INVISILINE: Invisible Plausibly-Deniable Storage

Abstract—**Powerful anti-encryption laws and adversaries who can coerce users to provide encryption keys and passwords weaken encryption's ability to protect sensitive information. Plausibly-deniable (PD) storage systems address this problem by enabling users to securely hide data and plausibly deny its presence when challenged. However, PD systems need specialized software that renders them detectable by suspicious adversaries questioning the very use of a PD system. To address this fundamental problem, we introduce and formally define the notion of plausible invisibility, preventing adversaries from determining whether a PD system was used in the first place. INVISILINEis a plausibly invisible system that formats and stores data in a manner compatible with existing off the shelf storage layers such as dm-crypt, a widely available block device encryption subsystem. Users can install INVISILINE binaries at runtime but in the presence of adversaries they can continue to access the public data using dm-crypt. Importantly, INVISILINE is invisible even for multi-snapshot adversaries that can see the device multiple times. INVISILINE can securely and invisibly hide 19GB on a 1TB disk with no impact on public data I/O, and an average of 4.5MB/s throughput for writing hidden data.**

1. Introduction

With increasingly abusive and restrictive governments and law enforcement agencies, it has become imperative to provide human rights activists, whistle-blowers, investigative journalists, and regular users and companies, techniques to ensure the privacy and confidentiality of their data. For instance, the Cloud Act [1] allows both US and non-US government agencies to compel telecommunications companies to disclose the content stored on their servers and data centers both in the US and overseas [2], [3]. Other examples include coerced inspections of mobile devices at checkpoints and border crossings in Burma [4] and even a videographer hiding a micro-SD card with evidence of human rights violations, in a wound when leaving Syria [5].

Encrypting data at rest is ineffective against adversaries that can coerce users to reveal the encryption keys. Plausibly-deniable (PD) systems thwart such coercion by providing mechanisms for users to hide sensitive information and later deny its existence, even when an adversary has access to the storage medium and encryption keys. Disk encryption tools such as TrueCrypt [6], Rubberhose [7], or Shufflecake [8] provide storage deniability for a single-snapshot adversary, that can access the user device only once. However, a multi-snapshot adversary can access the device multiple times. Example multi-snapshot opportunities include repeated government requests to cloud providers [1], [2], [3], crossing a country's border more than once, or an oppressive government colluding with a hotel maid.

More recent PD systems also protect against multi-snapshot adversaries [4], [9], [10], [11]. However, PD systems require specialized software and artifacts that render them vulnerable to adversaries questioning the use of such artifacts. Examples include (1) special software e.g., the Shufflecake tool [8] or a modified file translation layer (FTL) in PEARL [12], (2) system-specific metadata e.g, special indexing structures stored on disk to track hidden information, e.g., in HIVE [9], DataLair [10], PD-DM [11], or (3) non-standard on-disk data layouts, e.g., in DEFY [4] or PD-DM [11]. The presence of such specialized artifacts ultimately reveals to the adversary that the user is using a plausible deniable storage system to hide data.

Instead, an invisible PD solution would allow the user to deny the use of special software to write data. Thus, an invisible PD solution needs to ensure that any disk changes that result from changes to the hidden data between adversary snapshots, can be plausibly explained using changes to public data that are *compatible with regular use of an off-the-shelf software.* Further, the adversary should be able to successfully read all and only the public data stored on the device, using only such software.

In this paper we introduce and define the requirements for invisibility for plausibly-deniable storage. We introduce INVISILINE, a new *invisible* plausibly-deniable system effective against multi-snapshot adversaries, that leverages dm-crypt [13], the Linux distribution full disk encryption solution. During *sessions* taking place between adversary snapshots, the user-installed INVISILINE creates logical public and hidden dm-crypt devices on the physical disks, and uses them to write public and hidden data. INVISILINE opportunistically relies on the user's genuine writes and updates to public data during the session, as plausibly-deniable cover to write and update hidden data. At the end of each session, the user uninstalls INVISILINE and installs the vanilla dm-crypt.

INVISILINE achieves invisibility and plausible deniability by using a data layout and encoding that is compatible with dm-crypt. More specifically, it stores hidden data in the *initialization vectors* (IVs) used by dm-crypt block ciphers to encrypt public data. Since the adversary can take storage snapshots only at the end of a session, the hidden data-storing IV associated with a sector where public data has been written during the session can be updated any number of times during the session.

To an adversary studying the disk across snapshots, IV changes can plausibly be explained as changes in the randomness used to encrypt public data. Further, on coercion, the user can claim that only the vanilla dm-crypt was used to store the public data.

INVISILINE avoids the pitfalls of imitating applications, data formats or user behaviors that were shown to leave telltale traces for a monitoring adversary to discover hidden behaviors [14]. Instead, INVISILINE uses an existing storage format (dm-crypt) with the existing logic to manage the volume, and uses actual behavior of the user (e.g., browsing) and the OS (e.g., updated logfiles) to provide the cover needed to create the space for hidden data.

INVISILINE invisibly maps the logical hidden space to the physical sector space. It persists this address translation map (ATM) on disk along with the hidden data, and efficiently reads and stores it into memory for easy access. We further introduce INVISingle, an INVISILINE optimization for single-snapshot adversaries, that eliminates the need for an ATM table, increases the size of data that can be hidden on the disk, and improves the performance of accessing it.

We analyze INVISILINE and show that it satisfies the plausible invisibility requirements we introduced. Further, we implemented and tested INVISILINE on the Linux 5.19 kernel. On a 1TB disk, INVISILINE can invisibly store and map more than 19GB of hidden data, for which it requires 26s to reconstruct the ATM table. On the same disk, INVISingle can store 25GB of hidden data. Both INVISILINE and INVISingle have minimal impact on reading public data. INVISILINE achieves a throughput of 2.93 - 4.46 MB/s for hiding data at sequential and random addresses, and 3.73 - 6.55 MB/s for reading hidden data. The INVISingle optimization leads to a more than 50% throughput improvement.

2. Background

This section provides background on disk encryption software, in particular the dm-crypt and dm-integrity device-mapper targets in the Linux kernel, and the cryptsetup utility.

