by public data operations or other bookkeeping operations e.g., garbage collection.

Achieving Compatibility With a Standard File System. For invisibility, we need to ensure that all INFUSE-specific on-disk metadata and design artifacts resemble a standard file system – on coercion the user can claim that a standard file system is being used to store public data.

To achieve this, INFUSE incorporates design features from YAFFS [7], a widely deployed Linux file system for flash devices. INFUSE partitions (even in the presence of hidden data) have identical on-disk layouts to YAFFS partitions. Also, all bookkeeping operations

and data access patterns in INFUSE INFUSE look identical to YAFFS. This also allows INFUSE partitions to be mounted with YAFFS without any modifications. Specially, the same disk partition can either be mounted with INFUSE to reveal both public and hidden data stored in the side channel, or with YAFFS (when coerced by an adversary) to reveal only public data.

INFUSE has been implemented and evaluated on simulated flash chips, which support manipulating threshold voltages to desired values for storing additional bits per cell. Public data operations in INFUSE are less than 15% slower than a standard non-PDS baseline, which are faster than existing plausible deniability systems against multi-snapshot adversaries. Hidden data operations are of the same order of magnitude as those in existing plausible deniability systems.

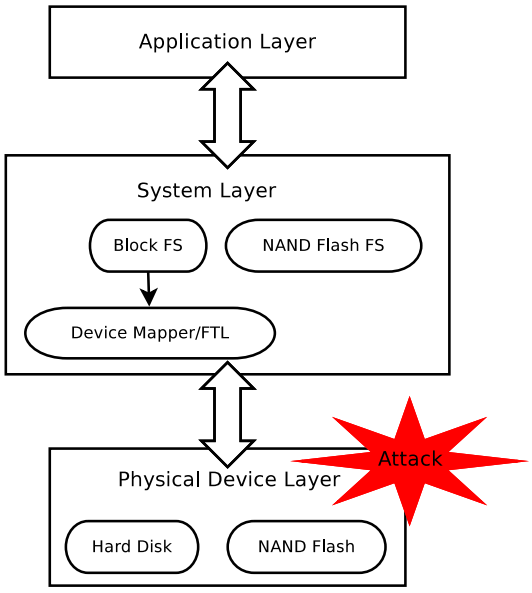

Fig. 1. Architecture of a storage system. The adversary considered in this paper targets the physical device layer for user's sensitive data.

2 Related Work

Plausibly-deniable encryption (PDE) was first explored by Canetti et al. [13]. PDE allows a given ciphertext to be decrypted into multiple plaintexts, by using different keys. The user reveals the decoy key to the adversary when coerced and plausibly hides the actual content of the message. Some examples of PDE enabled systems are [11, 18]. Unfortunately, PDE schemes are usually practical for only short messages and not very suitable for data storage.

For storage devices, Anderson et al. explored the idea of steganographic file systems in [10]. McDonald and Kuhn [20] implemented a steganographic file system for Linux on the basis of the solution proposed in [10]. Pang et al. [22] improved on the previous constructions by avoiding hash collisions and more efficient storage.

The aforementioned steganographic filesystem only defend against a single-snapshot adversary. Han et al. [17] designed a Dummy-Relocatable Steganographic (DRSteg) filesystem where multiple users can share the same hidden and runtime relocation of data ensures deniability against a multi-snapshot adversary. However, this does not scale well to practical scenarios. The only steganographic file systems that protects against multi-snapshot adversaries and scales well in practice is DEFY [23], a file system for flash devices that offers plausible deniability by leveraging secure deletion of data.

At device-level, disk encryption tools such as Truecrypt [8] and Rubberhose [18] provide deniability but cannot protect against a multi-snapshot adversary. Mobiflage [24] also provides PD for mobile devices against a single-snapshot adversary. One way to achieve plausible deniability against multi-snapshot adversaries at the device level is to use an oblivious RAM (ORAM) [12, 14]. However, due to the overheads introduced by the ORAM (mainly due to randomization of access patterns), these solutions are not practical. Recently, Chen et al. [15] introduced a locality-preserving plausible deniability solution named PD-DM which eliminates the randomness introduced by ORAM-based solutions and significantly increases overall throughput.

3 Model

Plausibly-Deniable Storage. The data stored on storage devices is usually managed by system software, e.g. a file system. A PD storage system comprises of both the software and the underlying hardware (Figure 1), and provides stronger protection for sensitive data than disk encryption. In INFUSE, the goal is to eliminate all evidence that a PD storage system is being used to hide data. Thus, this reduces suspicion and the likelihood of "rubberhose attacks".

3.1 Threat Model

In this paper, we consider adversaries that only probe the physical device (Figure 1) for sensitive data. This

corresponds to realistic scenarios e.g., officials scanning devices at border crossings. As with previous work, the running state of the device and the contents of the DRAM/caches etc. cannot be observed. This is a reasonable assumption since the multi-snapshot adversary can access the device only after an unmount when the in-memory contents will be cleared. Thus, protecting against a *runtime adversary* e.g., an adversary that can install malware/spyware to track user accesses requires stronger, complimentary protection mechanisms.

On-Event Multi-Snapshot Adversary. The typical PD storage adversary is considered to be computationally bounded and "rational" [12, 14]. The amount of information that an adversary can learn about the storage system is related to the frequency of observation. The adversarial model considered in this paper has two distinct properties:

- *Multi-snapshot:* The adversary can inspect the user's device multiple times and compare device "snapshots". The adversary can also perform forensics or statistical tests etc., e.g., by probing the threshold voltage levels of individual NAND flash cells.
- *On-event:* The adversary can access the device only after specific "events" such as disk "unmounts", graceful power-downs etc. This is a reasonable assumption since the multi-snapshot adversary will be able to obtain the device only after the user hands over the device e.g., at border crossings.

As with all existing PD solutions, INFUSE does not protect against denial of service attacks. Specifically, in INFUSE, the adversary can overwrite all data on disk thus destroying hidden data (if any).

3.2 Solution Space

A PD storage system can provide different levels of deniability when coerced by an adversary. This categorizes the solution space based on the components that the solution effectively "hides" from an adversary:

- *Hiding data contents:* Typically, all PD storage systems hide contents of sensitive data by encrypting it and claiming that the data is either random data or remnants of deleted public data.
- *Hiding access patterns:* Hiding contents is not enough to explain accesses to hidden data between snapshots. This is because random data should not change due to only public accesses. As shown in [12, 14], achieving plausible deniability against a multi-snapshot adversary also requires hiding accesses patterns.
- *Hiding PD mechanism:* Hiding the presence of any PD-related component in the system from adversaries is critical in real life. Although we can find various excuses for using a PD system, e.g. for full disk encryption, it is still suspicious that a storage system contains PD-related components. In other words, merely the use of a PD storage system is suspicious enough such that adversaries will not stop asking for more encryption keys.