Disk Encryption. Modern operating systems have built-in software for performing disk encryption, e.g., dm-crypt [13] for Linux, BitLocker [15] for Windows, FileVault [16] for MacOS. Full disk encryption software encrypts everything on the disk except the boot loader. When the disk is mounted, once the user enters a password, the software decrypts the disk contents and presents it to the OS.

Disk encryption software can be classified based on where the encryption takes place. In file system-based encryption, it takes place inside the file system or as a layer stacked beneath the existing file system. For instance, eCryptfs [17] and EncFS [18] encrypt the data when writing to the disk, and decrypt it when data is read from the disk, before passing it to the upper layer. Block device encryption systems, e.g., dm-crypt [13], TrueCrypt [6] and Veracrypt [19], operate at the block device layer, and encrypt everything written to the block device. One advantage of this approach is that an adversary with access to an offline disk encrypted with a block device encryption system cannot interpret the file system and its contents. In the following we provide more details on the operation of dm-crypt [13].

dm-crypt. dm-crypt [13] is a transparent block encryption technique implemented as a device-mapper target in the Linux kernel. The device-mapper is a framework in the Linux kernel for mapping physical block devices onto virtual block devices. Device mappers pass data from a virtual (logical) block device to another block device; the data can be modified while in transition. In the dm-crypt target case, the modification consists of encrypting and decrypting the data. dm-crypt recommends that before using the disk, one should perform a secure erase by overwriting the disk with pseudorandom data. [20].

dm-integrity. dm-integrity is a device-mapper target that performs transparent read-write integrity checking of the underlying block device data. dm-integrity emulates an additional data integrity field for each data sector. The dm-integrity target can either be used in standalone mode, where it computes and verifies the integrity data internally or can be used along with dm-crypt, where the dm-crypt target supplies the integrity data along with the actual data. In both cases, any inconsistencies to the on-disk integrity data are identified, and an error is reported instead of returning incorrect data.

cryptsetup. *cryptsetup* [21] is a utility to create and manage dm-crypt based device-mapper mappings. It supports the Linux Unified Key Setup (LUKS) [22] extension to store all the setup information and to manage keys. The master key used to protect an encrypted block device can either be memorized as a passphrase or stored in a key file. *cryptsetup* supports different encryption ciphers and hashes, and relies on kernel cryptographic backend features. To use cryptsetup, the device needs to be setup with a LUKS header and the master key needs to be encrypted. The *cipher* used with cryptsetup specifies the desired block cipher and IV mode.

Disk Encryption Modes. A disk is normally divided into *sectors*, e.g., 512 or 4096 byte long, which are independently encrypted. To ensure confidentiality, disk encryption software needs to ensure that all encrypted sectors are indistinguishable, even if repetitions occur in the plaintext. This eliminates the ECB mode. Since the sector size is larger than the block size of standard encryption algorithms (e.g., AES), chaining modes need to be used. However, the CBC mode is vulnerable to malleability attacks [23], while the CTR mode is vulnerable when the keys and counters repeat. Tweakable ciphers like AES-XTS prevent such attacks by including a *tweak* argument that leverages sector numbers and provides explicit variability for every invocation of the block transform.

The default cipher to setup a dm-crypt device using LUKS is *aes-xts-plain64*, that uses the sector number for the initialization vector (IV) during encryption. This work instead relies on the *aes-xts-random* mode where the IV is pseudorandomly generated. This mode needs an additional 16 byte space per sector for storing the generated IV.

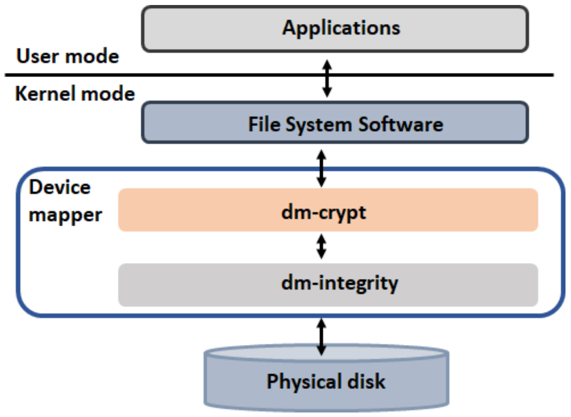


Figure 1. dm-crypt and dm-integrity software stack.

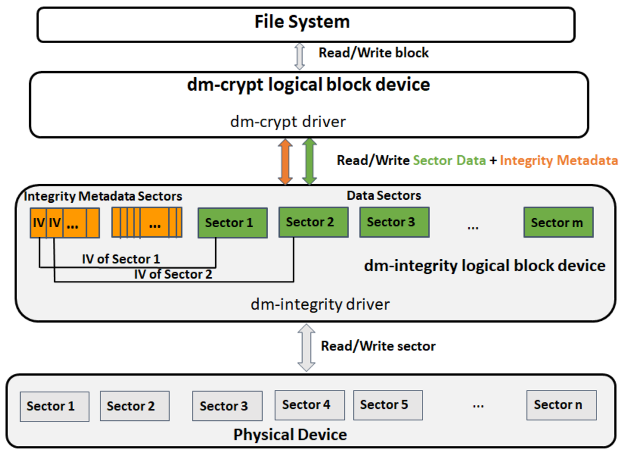


Figure 2. Details of dm-crypt and dm-integrity stack: the block devices involved. dm-integrity expects encrypted data and metadata from dm-crypt. dm-integrity writes the metadata to separate dedicated sectors, associated to the sector where it writes the data.

3. Model

3.1. System Model

We assume that the user owns a storage device $\mathcal{D}$ with n sectors. The *state* of the device is denoted by the data stored on it, $S(\mathcal{D}) = \cup_{i=1}^{n}(i, data_i)$, where $data_i$ is the data stored at sector i. The storage device is used to store and access both *public* and *hidden* data. The user is comfortable sharing the public data with the adversary upon coercion, but strongly prefers to not reveal even the existence of hidden data on the device.

Further, we assume that users use their devices during *sessions*. A session is defined by an explicit start time, when the user turns on the device and logs in, and an end time, when the user logs out or turns off the system.

Figure 1 shows the software stack and Figure 2 shows the block devices considered in this work. The lowest layer is the actual physical device consisting of n sectors. The dm-integrity target is a logical device mapper target built on top of the physical device. dm-integrity separates the sectors that hold the actual data from the sectors that hold integrity metadata (e.g., IVs, authentication tags) for the data sectors. Each data sector has its corresponding integrity metadata stored at an offset in a metadata sector. dm-integrity has a deterministic logic to get the integrity metadata, given a data sector number.

The dm-crypt target runs on top of the dm-integrity layer, and exposes another logical device on top of the dm-integrity logical device, to store encrypted data. When dm-crypt queries a data sector from dm-integrity, it receives both the data and the integrity metadata. The metadata is then used to decrypt and validate the data. Conversely, when dm-crypt writes data to a sector, it needs to supply to dm-integrity both the data and integrity metadata. The file system (e.g., EXT4) is mounted on top of the dm-crypt logical block device, to store files and directories, see Figure 2.

We assume that the *aes-xts-random* mode has been adopted by a reasonably large subset of dm-crypt users, e.g., because it provides better security at the expense of a small overhead. Even if only 1% of dm-crypt users have adopted the random mode, INVISILINE can achieve invisibility. The fact that random modes have been implemented at all suggests this to be indeed the case.

Crypto Tools. We consider a symmetric key cryptosystem with semantic security / IND-CPA *indistinguishability*: no polynomial time algorithm can distinguish (with non-negligible advantage) the ciphertexts of any two (different) same-length plaintexts [24]. In the following the notation $E_K(IV, M)$ denotes the symmetric key encryption of message M with key K and initialization vector IV. Further, we assume the use of a pseudorandom generator, i.e., whose output is indistinguishable in polynomial time from a uniform ensemble [25].

3.2. Plausible Invisibility Requirements

Informally, a system writing public and hidden data provides *plausible invisibility* if it satisfies the following:

- **Plausible Deniability**. Any changes to the disk, including changes to hidden data, can be plausibly explained using only changes to public data.
- **Readability with Off-the-Shelf Software.** The disk state should be sufficient to successfully read using only off-the-shelf software.
- **Public Data Non-interference**. Storing and modifying hidden data should not result in changes to public data.

Intuitively, plausible invisibility means that data stored on the device cannot be used to determine if the user has written it using anything besides off-the-shelf software.

However, anybody using off-the-shelf software to write to the disk inadvertently (unaware of the existence of hidden data) can and most likely will overwrite some of the stored hidden data.

3.3. Adversary Model

The **adversary** can capture device snapshots. Further:

- **Single vs. Multi-Snapshot**. Both single-snapshot and multi-snapshot adversaries are of concern. Single snapshot adversaries can inspect the device at most once, whereas multi-snapshot adversaries can inspect and record the device state at multiple points in time.
- **In-Session Snapshots**. At runtime, during a session, the adversary cannot take snapshots or monitor the CPU, RAM, I/O etc.
- **Software Inspection**. The adversary can inspect the data on the device $\mathcal{D}$, but not the software used to store the data or the system logs. § 5.3 discusses possible implementations of this assumption.
- **Access to Raw State**. While the adversary can use the standard device I/O interface to read data, it does not have access to any substrates and/or device-specific underlying state such as raw flash memory chip address spaces on SSDs etc. Justification and details included in § 6.

Plausible Invisibility (P-INV) Game. We extend the game from [11] to introduce a *plausible invisibility* (P-INV) game between an adversary $\mathcal{A}$ and challenger $\mathcal{C}$ on a storage device $\mathcal{D}$. The game takes place over r rounds, and uses the notation $S(\mathcal{D})[k]$ to refer to the storage state of device $\mathcal{D}$, i.e., the data stored on it, at the end of round $k = 1..r$.

Consistent with seeking to be indistinguishable from popular off-the-shelf software, the game is defined for two systems: HideSys – a software claimed to provide invisibility for plausibly-deniable storage, and OTSS – off-the-shelf software such as the popular dm-crypt and dm-integrity systems (§ 3.1).

An *access pattern* $\mathcal{O}$ consists of a sequence of disk write operations, where each operation writes either public or hidden data. The P-INV game relies on an $IsExecutable(\mathcal{O})$ predicate that returns true if and only if the input access pattern $\mathcal{O}$ can be fully executed. Intuitively, the predicate is used to model whether there are sufficient public writes in $\mathcal{O}$ to provide cover for the hidden data writes. In practice if there are not sufficient public writes a standard I/O error (e.g., BUSY or RETRY) will be returned by the hidden layer block device. The analysis section (§ 6) instantiates $isExecutable$ for INVISILINE.

The game proceeds through the following steps:

- The challenger $\mathcal{C}$ chooses public and hidden encryption keys K_p and K_h using a security parameter s. $\mathcal{C}$ sends K_p to the adversary $\mathcal{A}$.
- $\mathcal{C}$ selects a random bit b. If $b = 0$, $\mathcal{C}$ installs OTSS on device $\mathcal{D}$. If $b = 1$, $\mathcal{C}$ sets up HideSys on $\mathcal{D}$.
- $\mathcal{A}$ and $\mathcal{C}$ engage in r rounds as follows (where $k = 1..r$ is the current round):
  - $\mathcal{A}$ generates a set of public data writes $PW_k = \{(s_i, pd_i)\}$, writing public data pd_i to sector s_i. $\mathcal{A}$ also generates a set of hidden data writes $HW_k = \{(hs_j, hd_j)\}$, writing hidden data hd_j at hidden sector number hs_j. Let $\mathcal{O}_k^0 = PW_k$ and $\mathcal{O}_k^1 = PW_k \cup HW_k$. PW_k and HW_k are chosen such that $IsExecutable(\cup_{l=1}^{k} \mathcal{O}_l^1) = True$.
  - $\mathcal{A}$ sends $\mathcal{O}_k^0$ and $\mathcal{O}_k^1$ to $\mathcal{C}$.
  - $\mathcal{C}$ executes $\mathcal{O}_k^b$ on $\mathcal{D}$
  - $\mathcal{A}$ uses the standard device I/O interface to take snapshots, possibly through some existing software on top, including OTSS or HideSys.
- $\mathcal{A}$ outputs $b^{'}$, $\mathcal{A}$'s guess for b.