Table 1 compares existing PD storage solutions based on their security guarantees. As listed, none of the existing works hide the existence of a PD solution from a multi-snapshot adversary. *INFUSE is the first solution to provide deniability of all the three aspects listed above against multi-snapshot adversaries, and is thus more secure for real world deployment.*

4 Building Blocks

4.1 NAND Flash Device

Unlike the traditional magnetic disk that stores data by magnetizing the ferromagnetic material on a disk, NAND flash stores data with a large array of floating-gate cells. Cells constitute the basic unit of I/O in a NAND flash, while read and writes are performed at page-level granularity. However, a block-level erase operation is always required before writing a page. Flash memory can only withstand a finite number of program-erase (P/E) cycles.

Bit Storage. In NAND flash memory, a cell stores one or more bits based on the presence or absence of charge – one bit per cell for a single-level cell (SLC), and two or more bits per cell for a multi-level cell (MLC)). In particular, a specific *threshold voltage* is required to be applied to the control gate of each cell to make the transistor conductive. Charging (programming) a cell increases the threshold voltage, while "erasing" a cell removes the charge from the floating-gate and thus decreases the threshold voltage instead.

Reading bits stored in a cell is done by comparing the threshold voltage with certain predefined *reference voltage levels* – $2^k - 1$ levels for a cell storing k bits. Instead of programming the threshold voltage to specific

	Year of publication	Presence of data	Access pattern of data	PD-related component	Multi-snapshot
DataLair [14]	2017	Yes	Yes	No	Yes
DEFY [23]	2015	Yes	No	No	Yes
HIVE [12]	2014	Yes	Yes	No	Yes
TrueCrypt [8]	2004	Yes	No	No	No
StegFS [20]	1999	Yes	No	Yes	No
StegFS [13]	1997	Yes	No	No	No
INFUSE	–	**Yes**	**Yes**	**Yes**	**Yes**

Table 1. Comparison of existing PD solutions and INFUSE about security level.

values, bits are stored in cells by controlling the threshold voltage in certain voltage windows. By changing the threshold voltage of a cell or the reference voltage levels, the bit(s) stored in a cell can be interpreted differently. For example, an SLC with threshold voltage $V_{th} = 3V$ will be interpreted as bit "1" when the reference voltage level is $V_r = 3.5V$. And its logical value will become "0" if the reference voltage level is changed to $V_r = 2.5V$ or the threshold voltage is increased to $V_{th} = 4V$.

With advances in technology, NAND cells can be programmed more accurately, resulting in more fine-grained threshold voltages. In fact, some recent flash controllers are able to operate the same cell in both SLC and MLC mode. This allows the design of a flexible flash file system that performs comparably with SLC devices and has better storage capabilities [19].

Hiding Bits in Flash Cells. The ability to program and operate the same cell as an SLC or an MLC provides an opportunity to hide bits in a flash device – *multiple bits can be stored in a particular cell using an "MLC-style" encoding but on inspection claim that the cell is in SLC mode and stores only a single bit.* Obviously, the prerequisite in this case is the SLC mode of operation should be indistinguishable from the MLC-style mode with respect to the variable parameters of the cell, e.g., threshold voltage etc. In particular, storing hidden bits in NAND flash cells may result in modification of threshold voltages in the respective cells. Fortunately, the inherent variability of threshold voltages in NAND flash memories can effectively "cloak" these variations.

In particular, due to variations during the manufacturing process, all cells storing the same logical bit in a flash device will not have the exact same threshold voltage. Instead, the threshold voltage varies among cells in a single page as well as across pages. This means that threshold voltage levels in blocks always have manufacturing-related noise that are unpredictable.

As a workaround, flash devices are designed to tolerate some *noise* while interpreting the threshold voltage in a cell as bits. Thus, it is possible to explain the bias of threshold voltages resulting from extra bits stored in a cell as unavoidable noise. Of course, the bias cannot exceed the scope of variation.

Zuck et al. [26] use this idea to hide bits in NAND flash devices by manipulating the threshold voltage of randomly chosen cells. They propose VT-HI and experimentally verify that a device with hidden bits appears indistinguishable from a device without hidden bits. As a result, by simply probing the NAND flash device, an adversary will not be able to identify how many bits are actually stored in each cell.

4.2 YAFFS

YAFFS [7] is an open-source highly portable and robust *log-structured* flash file system. It typically outperforms most alternatives [3] and was used as the default file system in Android before 2011, in addition to many other systems. The unit of read/write is called a chunk in YAFFS, which usually corresponds to a page in NAND flash hardware. Each page contains an OOB (out-of-band) area, which can be used to store the related ECC and some other meta information for a chunk. Everything in YAFFS is an object identified by a unique integer object ID. Objects constitute regular data files, directories, hard links, symbolic links etc.

There are two modes: YAFFS1 and YAFFS2. The main difference lies in two aspects. First, YAFFS1 supports page size of less than 1KB while YAFFS2 allows larger pages. Second, YAFFS1 deletes a chunk by overwriting its deletion marker while YAFFS2 doesn't need any overwrite and only writes sequentially. The rest of this paper refers only to YAFFS2 unless otherwise stated as it is more widely used.

Structures. YAFFS writes data in the form of a sequential log, and the entry of the log is a chunk. This inherently spreads out the wear. Chunks are written in strict order in one block, and blocks are assigned with monotonically increasing sequence numbers at writing time. There are two types of chunk in YAFFS: data chunks and object headers. Each chunk has tags associated with it. The tags comprise several fields such as ObjectID, ChunkID and SequenceNumber. The tags indicate – i) which object a chunk belongs to, ii) where a chunk is within a file, and iii) which chunk is the current one.

Each object in YAFFS has an object header which is a descriptor for the object. It contains object details such as the identifier for the parent directory, the object name etc. Data file objects also include a few data chunks. When a data chunk is updated, the corresponding object header is updated at the same time. With object headers and tags, no file allocation tables or similar structures are needed and the state of file system can be recreated regardless of the placement of chunks in NAND. It improves the performance and keeps the robustness of the file system.

In addition to chunks on the device, YAFFS also maintains several in-memory data structures which store information about blocks, objects, file structures, directory structures etc. This improves the system performance dramatically as metadata accesses can be performed with minimum latency.

Garbage Collection. As a log-structured file system, YAFFS performs chunk updates out-of-place. Thus, the storage space is used up gradually since a page can not be overwritten before the block it belongs to is erased. To overcome this, YAFFS tracks outdated pages and erases them accordingly in block granularity to reclaim space. This process, called garbage collection, copies useful chunks in one block to somewhere else and erasing the block for future use.

There are two heuristic ways to determine which block will be garbage collected next in YAFFS: *passive* and *aggressive* garbage collection. Passive garbage collection happens when there are still enough erased blocks available. In this case, only blocks with very few chunks in use will be garbage collected to minimize data copies. The whole process can also be spread over many garbage collection cycles to reduce load and improve system responsiveness. Aggressive garbage collection is performed when the available erased blocks are fewer than a particular threshold. In this case, the "dirtiest" block has to be collected during one garbage collection cycle regardless of the number of chunks in use in that block.