The advantage of adversary $\mathcal{A}$ in the P-INV game is $Adv(\mathcal{A}) = |P(b' = b) - P(b' \neq b)|$. $HideSys$ is said to provide *plausible invisibility* if any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ has only negligible advantage in the P-INV game.

The number of rounds r is 1 for a single-snapshot adversary, and polynomial in the case of a multi-snapshot adversary. The access patterns $\mathcal{O}_k^0$ and $\mathcal{O}_k^1$ are inherited from the PD games [11], [10]: the former contains only writes to public data, and the latter contains a mix of writes to public and hidden data.

P-INV vs. IND-PD Game [11]. The challenger $\mathcal{C}$ in the IND-PD game of [11] (Appendix A) always uses HideSys, and the adversary $\mathcal{A}$ needs to determine if $\mathcal{C}$ used HideSys to write only the public data or the public and the hidden data. In contrast, the P-INV game requires that an adversary cannot use snapshots of the device storage to determine the software used by the challenger, i.e., cannot distinguish between a challenger using HideSys to write both hidden and public data (provided by the adversary) and the challenger using OTSS to write only the public data. Intuitively, the P-INV game models the stronger guarantee that the adversary is not able to determine the software (HideSys or OTSS) used to write the data.

Further, previous IND-PD games [11], [10] require the challenger to send storage snapshots, implicitly captured using HideSys, to the adversary. In contrast, the P-INV game gives more control to the adversary, by allowing it to capture device storage snapshots using OTSS and/or HideSys. The P-INV game still enforces the in-snapshot adversary restriction, by requiring the challenger to signal to the adversary when it can take snapshots.

Thus, in the P-INV game, the failure to read a complete device snapshot, including all the public writes from $\mathcal{O}_l^0$, $l = 1..k$, using OTSS, or failure to read the same public data using OTSS and HideSys, can be used by the adversary to correctly guess that the challenger bit b is 1. If HideSys provides *plausible invisibility*, the adversary would be able to read public data using OTSS even if the challenger used HideSys for writing.

In addition, the P-INV game extends the IND-PD game to allow the adversary to specify the sector numbers s_i where public writes are to take place; also logical hidden sector numbers hs_j for hidden writes. This models cases where the user overwrites previously written public and hidden data. In practice however, adversaries do not really know the sector numbers used by the user to store hidden data, or indeed whether the user has written hidden data.

P-INV Implies IND-PD [11]. P-INV (§ 3.3) implies plausible deniability as defined in the IND-PD game of [11] (Appendix A). This follows straightforwardly: the challenger uses HideSys to write both public and hidden data when bit b is 1. If HideSys satisfies P-INV, it has to emulate OTSS for writing public data. Otherwise, the snapshot would be

System	Layer	Multi snapshot	Plausible Inv. PD+OTSS Rd.	Media Requirements
DenFS [26]	FS	○	○	○
DEFY [4]	FS	●	○	○
TrueCrypt [6]	BD	○	○	○
VeraCrypt [19]	BD	○	○	○
Rubberhose [7]	BD	○	○	○
Shufflecak [8]	BD	○	○	○
Mobiflage [27]	BD	○	○	○
HIVE [9]	BD	●	○	○
MobiCeal [28]	BD	●	○	○
DataLair [10]	BD	●	○	○
PD-DM [11]	BD	●	○	○
ECD [29]	Firmware	●	○	access to SSD firmware
StegFS [30]	FS	○	●	○
INFUSE [31]	Firmware	●	●	FTL access, voltage manipulation
PEARL [12]	Firmware	●	○	FTL access
INVISILINE	Block Device	●	●	○
INVISingle	Block Device	○	●	○

TABLE 1. COMPARISON OF EXISTING WORK'S PROPERTIES. FULL AND EMPTY CIRCLES DENOTE THAT THE SOLUTION SATISFIES OR DOES NOT SATISFY THE PROPERTY, RESPECTIVELY. INVISILINE IS THE ONLY MULTI-SNAPSHOT, PLAUSIBLY-INVISIBLE STORAGE SOLUTION THAT HAS NO MEDIA REQUIREMENTS.

different from the expected output of OTSS on the public data, and the adversary would guess that the challenger's bit b is 1. Then, an adversary that has a non-negligible advantage in distinguishing between HideSys storing hidden and public data vs. HideSys storing only the public data (IND-PD game) also has a non-negligible advantage in distinguishing HideSys storing hidden and public data vs. OTSS storing only the public data (P-INV game).

P-INV Captures OTSS Readability. A HideSys that does not provide OTSS readability does also not provide plausible invisibility. This is because an adversary can leverage the discovery that HideSys does not provide OTSS readability to gain an advantage in the P-INV game.

OTSS readability indicates that the adversary is able to use an OTSS to read all and only the data that was written by it. If HideSys does not satisfy the public data non-interference property, i.e., hidden data writes result in changes to public data, then OTSS readability is also lost: the adversary reads different data than what is expected. Therefore, *OTSS readability implies public data non-interference (§ 3.2).*

4. Related work

Chen et al. [32] provide an in-depth systematization of plausible deniability knowledge. This section focuses on prior work most closely related to this paper.

Plausibly-Deniable File Systems. Anderson et al. [33] introduced *steganographic filesystems*. Later, McDonald and Kahn [30] developed StegFS, that implemented the approach proposed in [33], for Linux. Pang et al. [34] improved on previous constructions by providing more efficient storage and avoiding hash collisions. These early solutions do not protect against multi-snapshot adversaries.

Han et al. [35] designed DRSteg, a Dummy-Relocatable Steganographic filesystem where multiple users can share the same hidden data, and relocate it to ensure deniability against a multi-snapshot adversary. However, since DRSteg attributes deniability to joint ownership of sensitive data, it is not the ideal solution for single-user devices.