Mounting YAFFS. During a mount, YAFFS scans the entire partition unless a previous checkpoint is available. Generally speaking, the system performs a pre-scan followed by a backwards-scan. A pre-scan reads sequence numbers for all blocks and then sorts them chronologically. Then the tags of all chunks are read in reverse chronological order to rebuild the in-memory data structures in the backwards-scan stage. A checkpoint mechanism can speed up mounting where a checkpoint records the YAFFS runtime state in a few blocks on the disk during an unmount of a sync. YAFFS searches for the most recent checkpoint and rebuilds the file system state based on it during subsequent mounts.

5 Challenges

As discussed above (Section 4.1) NAND flash devices provide additional "out of band" information encoding opportunities. It is feasible to undetectably manipulate threshold voltage levels of randomly selected flash cells to encode additional bits at the physical level [26]. The resulting set of variations in cell voltages is indistinguishable from random noise to an adversary that does not know the exact locations of manipulated cells.

It is important however to manage any such hidden bits securely through upper layers (e.g., file systems) with careful consideration to avoid leaking information through inter-layer interactions or from any metadata.

It is the object of this work to take this basic cell-level encoding mechanism and build a file system that is "invisible" (identical layout with standard file system) and efficient, provides redundancy, handles overwrites, survives data loss, and is secure in the presence of multi-snapshot adversaries. This is non-trivial and faces a number of challenges which we discuss in the following.

– *Hiding the hidden bit-embedding process:* Even if the inherent variation in threshold voltages protects hidden bits, for a usable file system solution it is essential that the process followed to "embed" hidden bits in cells is protected from the adversary.
INFUSE hidden bits are stored at randomly selected locations in a block, ensuring that changes in threshold voltage distribution is distributed uniformly across the device data and an adversary cannot locate "hot patches" – a specific group of cells

with a notably high density of hidden bits. Hidden bits in one physical block are organized as virtual hidden pages and written at page granularity. The increase of cell voltage that an on-event multi-snapshot adversary can observe is always the result of both public bits programming and hidden bits programming, which can always be explained as a result of writing public bits. (More details in Section 6).

– *Hiding file system metadata:* To track updates to data, file systems typically store additional metadata (directory trees, inodes, time stamps etc.). To ensure plausible deniability, more metadata may be required. Metadata for hidden files cannot be visible to the adversary in a PD file system.
  INFUSE manages metadata by storing chunk-specific *tags* along with the chunk itself (as in YAFFS), and uses in-memory directory-structures. For plausible deniability, the tags can be easily hidden in the same way as the chunks themselves, while the in-memory data structures are not accessible to multi-snapshot adversaries.
– *Decoupling file system operations from hidden data:* Except for I/O on public data, flash-specific file systems also preferably perform additional essential bookkeeping operations such as garbage collection. These operations should not depend on the amount/content of hidden data.
  INFUSE ensures that garbage collection is not affected by the presence of hidden data. Specifically, the choice of garbage collected blocks is unrelated to the hidden files that use it. Hidden data is relocated during garbage collection similar to public data in the collected pages.
– *Ensuring hidden data integrity:* Hidden bits are stored in the same locations with public bits using the cell-level bit-embedding mechanism. As the file system operations are decoupled from hidden data, we must carefully incorporate mechanisms (without sacrificing security) to avoid hidden data loss due to operations to public data.
  INFUSE ensures this by including margin hidden pages and data duplicates. Margin hidden pages avoid hidden data loss during garbage collection while data duplicates prevent hidden data loss after mounting the device as YAFFS.
– *Managing Publicly-Visible Software Components:* Although, a multi-snapshot adversary typically only has access to the storage device, in certain cases, it may be the case that the adversary also inspects the installed software. In that case, the presence of the INFUSE code in the system software stack will obviously leak that a special file system is being used to manage hidden data.
  This is why, to ensure privacy, we design INFUSE in such a way that the only change to the kernel software is the addition of the INFUSE kernel module, which can be inserted and removed on demand. The user should uninstall and remove the module and traces of its installation when there is a possibility of encountering a coercive adversary e.g., when crossing borders. Note that even without the INFUSE module, the public data is perfectly accessible as a standard YAFFS partition.

6 INFUSE: Detailed Design

INFUSE is a NAND flash file system supporting two security levels: public and hidden. The files that can be disclosed to adversaries are stored as public files, while the files that need to be protected from adversaries are stored as hidden files. Hidden files are stored using the bit-embedding technique described in Section 4.1. Specifically, hidden bits are encoded in a tuning of the threshold voltage of a cell containing public data. As we discuss later, over time, for a given addressable hidden bit, this public cover data can change (due to garbage collection) and the bit may end up elsewhere.

INFUSE is designed based on YAFFS, an open-source file system specifically designed for raw NAND and NOR flash and widely used in embedded systems. Similar to YAFFS, INFUSE is also a log-structured file system. Two logs are maintained in INFUSE for the two security levels respectively. Files are stored as *objects* on the device, similar to YAFFS. Public files are stored as they would be in a YAFFS partition, while hidden files are stored using the bit-embedding technique discussed in Section 4.1. Hidden files are replicated throughout the device for redundancy, so that data integrity can be preserved even if some pages are overwritten. INFUSE needs a password from users do derive keys for the encoding and encryption of the hidden data.

In addition to the data on the device, INFUSE also includes a few data structures in memory. The relationship between INFUSE and YAFFS is illustrated in Figure 2. The layout of INFUSE on the flash device appears indistinguishable from a YAFFS partition. In other words, INFUSE partitions are compatible with YAFFS partitions. Whenever a INFUSE partition is mounted as a YAFFS, only operations to all non-hidden

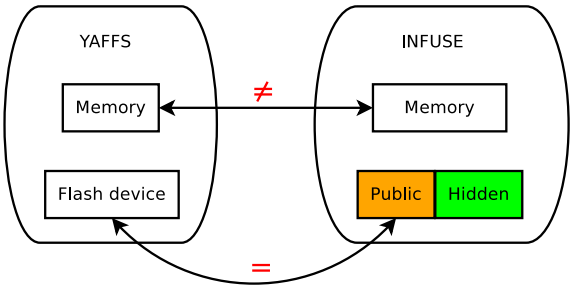

Fig. 2. INFUSE vs. YAFFS. Both file systems are composed of in-memory data structures and data on device. The data organization in the underlying flash device is identical regardless of the file system used. However, the in-memory data structures are different.

files on the device are supported. For the sake of security, a user can unmount the device and uninstall the INFUSE software completely from the system after sensitive data has been written. An adversary in possession of the device later will be misled to believe that this is just a YAFFS partition with only public files inside. The protected data can be recovered by mounting the device with INFUSE again in the future.

6.1 Layout

In order to use the bit-hiding technique for NAND flash devices (Section 4.1), INFUSE is equipped with an "intermediate layer" to translate operations on public and hidden data into operations on physical pages in which the hidden data is "embedded". This is detailed below.

Objects and Chunks. INFUSE stores all data as objects, similar to YAFFS. Objects corresponding to the data in the hidden level are called *hidden objects* while all other objects are *public objects*. Specifically, a hidden file consists of a hidden object header chunk and a few hidden data chunks, while a public file consists of a public object header chunk and a few public data chunks.

Each chunk in INFUSE has a relevant tag which records the meta information about the chunk, similar to tags in YAFFS. The size of the tag in INFUSE is 42 bytes. Hidden chunks and their corresponding tags will be encrypted with authenticated encryption (AES-GCM) before being written. Thus, INFUSE can verify whether the data in a hidden chunk has been tampered with. Further, each hidden chunk is associated with a 96-bit IV (initial vector) [16] and an 128-bit authentication tag after encryption. The hidden tag, and the corresponding IV and authentication tag are stored alongside the hidden chunk.

Public Page and Hidden Page. We introduce an "intermediate layer" in INFUSE between the logical file system layer and the physical layer, as shown in Figure 3. In the intermediate layer, public virtual pages and hidden virtual pages coexist in one block. For the sake of simplicity, we call them public pages and hidden pages in the rest of the paper. Without loss of generality, only one public page and one hidden page per block are presented in Figure 3.

Both public pages and hidden pages consist of a data area and a spare area (OOB area). The public page is the same size as a physical page in the flash device, while the size of a hidden page may vary in order to judiciously utilize the hidden bits that can be embedded in the physical device.

INFUSE stores public chunks in public pages and hidden chunks in hidden pages, respectively. Typically, a flash memory with 2KB pages (2KB of data area) has an OOB area of 64 bytes per page. A hidden page in this case will typically have a data area of the same size (2KB) and an OOB area of 70 bytes. The size of the OOB area in a hidden page is the total size of a hidden tag (42B), a random IV (12B) and an authentication tag (16B). The size of the data area in a hidden page can be made smaller if required, based on the number of hidden bits that can be embedded in one physical block without violating security .

The example in Figure 3 illustrates how a public file and a hidden file are stored. Only one public page and one hidden page per block are presented in Figure 3 for simplicity.Each file is assumed to have only one data chunk for simplicity. The two chunks with ChunkID 0 are the public object header and hidden object header respectively. Chunks with the same Object ID belong to one object, and thus one file.

Moreover, the last hidden page in each block may be set to be smaller than the others if a block can have more than one hidden page. YAFFS uses a *block summary mechanism* to accelerate the mounting process. Specifically, the tags of all pages in a block are grouped together and stored in the last few (typically only one) page of that block. In this case, instead of reading the whole block for only the OOB areas, YAFFS only needs to read the last page of each block during scanning to rebuild the whole directory structure of the file system. As there are usually more than 64 pages in one block, the whole summary page is efficiently utilized. The hidden tags in one block cannot be written together with the public tags into the public summary page lest the

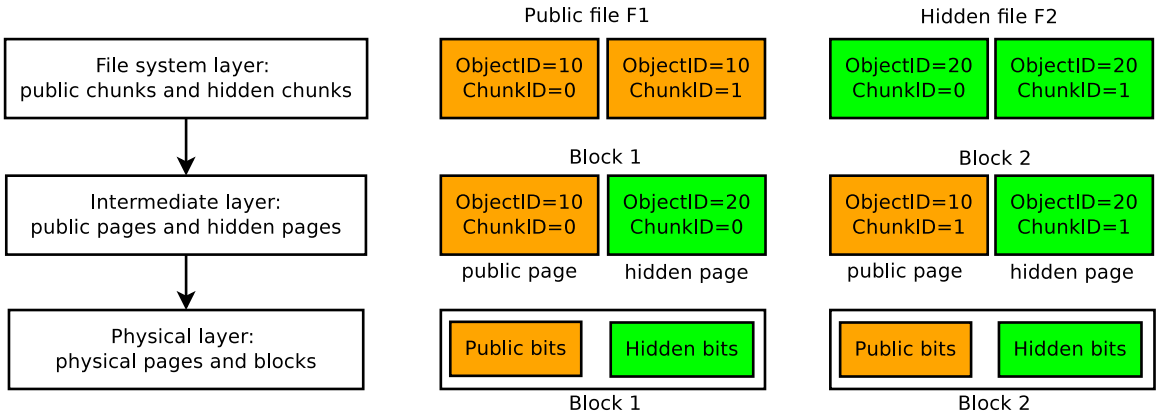

Fig. 3. The INFUSE infrastructure. Physical pages in flash devices are interpreted as public pages and hidden pages (used to store public files and hidden files respectively) in the intermediate layer. Similar to YAFFS, ObjectIDs and ChunkIDs are used to structure the file system regardless of the chunk locations.

existence of hidden pages is revealed. Instead, one hidden page needs to be used for hidden tag summary in one block. However, in INFUSE the number of hidden pages in one block is much less than that of the public pages. The space required to store the summary information of those hidden pages is much smaller than a whole page. Thus, in order to effectively utilize the "embedded" hidden bits in one block, we assign less bits to the summary hidden page. The summary function for the hidden pages can even be disabled by the users without any effect on the public page summary. In this case, none of the hidden bits in one block is used to store redundant information.

Physical Layout. Public pages and hidden pages in the intermediate layer are made up of the public bits and hidden bits in the device. In INFUSE, each physical page can now contain both public bits and hidden bits.

The number of public bits is the same as the number of cells in a physical page, while the number of hidden bits is restricted to a much smaller number. This is necessary to ensure that variations in threshold voltage as a result of hiding bits does not exceed the scope of inherent variations.

INFUSE organizes the hidden bits in one physical block as a few hidden pages depending on their number. The hidden page is aligned with the physical page, so that all the hidden bits in one physical page belong to one hidden page and will be programmed together. A parameter m denotes the number of physical pages that are needed to store all the hidden bits in one hidden page. For example, if one block contains K physical pages of size 2112B (2KB+64B), p percentage of cells in each page can be used to encode 2 bits. Consequently, a hidden page is spread into $m = \lfloor 2118/(2112 * p/100) \rfloor$ physical pages. In other words, a block contains K public pages and at least K/m hidden pages (assuming that summary for hidden pages is disabled).

How INFUSE read and write a hidden page can be described with Algorithm 6.1 and 6.2 respectively. The two hardware-backed operations $\mathsf{read\_hidden\_bits}(\mathsf{M}, \mathsf{addr})$ and $\mathsf{write\_hidden\_bits}(\mathsf{M}, \mathsf{addr}, \mathsf{data})$ denotes read and write the hidden bits from one physical page respectively. Here, M represents a bitmask indicating the physical cells selected for storing hidden bits in a physical page. Specifically, hidden bits are stored in physical devices by manipulating the voltage of selected physical cells gradually through a series of up to k partial programming (PP) steps according to [26]. And hidden bits are read from physical devices by adjusting the reference voltage using the read retry command supported by latest NAND flash memory [26]. Importantly note that the locations of hidden bits are randomly selected – an adversary cannot identify the cells containing hidden data with more than negligible advantage without having access to the hidden key.