Gasti et al. [26] introduced DenFS, a deniable shared file system designed for cloud storage, and whose security depends on processing data temporarily on a client machine. DenFS is built on a public key encryption scheme using RSA-OAEP and the Damgård-Jurik generalization of Paillier's encryption scheme. Thus, DenFS is not invisible. Peters et al. [4] developed DEFY, a file system for flash devices that leverages secure deletion of data to provide plausible deniability (but not invisibility) against multi-snapshot adversaries.

Plausibly-Deniable Block Devices. Disk encryption tools were further leveraged to support plausible deniability at block device level. Tools like Truecrypt [6], VeraCrypt [19], Rubberhose [7], Shufflecake [8], or Mobiflage [27] provide plausible-deniability only for single-snapshot adversaries. For instance, a multi-snapshot adversary can infer the presence of hidden data in TrueCrypt [6] through unexplainable changes to the random uninitialized data.

Several solutions provide plausible deniability against multi-snapshot adversaries by using an oblivious RAM (ORAM) [9], [10], [28]. For instance, Blass et al. [9] proposed HIVE, the first PD solution against device-level multi-snapshot adversaries, using a write-only ORAM (wORAM). MobiCeal [28] improved performance by replacing wORAMs with dummy write operations coupled to public writes. MobiCeal uses two types of volumes (dummy and hidden) and generates dummy writes to store hidden data. MobiCeal does not provide invisibility, since the presence of dummy volumes and writes signals attempts to hide data. Chakraborti et al. [10] further built DataLair that improved on HIVE through the observation that operations on public data do not need to be hidden but can in fact be used to reinforce deniability.

Other solutions [29], [11] use canonical forms used for instance in log-structured file systems [36] to decouple the user's logical from physical access patterns. In particular, Zuck et al. [29] developed ECD that partitions the device into a public and a hidden volume. Chen et al. [11] introduced PD-DM, a locality-preserving PD solution that divides the disk into a data segment and a mapping segment that stores multiple data structures for logical-to-physical mapping. The processing done by ECD and PD-DM render them observable.

A key difference between existing multi-snapshot PD storage solutions and INVISILINE is *invisibility*: Most PD solutions, and block device-level solutions in particular, modify the disk layout and encoding schemes, making it necessary for special software to be used to read the public data. The presence of such software and layout implies that an adversary in the P-INV game of § 3.3 would fail to read snapshots using OTSS, thus would acquire a non-negligible advantage in determining when the challenger's bit b is 1.

Efforts Towards Invisible PD. A few PD systems attempted invisibility. StegFS [30] aimed to look like EXT2 and INFUSE [31] aimed to look like YAFFS [37], a popular file system for flash devices. PEARL [12] surreptitiously hides data in the WOM codes of public data.

Unfortunately, StegFS is not really designed for invis-

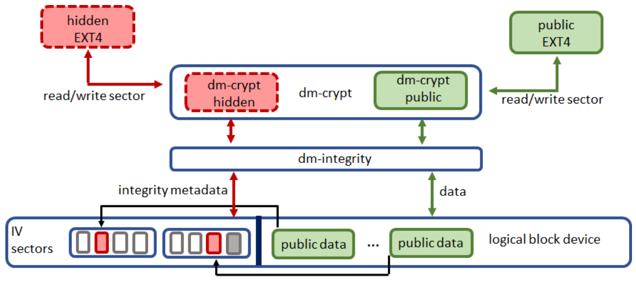


Figure 3. System architecture. INVISILINE has two instances (hidden and public) of dm-crypt target instantiated for a physical device. The instances share data structures and keys, e.g., the public key K_p and the hidden key K_h. Both dm-crypt instances operate on top of the dm-integrity logical device mapped to the underlying physical device. Hidden data is embedded into IVs associated with physical sectors where public data was written.

ibility and performs several changes to the disk that an adversary can easily catch: deleted public files are replaced with random data; new block allocation is changed for new files to plausibly justify hidden blocks; a block table is needed and stored in a non-hidden file that contains inode numbers for inode blocks, etc. Any of these (and more) immediately break plausible invisibility (§ 3.3).

Further, INFUSE only works for flash and to hide data, requires hardware support, and firmware that enables precise manipulation of flash cell voltages, currently unavailable in most NAND flash chips.

PEARL is based on customized, adversary-observable WOM codes, and requires a modified file translation layer (FTL), thus also breaking OTSS readability.

The INVISILINE design aligns with some of the intuitions of Houmansadr et al. [14] to hide data into the features of a cover medium provided by an existing protocol. INVISILINE hides data into the IVs used to encrypt public data stored by the popular dm-crypt disk encryption software available in the Linux distribution. Public data encrypted with INVISILINE can be accessed and decrypted with the standard dm-crypt, thereby enabling coerced users to deny the presence of data hidden with any other software.

5. INVISILINE

5.1. Overview

INVISILINE extends the device-mapper dm-crypt target (§ 2). The resulting dm-crypt-hidden target, see Figure 3, achieves invisibility while storing data with plausible deniability. INVISILINE exposes two logical block devices to the upper layers for each physical device, one for storing hidden data and the other for public data. The logical *hidden* block device enables storing and accessing hidden data, while the logical *public* block device stores public data. Both devices store data in encrypted format. All operations on the hidden and public devices in a session are mapped to the same dm-integrity logical device, thus to the same physical device.

INVISILINE stores encrypted hidden data in the integrity metadata space provided by the underlying physical layer.

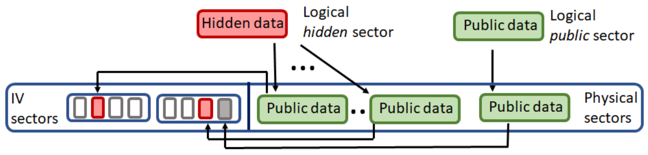


Figure 4. Mapping of logical public and hidden sectors to physical sectors. Logical hidden sectors (large red rectangle) are mapped to the IVs (small red rectangles) of multiple physical sectors storing public data (large green rectangles). IVs are stored in dedicated integrity metadata sectors.