Algorithm 6.1 data=$\mathsf{read\_hidden\_page}(\mathtt{BA}, \mathtt{PA})$**: Read a hidden page in the intermediate layer**

Input: $\mathsf{read\_hidden\_bits}(\mathsf{M}, \mathsf{phy}_{\mathsf{addr}})$, m: number of physical pages across which a hidden page is stored
Ouput: *data*
1: $i = 0$
2: **while** $i < m$ **do**
3: $\mathsf{phy}_{\mathsf{addr}} = \mathtt{BA} \ll \mathtt{offset_b} + \mathtt{PA} * \mathsf{m} + \mathsf{i}$
4: Read public data d_{pub} in physical page $\mathsf{phy}_{\mathsf{addr}}$

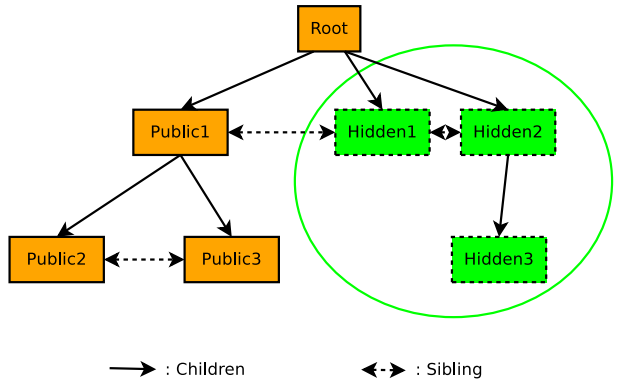

Fig. 4. The directory structure in INFUSE.

5: Generate bitmask M for cells storing hidden bit based on d_{pub} and $\mathsf{PRNG}(\mathsf{K_{hid}}, \mathsf{phy_{addr}})$
6: $\mathsf{d_i} = \mathsf{read\_hidden\_bits}(\mathsf{M}, \mathsf{phy_{addr}})$
7: $i = i + 1$
8: $data = \{d_0, d_1, ..., d_{m-1}\}$

Algorithm 6.2 $\mathsf{write\_hidden\_page}(\mathtt{BA}, \mathtt{PA}, \mathsf{d_{hid}})$**: Write a hidden page in the intermediate layer**

Input: $\mathsf{write\_hidden\_bits}(\mathsf{M}, \mathsf{phy_{addr}}, \mathsf{bits_{hid}})$, m: number of physical pages that a hidden page is spread into, l: number of hidden bits storing in one physical page

1: $i = 0$
2: **while** $i < m$ **do**
3: $\mathsf{phy_{addr}} = \mathtt{BA} \ll \mathtt{offset_b} + \mathtt{PA} * \mathsf{m} + \mathsf{i}$
4: Read public data d_{pub} in physical page $\mathsf{phy_{addr}}$
5: Select cells storing hidden bit based on d_{pub} and $\mathsf{PRNG}(\mathsf{K_{hid}}, \mathsf{phy_{addr}})$ and generate bitmask M
6: $\mathsf{write\_hidden\_bits}(\mathsf{M}, \mathsf{phy_{addr}}, \mathsf{d_{hid}}[\mathsf{i} \cdot \mathsf{l} : \mathsf{i} \cdot (\mathsf{l}+1)))$
7: $i = i + 1$

Directory Structure. Objects in different security levels form independent object sub-trees under the directory root in INFUSE, as shown in Figure 4. Thus, a public object will not have a hidden object as its parent, and vice versa. This follows typical security conventions such as file system permissions. As a result, public files are isolated from hidden files in INFUSE, thus ensuring minimal traces of hidden files in the public level. Note that, although an object contains information such as siblings which may link a public object with a hidden one, this information is maintained in memory only. An adversary cannot observe the in-memory data structures when an INFUSE partition is mounted, and thus will be oblivious to this information.

6.2 Page Allocation

Recall that INFUSE maintains two logs, each corresponding to a particular security level. The page allocation principle for public data in INFUSE follows the same principle as YAFFS to ensure that adversaries cannot infer the deployment of INFUSE from the page allocation mechanism. As a log-structured file system, the public pages in the current block of the log will be allocated one by one for public data. Then the current block will move to the next available block in the allocation pool.

For hidden pages, another log head is maintained to mark the current block for the hidden level. The hidden log head will always *follow* the public log head since the hidden pages cannot be programmed in a particular block until all public pages in that block are used. For example, if public files are written to physical block 1, 3, 4 in order, the hidden log will follow this order.

Figure 5 shows how the state of block transforms during runtime in INFUSE. The five states with solid border originate from YAFFS and the other two states in dashed boxes are specific to INFUSE:

- *Empty:* The block is empty and is ready for allocation.
- *Allocating:* The public pages in the block are currently waiting for allocation by the public chunk allocator.
- *Full:* The public pages in the block have been allocated.
- *Collecting:* The block is undergoing garbage collection.
- *Dirty:* Both the public pages and the hidden pages in the block do not contain any useful information.
- *Hid-allocating:* The hidden pages in the block are currently waiting for allocation by the hidden chunk allocator.
- *Hid-full:* Both the public pages and the hidden pages in the block are used up.

It is worth noting that a block may be garbage collected at full, hid-allocating or hid-full state since the garbage collection block selection is independent of the state of hidden pages in blocks in INFUSE (as described next). Moreover, the block state can change directly from full to hid-full and bypass the hid-allocating state. This scenario corresponds to when INFUSE is unmounted. As adversaries have access to the flash device after the partition is unmounted, a block in full state cannot go to hid-allocating state anymore. Writing the hidden pages in the block will change the threshold voltages of cells in the block and reveal the existence of hidden data to adversaries.

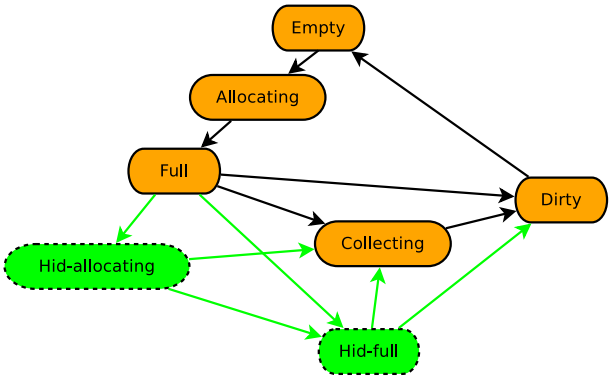

Fig. 5. The state transition diagram of block state in INFUSE. The two states with green background color and dashed border are new states in INFUSE and the other five states are the regular runtime states in YAFFS.