Specifically, INVISILINE stores hidden data in the IVs used to encrypt public data sectors. INVISILINE opportunistically uses the user's actual writes to public data within a session as plausibly-deniable excuses to recalculate the IVs of public data blocks in which hidden data is to be written. Since IVs are pseudorandom in INVISILINE, an adversary cannot distinguish IVs from encrypted hidden data.

User Sessions. INVISILINE users use their devices during explicit sessions, see (§ 3.1). At the start of each session, the user installs INVISILINE and mounts both the *public* and *hidden* logical devices. At the end of each session, in order to maintain invisibility, the user unmounts the devices, uninstalls INVISILINE and erases the volatile memory and caches and uninstalls all modules mounted at the start of the session. The user then mounts the vanilla dm-crypt module.

Logical and Physical Sectors. In the following, *physical* sectors denote sectors addressed by the dm-integrity logical device, and *logical* sectors denote sectors used by the upper file system layer when communicating with dm-crypt, see Figure 4. For the public device, logical sector numbers are the same as physical sector numbers. However, since the hidden data is stored in IVs of physical sectors, each physical sector can contribute to the storage of up to 16 bytes of hidden data.

INVISILINE maintains in memory the ATM table that maps logical hidden sectors to physical sectors. To reconstruct this table after sudden failures, INVISILINE also implicitly stores this table along with the hidden data.

Storage Factor. INVISILINE uses a sector size of 512 bytes for both logical and physical devices. Then, a logical hidden sector of 512 bytes needs a *storage factor* f of at least 32=(512/16) physical sectors. That is, the data in a logical hidden sector is stored in the IVs corresponding to f physical sectors, see Figure 4. Thus, to read and write one hidden sector, INVISILINE reads and writes f physical sectors. We show later that the storage factor increases due to requirements to populate IVs with hidden metadata alongside hidden data. However, the storage factor is an important design factor, since it determines how much hidden data can be stored on a disk (refer section 8 for exact values).

Invisible Plausible-Deniable Storage. The data layout and encoding on the disk used by INVISILINE is indistinguishable from the ones used by off-the-shelf software (OTSS), e.g., vanilla dm-crypt. Further, the public data stored by INVISILINE is accessible with OTSS. More specifically, if INVISILINE is used to store both hidden and public data on a disk, when the disk is unmounted and INVISILINE is uninstalled, the user can still read the public data using vanilla dm-crypt.

5.2. INVISILINE Requirements and Implications

In addition to the requirements of § 3.2, INVISILINE needs to also satisfy the following:

- **Cover public data writes**. The system relies on the user's public data write operations to provide cover for writing hidden data.
- **Explainable re-encryption**. Public sectors which are not modified during a session should not be re-encrypted as a result of changes to hidden data.
- **In-session public data writes**. For each session, public data changes need to be remembered to be used later as cover for hidden writes: future changes to hidden data can be mapped to public sectors that were modified within the session.
- **Hidden data non-interference**. Storing and modifying public data should not result in changes to hidden data. Note that this is different from **public** *data non-interference* discussed in § 3.2. The former is captured in the game and the latter we prove in the Analysis section § 6.
- **Security non-interference**. Embedding hidden data in IVs used to encrypt public data should not break encryption security.
- **Logical-to-physical sector map**. Data from a logical hidden sector can be stored on the IVs of any public sector. Thus, logical-to-physical sector maps need to be persisted between sessions.

For the first requirement, we note that an honest disk contains many genuine public data writes from constantly changing files, e.g., the web browser cache and logfiles. Further, the public data writes must take place during the same user session, before the hidden data write. To see why this is the case, consider a scenario where there are only hidden data changes between two snapshots. If the IVs alone are modified, even if the corresponding public data sectors are re-encrypted with the modified IVs, the adversary would observe that the public data has not changed. Thus, any write to the hidden data stored in an IV must be accompanied by a change to the public data written in the corresponding public sector. The following details how INVISILINE addresses these requirements and challenges.

5.3. Solution Details

Setup. Similar to regular disk encryption using dm-crypt, INVISILINE uses cryptsetup (§ 2) to setup the device to store both hidden and public data. However, INVISILINE modifies cryptsetup in two ways. First, it implements a new option for the *luksFormat* action, to set up passphrases for accessing both public and hidden devices. Second, it modifies cryptsetup's *open* action to prompt the user to enter unlocking passphrases for both the devices. cryptsetup uses the public passphrase to generate a master public key K_p, to encrypt the public data. Further, it uses the hidden passphrase to derive the master hidden key K_h, to encrypt the hidden data.

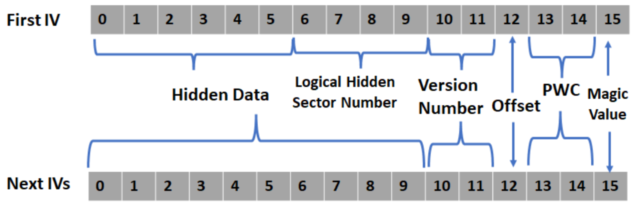


Figure 5. IV format for (top) first IV in a sequence of f hidden data-storing IVs, and (bottom) the other IVs in the sequence.

After checking the passphrases, cryptsetup uses the *open* action to create a dm-integrity mapped device and two dm-crypt mapped device mapper devices: the hidden and the public device (see § 7 for implementation details).

If built into the standard Linux distro, INVISILINE software would immediately provide plausibility for its presence. Before that happens however, the INVISILINE module can be downloaded (online or from external media such as USB sticks) and inserted on demand at runtime. If system logs are set to record module inserts and are stored on persistent media, we may want to ensure that the module is named with the same name as one of the (hundreds) existing standard Linux kernel modules. The system can be set up with ephemeral logs that do not persist after reboot, and in any case, the adversarial model § 3.3, does not allow access to system logs. Finally, another alternative is booting from a USB stick with an OS pre-loaded with INVISILINE.

Freelist: Tracking Changes to Public Data. INVISILINE addresses the requirement to track public data writes (see § 5.2) by maintaining in memory, a *freelist* that stores information about all the public writes that take place within the session. The *freelist* stores a list of all the public sectors that have been written, whose IVs do not contain hidden data or contain stale versions of the hidden data (see version number paragraph below). Some of these public sectors may be encrypted at least twice (see § 9): once with a pseudorandom IV during the public write, then again with hidden data as IV during a hidden write. These indicate sectors where hidden data can be stored without raising suspicion at the next snapshot. INVISILINE maintains the *freelist* in dm-crypt for every physical disk that it manages. The list is erased at the end of each session. This avoids using public sectors that may have been recorded by the adversary in snapshots taken between sessions. A key point to consider is that INVISILINE utilizes the public writes generated through regular user activity, and does not mandate the creation of explicit public writes by the user for storing hidden data. However, if there are insufficient public writes available, an IO error will occur.

Hidden ATM Table. To address the logical-to-physical sector mapping requirement of § 5.2, INVISILINE maintains an ATM between logical and physical sectors. Each entry in the ATM is a tuple $(lhs, phys\_sector, VerNo, d, m)$, where lhs is a logical hidden sector number, $phys\_sector$ is the starting sector number of f physical sectors whose IVs store the data of lhs, $VerNo$ is a versioning number, d is a boolean that indicates if the sector has been deleted and m is a boolean that indicates if the sector has been modified in the current session.

While the translation table is stored in memory, it needs to be persistent across sessions. This can be achieved by storing the table explicitly on disk. The ATM table then needs to be updated on every hidden write operation. However, since the table maintains metadata for hidden data, any changes to the table need to also be plausibly deniable to the adversary. This implies that the table cannot be stored at fixed locations, since frequent changes to those locations would be impossible to explain.

Instead, INVISILINE uses an implicit approach to persist the address translation information, i.e., store it alongside the hidden data. More specifically, each IV that stores hidden data also stores the logical hidden sector number (lhs) to which it belongs. The hidden data and this metadata are encrypted together using the master hidden key K_h.

Figure 5 details the format of data-storing IVs. In the following we detail each field.

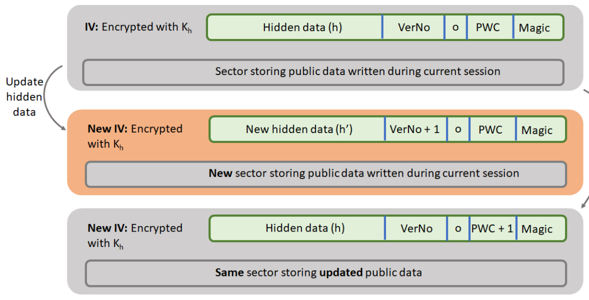
IV Offset. Since hidden data can be stored in the IVs of any physical sector, the *offset* of each hidden data-storing IV (see Figure 5) records the IV's order in the sequence. Valid values are between 0 and $f-1$.

Magic Value. To determine if an IV stores hidden data or not, INVISILINE stores a one byte magic value inside each IV that stores hidden data, see Figure 5. Upon decryption of the IV of a sector storing public data, INVISILINE compares the last byte in the result against the magic value. Section 6 discusses how to reduce false positives.

Hidden Data Version Numbers. When the user updates hidden data stored in f IVs, INVISILINE cannot force the user to also update the public data stored on the sectors corresponding to the IVs. This is because the adversary would observe during the next snapshot if an unnecessary public data re-encryption has taken place: the IVs have been updated, the corresponding public data has been re-encrypted with the new IVs, but the public data did not change. Instead, when the user updates hidden data, INVISILINE writes the new hidden data to another set of f IVs, found in the freelist. Figure 6 (top and middle) illustrates this process for one public data-storing sector and corresponding hidden data-storing IV.

This results in INVISILINE storing both old and new versions of the hidden data. To identify the latest version of a logical hidden data sector, INVISILINE maintains a 2 byte $VerNo$, a *version number* for each logical hidden sector, and includes it in each IV that stores hidden data, see Figure 5. The adversary is unable to find version numbers in IVs, since all hidden data and metadata in an IV is encrypted with the master key K_h before being stored.

INVISILINE increments $VerNo$ on every update to the logical hidden sector. Since $VerNo$ is part of every IV, this ensures that the IVs containing hidden data are not the same even if the hidden data remains the same. $VerNo$ ensures that IVs used for encrypting public sectors are not reused, see "Security non-interference" requirement (§ 5.2). We later detail the INVISILINE process to ensure that the in-memory ATM table stores the latest $VerNo$ for each logical hidden sector, even in the presence of sudden device failures.



IV: Encrypted with K_h
Hidden data (h)
VerNo
o
PWC
Magic
Sector storing public data written during current session
Update hidden data
New IV: Encrypted with K_h
New hidden data (h')
VerNo + 1
o
PWC
Magic
New sector storing public data written during current session
Update public data
New IV: Encrypted with K_h
Hidden data (h)
VerNo
o
PWC + 1
Magic
Same sector storing updated public data

Figure 6. Disk changes following data updates. (Top) Physical sector storing public data and corresponding IV storing hidden data. (Middle) Updated hidden data from (top) written to a new physical sector. (Bottom) Updated public data and IV stored in-place of (top) example.

Public Writer Counters. What happens when the user updates public data stored in a sector whose corresponding IV stores hidden data? The vanilla dm-crypt would generate a new pseudorandom IV and use it to re-encrypt the updated public data. This would erase the hidden data. To preserve the hidden data while being indistinguishable from vanilla dm-crypt, INVISILINE needs to also update the hidden data. However, the user is unlikely to sync updates to the public and the hidden data. Instead, INVISILINE stores a 2 byte *Public Write Counter* (PWC) in each hidden data-storing IV, see Figure 5. On each update to the public data, INVISILINE decrypts the IV. If it stores hidden data (see the magic value), it increments PWC before re-encrypting the IV and using the new IV to re-encrypt the updated public data. This process is illustrated in Figure 6 (top and bottom). This ensures that once the IV is re-encrypted, it will be considered to be random and indistinguishable from the previous IV value.