6.3 Garbage Collection

To ensure that an INFUSE partition looks indistinguishable from a YAFFS partition, the garbage collection heuristics in INFUSE is identical to the one used in YAFFS. Note that if garbage collection is affected by hidden data, then the choice of recycled blocks will reveal the existence of hidden data. This however presents several challenges.

First, garbage collection in YAFFS will always increase the number of available pages in the system since it chooses the dirtiest block with the fewest useful chunks. However, since in INFUSE garbage collection does not account for the hidden data in the blocks, there is no guarantee that the number of available hidden pages will increase after one garbage collection cycle, particularly when few public pages in the recycled block are still useful. In the worst case, the number of available hidden pages may even decrease if all the hidden pages in the recycled block are still useful.

To prevent running out of hidden pages and ensure hidden data integrity, INFUSE defines *margin pages* as all the hidden pages between the public log head and the hidden log head. A warning message will be sent to users once the number of margin pages is below a certain threshold. The user should stop writing more hidden files until enough margin pages are available again after a few rounds of garbage collection. Otherwise, the hidden data may be lost during garbage collection.

Second, a block that is recycled may contain valid hidden chunks. These need to be relocated before the block is erased. INFUSE achieves this as follows:

- When there are abundant margin pages, a valid hidden chunk in the recycled block will be copied to a new location during garbage collection, regardless of the number of copies that already exist. This effectively duplicates the hidden data.
- When the number of margin pages is below a certain threshold, INFUSE will copy only hidden chunks with no duplicates.

More details about the data relocation procedure are provided in Section 6.5.

6.4 Mount & Unmount

Mounting an INFUSE partition requires a password/encryption key from users. INFUSE searches for available checkpoint blocks for both public and hidden levels at first. As the location of hidden checkpoint is not recorded, INFUSE tries to decrypt every hidden page to locate the checkpoint. If both checkpoints are found, rebuilding the file system state is straightforward. Otherwise, INFUSE scans the entire partition to rebuild state, similar to YAFFS.

Unmount in INFUSE is more complicated since it needs to "close" the hidden level and hide any evidence of its use. First of all, INFUSE performs an "enforced garbage collection". This ensures that the hidden files in the INFUSE partition will not be destroyed when the partition is later mounted as a YAFFS partition.

Secondly, INFUSE writes all the available hidden pages in the hidden allocation pool with backup hidden chunks. This is detailed in Section 6.5.

Note that the hidden pages in the public checkpoint blocks are not written in INFUSE. This is because that the public checkpoint blocks will be dirty as soon as any writes are performed after mounting the device again. Instead, INFUSE writes hidden checkpoint to margin pages. The intuition here is that the probability that the hidden checkpoint will remain intact after a few YAFFS writes is high since the checkpoint resides in the *most recently written* block, which is expected to contain up-to-date data and is unlikely to be overwritten soon.

6.5 Data Loss

The flash device will be mounted as a YAFFS image whenever it is under surveillance of an adversary. Thus, if a garbage collection is performed after mounting, it is possible that the hidden data residing in the collected block will be destroyed when the block is erased. Furthermore, the choice of erased blocks depends only on

the public work load when the device is mounted as YAFFS. Thus, it is impossible to ensure the integrity of all hidden data. However, we can take measures to improve the hidden data survival rate.

Our strategy to reduce the data loss is to make duplicates of hidden chunks during unmount. Remember that there are always several empty hidden pages that are maintained in the hidden allocation pool (Section 6.2). These pages cannot be used anyway once the partition is unmounted, to prevent the adversary from detecting the voltage threshold changes resulting from hidden chunk programming. Thus, we make duplicates for the hidden chunks to improve the hidden data survival rate.

There are several rules for this data duplication. First, the duplicates of the same chunk should not be relocated in one block since it will not help reducing the data loss. Secondly, the hidden chunks in the block with the smallest public sequence number should have a higher duplication priority. Although the block chosen by garbage collection depends on the public workload that invalidates the public chunks, it is also related to the block sequence number as YAFFS mandatorily garbage collects the oldest block. Thus, the hidden chunks in blocks with smaller public sequence numbers have higher risk of being erased and should be backed up somewhere else.

A back up chunk has the same chunk ID and object ID as its original chunk. If more than one copy of certain chunk exists when the partition is mounted as INFUSE, the one with the largest sequence number will be considered during the device scan to build state.

6.6 Key Management

To detect hidden data tampering, INFUSE deploys authenticated encryption AES-GCM with 96 bit IVs and 128 bit authentication tags. Whenever a hidden chunk is relocated (for either garbage collection or data redundancy), the chunk is re-encrypted with a new randomly assigned IV.

INFUSE requires two keys that are derived from a user password. One is used to encrypt the hidden data. The other key is used to generate random numbers with the block sequence number and page number. The generated random numbers are used to select cells in a page to store the additional hidden bits. Note that public data is not stored encrypted (similar to YAFFS) but additional support can be added if required without affecting the hidden data management..

7 Security Analysis

As detailed in Section 3, PD storage solutions can be categorized by what they hide in practice. This includes (i) *content of hidden data*, (ii) *access pattern of hidden data*, and (iii) *the evidence of deployment of the PD system*. INFUSE is the first PD storage solution that hides all three elements from multi-snapshot adversaries. In the following, we summarize the security guarantees in INFUSE and leave formal proofs for future work.

Hiding Contents of Hidden data. To hide contents, INFUSE leverages the bit-technique proposed in [26] (as detailed in Section 6). Specifically, [26] shows it is feasible to undetectably manipulate threshold voltage levels of randomly selected flash cells to encode more than one bit. The resulting set of variations in cell threshold voltages is indistinguishable from random noise to an adversary that does not know the exact locations of manipulated cells. Effectively, a flash device storing hidden data appears indistinguishable to an adversary from an off-the-shelf flash device. INFUSE stores hidden bits in randomly selected location based on a keyed cryptographic hash – for a computationally-bounded adversary identifying cells which store hidden data will be as difficult as inverting a cryptographic hash.

Further, in INFUSE, hidden data is stored encrypted with a semantically secure encryption scheme. This ensures that even if an adversary assumes that all cells are being used to store hidden bits and uses fine-grained reference voltages to interpret data (the reference voltages used for "embedding" hidden bits may be known to adversaries), the semantic security of the encryption scheme will ensure that the contents appear as random data.

Hiding Access Patterns. To hide access patterns INFUSE ensures that all file system specific operations (e.g., garbage collection etc.) are not affected by the presence of hidden data. The design features that enable this in INFUSE are listed below.

- **Log-structured design:** INFUSE is a log-structured file system similar to YAFFS. The order in which public pages are written is identical regardless of the presence of hidden data. The hidden pages in each block will be written only if the public pages in that block have been written after the *latest* mount operation (each mount/unmount may correspond to an adversary gaining access to a device snapshot). Thus, writing hidden pages has no effect on the order in which blocks get written.

– **Hidden-data-independent garbage collection:** As detailed in Section 6.3, the selection of blocks for garbage collection in INFUSE does not depend on the state of hidden pages.
– **Attributing voltage changes to plausible public operations:** For a multi-snapshot adversary with the capability to take snapshots of the device state after any unmount operation, any change in the threshold voltage of flash cells can be detected. Threshold voltage changes can result from a public page write, a hidden page write, or an INFUSE garbage collection operation.
For security, these changes should be attributable to *only* public data writes and garbage collection operations. Since hidden data is written to blocks that have been written with public data after an INFUSE partition is mounted, threshold voltage changes can always be attributed to public data writes.

Hiding evidence of PD system. To hide evidence that INFUSE is being used, users can access the device public data using a standard YAFFS driver under duress. INFUSE page allocation and garbage collection principles ensure a partition layout identical to the layout of a standard public data YAFFS partition.

Also all file system operations in INFUSE remain unaffected by the presence of hidden data (as discussed above) and, from the point of view of a multi-snapshot adversary, they are performed identically as in the case of a YAFFS partition with only public data. As a result, the device data layout and associated patterns do not allow an adversary to distinguish between an INFUSE partition with hidden data and a YAFFS partition containing public data only.

Note that even if an adversary tries to mount a YAFFS partition with INFUSE (suspecting that a user has INFUSE installed), the mount will fail. In particular, to mount, INFUSE requires a secret key. Among other things, the key determines where hidden data resides. It is also used for encryption and integrity checks. Given a key, INFUSE will first try to decrypt the latest hidden checkpoint and verify integrity. If this fails, all blocks will be checked for hidden data, and if no data is found/the tags do not match, the mount fails. If an adversary attempts to mount a YAFFS partition (or an INFUSE partition with the wrong key) the INFUSE mount will fail allowing the user to claim that INFUSE is not being used.

8 Evaluation

8.1 Setup

Hardware. The process of embedding bits in flash chips [26] requires two hardware primitives – i) partial programming, and ii) read retry. Partial programming allows small adjustment of threshold voltages at page-level granularity. Read retry is a reference voltage control mechanism implemented in modern NAND flash chips. It is primarily used for bit error correction. Importantly, since these primitives are not specifically designed for hiding data, using a flash chip with such capabilities does not raise suspicion.

These operations are supported by multiple off-the-shelf flash chips (e.g., Hynix 1x nm NANDs, Samsung 3D NANDs). The granularity of the supported operations determines the amount of hidden data that can be stored. Moreover, the interfaces required to use these primitives are available in standard OS installations (e.g., provided by the MTD in the Linux Kernel).

However, one hurdle to overcome while implementing INFUSE is receiving exact read-retry specs (usually not public) from chip manufacturers. And despite extensive efforts, we have not been able to convince the hardware vendor to release the voltage specs and associated partial programming instructions under an NDA which allows us to publish results. On the chips we have received confidential instructions for, we have confirmed partial programming and read retry indeed work.

Therefore, to benchmark INFUSE in absence of the NDA, we modelled the timing and behavior of a commercially available flash chip (Hynix 1x nm NAND) in nandsim [4], a NAND flash simulator using underlying storage medium such as DRAM or a file. Additionally, we modified nandsim to simulate partial programming and the storage of embedded hidden bits. As an added benefit, using nandsim allows exploring the effects of hiding capacity (number of cells altered) on overall performance.

Corresponding to actual chip measurements, read, write and erase latencies are set to be 90us, 1200us and 5ms. The hidden bit programming latency is set to be 6900us. The average number of hidden bits in one physical pages varies from 512 to 8K, which results in realistic storage capacities. The NAND flash device size is set to 64GB to reflect a specific hardware chip's capacity. The page size is 4KB and the block size is 256KB. Experiments were run on Linux boxes with dual-core i5-3210M CPUs, 2GB DRAM, and kernel version 3.8.

Software. INFUSE has been implemented as a full kernel-space file system module for the Linux kernel. Typically, flash devices are recognized as Memory Technology Devices (MTD) in the Linux kernel unlike block devices. Standard Linux MTD implementations provide interfaces for partial programming and read retry – the two primitives required for INFUSE. Therefore, in order to manage hidden data with INFUSE, non-standard interfaces are not required. The only requirement is the INFUSE kernel module, which can be inserted (e.g., from a USB drive) and removed as needed.

8.2 Benchmark

Comparison with Existing Work. As discussed before, INFUSE works in a stronger (and also more realistic) security model, where the deployment of a PD system is considered a red flag for the adversary. To protect against this adversary, INFUSE hides not only the data but also evidence that a PD system is being used. All previous solutions unfortunately fail to hide the deployment of a PD system. Thus, comparing INFUSE with solutions that work in a strictly weaker threat model is not a fair comparison.

Moreover, comparing INFUSE with existing PD solutions experimentally is also problematic due to the different deployment settings. All existing PD solutions are either implemented as file systems equipped with the ability to "hide" data or block device mappers managing multiple volumes (some equipped with the ability to hide data). Block device solutions are typically optimized for SSDs [12, 14] and HDDs [15] and cannot be deployed on *raw* flash devices without additional firmware such as an FTL. Therefore, these solutions cannot be directly compared with INFUSE without additional instrumentation which may impact overall performance.

The only existing raw flash PD file system is DEFY [23]. However, its open-source implementation [1] can only be mounted on very small devices (≤ 64 MB). With such small devices, accurate performance is difficult to estimate, especially in the presence of OS-level caches.

Due to these limitations, we compare INFUSE against a YAFFS baseline with no plausible deniability guarantees to get a better idea of throughput overheads.

Throughput. We first benchmark both the sequential and random I/O throughput of both YAFFS and INFUSE with Filebench [2] on the same device simulated by nandsim. For this experiment, the number of hidden pages in one physical block is set to be 4.

It can be seen (in Figure 6) that reading public files in INFUSE is as fast as reading in YAFFS. Writing public files is slightly slower (no more than 15%) since INFUSE needs to take care of the hidden bits that are embedded in the device. Reading or writing hidden files sequentially is around 30-40 times slower than that of public files. As a hidden page is spread across several (16 in our setting) physical pages, it is not surprising that accessing the hidden files is much slower than accessing the public files. This throughput is in line with other existing PD systems with fewer security guarantees.

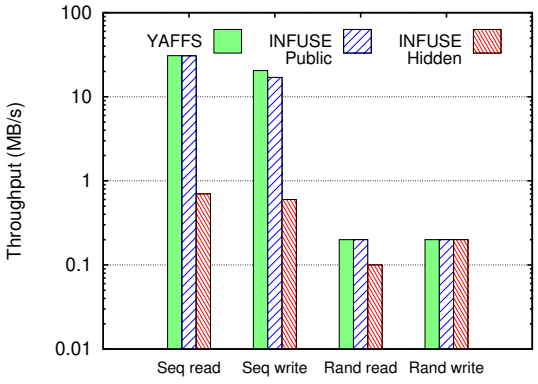

Fig. 6. INFUSE throughput comparison with YAFFS (presented in log scale, higher is better). The throughput is measured with Filebench. Throughput to the public files in INFUSE is almost the same or slightly slower than that in YAFFS, while throughput to the hidden files in INFUSE is 30 – 40 times slower.

Throughput vs. Number of Hidden Pages. We also explore the relationship between performance and the number of hidden bits stored in one physical block ("hidden data density"). Figures 7 and 8 show how throughput varies as the number of hidden pages in one physical block increases from 1 to 32. Since INFUSE also resiliently manages the (increasing number of) hidden bits when writing public data, public throughput drops with increasing hidden data density (Figure 7).

Hidden write throughput increases and is linearly correlated to hidden data density. This is not unexpected – as more hidden bits are stored in one physical page, less I/O is required to write one virtual hidden page to the device. However, increasing data density may also result in a device violating the conditions required for ensuring indistinguishability from a device without any hidden bits [26]. In fact, the range of allowable data densities is device-specific since different devices will have different security-performance trade-

offs for maintaining the required conditions for indistinguishability.

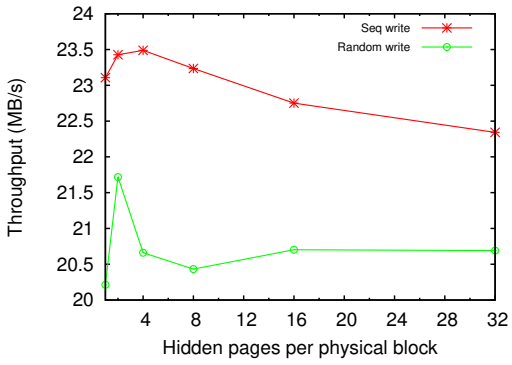

Fig. 7. Public INFUSE write throughput with increasing number of hidden pages per physical block (higher is better). Since INFUSE also resiliently manages the (increasing number of) hidden bits throughput drops with increasing hidden data density.

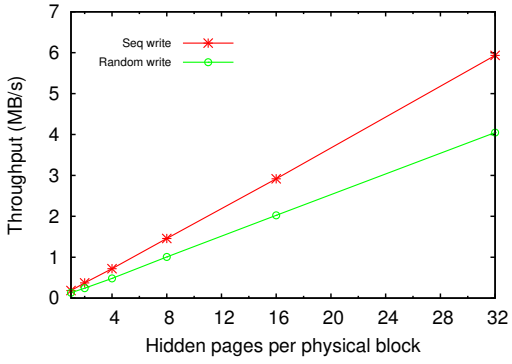

Fig. 8. Hidden INFUSE write throughput (higher is better). As expected, throughput increases with increasing hidden data density.

Hidden Data Storage Capacity. An important security parameter to determine prior to using a flash device to store hidden data is the amount of hidden information that can be stored. The parameter will be necessarily application and chip-specific, and depends on several factors:

1. *Accuracy of Partial Programming:* With more precise partial programming capabilities, more fine grained threshold voltage levels can be achieved, and thus more hidden data can be stored since small threshold voltage variations do not alter the voltage distribution enough to escape the acceptable noise band.
2. *Read Retry Capabilities:* In order to have more fine-grained voltage, the chip should also have better read capabilities to detect and interpret the voltages. This depends on the number of read retry modes supported by the chip.
3. *Reference Voltage Levels:* As noted in [26], the reference voltage used for hidden bits determines the number of bits that can be stored. A lower reference voltage implies more hidden bits can be stored without overly skewing the voltage distribution.

Keeping all these factors in mind, Zuck et al. [26] performed experiments to hide 256 bits per page and 2560 bits per page and evaluated the security implications. Overall, encoding capacity should be evaluated based on the partial programming and read retry capabilities, overall chip line characteristics, and acceptable noise band and associated risks.

9 Conclusion

INFUSE is a flash file system that is "invisible" (device layouts identical with that of a standard file system), provides redundancy, handles overwrites, survives data loss, and is secure in the presence of multi-snapshot adversaries. INFUSE is efficient and its public data operations are less than 15% slower than standard YAFFS. Hidden data operation throughputs are of the same order of magnitude as that in existing plausible deniability systems secure against multi-snapshot adversaries.

10 Acknowledgement

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. We would like to thank our shepherd, Alptekin Küpçü and the anonymous reviewers for their suggestions on improving the paper.

References

[1] Defy implementation. "https://bitbucket.org/solstice/defy/".
[2] Filebench. "https://github.com/filebench".
[3] Flash filesystem benchmarks 3.1. "http://elinux.org/Flash_Filesystem_Benchmarks_3.1".

[4] Memory technology devices. "http://www.linux-mtd.infradead.org/index.html".

[5] Missing thumb drive contains information about portland jetport employees, pilots. "https://www.pressherald.com/2016/06/21/missing-thumb-drive-contains-information-about-portland-jetport-employees-pilots/".

[6] Nasa breach update: Stolen laptop had data on 10,000 users. "https://www.computerworld.com/article/2493084/nasa-breach-update--stolen-laptop-had-data-on-10-000-users.html".

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

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

[9] Youth jailed for not handing over encryption password. "https://www.theregister.co.uk/2010/10/06/jail_password_ripa/".

[10] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *Information Hiding*, pages 73–82. Springer, 1998.

[11] D. Beaver. Plug and play encryption. In *Advances in Cryptology – CRYPTO'97*, pages 75–89. springer.

[12] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214. ACM, 2014.

[13] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable encryption. In *Advances in Cryptology – CRYPTO'97*, pages 90–104. Springer, 1997.

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

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

[16] M. J. Dworkin. Sp 800-38d. recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. 2007.

[17] J. Han, M. Pan, D. Gao, and H. Pang. A multi-user steganographic file system on untrusted shared storage. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 317–326. ACM, 2010.

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

[19] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim. Flexfs: A flexible flash file system for mlc nand flash memory. In *USENIX Annual Technical Conference*, pages 1–14, 2009.

[20] A. D. McDonald and M. G. Kuhn. Stegfs: A steganographic file system for linux. In *Information Hiding*, pages 463–477. Springer, 1999.

[21] J. Mull. How a syrian refugee risked his life to bear witness to atrocities. toronto Star Online, posted 14-March-2012, 2012. "http://www.thestar.com/news/world/2012/03/14/how_a_syrian_refugee_risked_his_life_to_bear_witness_to_atrocities.html".

[22] H. Pang, K.-L. Tan, and X. Zhou. Stegfs: A steganographic file system. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 657–667. IEEE, 2003.

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

[24] A. Skillen and M. Mannan. On implementing deniable storage encryption for mobile devices. 2013.

[25] Y. Wang, W.-k. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan. Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 33–47. IEEE, 2012.

[26] A. Zuck, Y. Li, J. Bruck, D. E. Porter, and D. Tsafrir. Stash in a flash. 2018.