Dirty List. Frequently modified sectors, both hidden and public, may overflow the 2 byte version number $VerNo$ and Public Write Counter PWC. INVISILINE avoid this by incrementing these counters at most once per session. INVISILINE uses the in-memory ATM table to avoid overflowing the version number. More specifically, when a logical hidden sector is modified the first time during the session, its data is moved to a new set of physical sectors. Then, the hidden sector's entry in the ATM table is updated to point to the new physical sector, its $VerNo$ value is incremented, and its *modified* bit m is set to true. During subsequent updates of this logical hidden sector in the session, the hidden data updates are not moved to other physical sectors, and the $VerNo$ value is not incremented. This works because (1) the adversary cannot access the device during sessions, and (2) during the next snapshot the user can justify the hidden data update through the changes to the new physical sectors in whose IVs the data is hidden.

To avoid incrementing the PWC quickly and risking rapid overflow, INVISILINE maintains in memory, a *dirty list* of all the public sectors that were written in the current

Algorithm 1 $Write\_Hidden\_Data(s_n, s_d)$.

```
Require: $s_n$, $s_p$                    ▷ write hidden sector number $s_n$ with data $s_d$
 1: $f := 52$
 2: $MAGIC := 0xAA$
 3: $(ps\text{-}start_n, VerNo, m) := getfrom\_ATM(s_n)$ ▷ Get ATM entry
 4: if ($ps\text{-}start_n$ == -1 || $m$ == 0) then
 5:     $(ps\text{-}start_n, error) := getfrom\_freelist(f)$ ▷ Get free sector
 6:     if ($error$ == "insufficient public writes") then
 7:         set IO_Error
 8:         return
 9:     end if
10: end if
11: $(Enc_{data}, IV_{data}) := Phy\_read(ps\text{-}start_n, f)$ ▷ Read $f$ sectors
12: $PubData := decrypt(Enc_{data}, K_p, IV_{data})$ ▷ Decrypt pub data
13: for $i \in \{0, \ldots, f-1\}$ do
14:     if $i$ == 0 then
15:         $IV_{data}[i] := buildIV(s_d, s_n, VerNo+1, i, 0, MAGIC)$
16:     else
17:         $IV_{data}[i] := buildIV(s_d, VerNo+1, i, 0, MAGIC)$
18:     end if                                        ▷ Construct IV
19:     $IV_{data}[i] := encrypt(IV_{data}[i], K_h, ps\text{-}start_n + i)$ ▷ Encrypt
20: end for
21: $Enc_{data} := encrypt(PubData, K_p, IV_{data})$ ▷ Encrypt pub data
22: $Phy\_write(ps\text{-}start_n, f, Enc_{data}, IV_{data})$ ▷ Write back data, IV
23: $addto\_ATM(s_n, ps\text{-}start_n, VerNo+1, 1)$ ▷ Update map, $m = 1$
24: $removefrom\_freelist(ps\text{-}start_n, f)$ ▷ Update freelist
```

session, and whose IVs store the latest version (i.e., active version) of some hidden data. When a public write occurs on a physical sector whose IV stores active hidden data, and the physical sector is not in the dirty list, INVISILINE increments the IV's PWC, re-encrypts the IV, then uses the updated IV to re-encrypt the public data. However, if the physical sector is present in the dirty list, INVISILINE uses the same IV (without incrementing the PWC) to encrypt the new public data. This ensures that PWC is incremented on only once per session.

A physical sector is removed from the dirty list if it no longer holds active hidden data, i.e., the latest version of the logical hidden sector.

Storage Factor. The outlined IV format reduces the number of hidden bytes per IV to 6 bytes. Thus, the storage factor f, i.e., the number of public sectors required to store a single hidden sector, increases to $\lceil 512/6 \rceil = 86$. INVISILINE reduces this number by storing the logical sector number only in the first IV of the f physical sectors. Figure 5 shows the IV format separately for the first IV storing the data from a hidden sector, and the remaining $f-1$ IVs in the sequence. Thus, the storage factor f becomes 52. INVISILINE assumes that the f physical sectors used to store the data of a hidden sector are sequential.

Next we detail how INVISILINE operates on hidden and public data. We then explain its reconstruction of the ATM table and its support for sector deletion.

Write Hidden Data. The $Write\_Hidden\_Data$ function, whose pseudocode is shown in Algorithm 1 is used when *dm-crypt-hidden* receives requests to write hidden data at a logical hidden sector s_n. The function first searches the ATM table to find a matching entry for the logical hidden sector number (line 3). If an entry exists and its modified bit m is set, it uses the mapped physical sectors from the entry. If not, it scans the freelist looking for f sectors to store the input hidden data (line 5). If the freelist does not have f available sectors, the function returns an error (line 7,8) (see the *hidden data writes can fail* requirement of § 5.2). This signals to the user that more public writes are required for the hidden write to succeed.

Once f sectors are identified, the function proceeds to read public data along with IVs from those sectors (line 11). The public data is then decrypted using the public key K_p and the corresponding IVs (line 12). It then constructs the f IVs using the hidden data as per Figure 5 with an incremented version number and the public write counter (PWC) set to 0 (line 15 and 17). Each of the f IVs is then encrypted with the hidden key K_h and the physical sector number as IV (line 19).

The f public sectors are then re-encrypted using the newly generated IVs (line 21) and written back to the underlying layer (line 22). The (logical sector number, physical sector number, incremented version number, set m bit) tuple is then added to the ATM table (line 23) and the freelist is updated to remove the allocated sectors (line 24).

Algorithm 2 $Read\_Hidden\_Data(s_n)$

```
Require: $s_n$                         ▷ read hidden sector number $s_n$
 1: $f := 52$
 2: $(ps\text{-}start_n, VerNo, m) := getfrom\_ATM(s_n)$ ▷ Read ATM
 3: $(Enc_{data}, IV_{data}) := Phy\_read(ps\text{-}start_n, f)$ ▷ Read $f$ sectors
 4: for $i \in \{0, \ldots, f-1\}$ do
 5:     $IV_{data}[i] := decrypt(IV_{data}[i], K_h, ps\text{-}start_n + i)$ ▷ Decrypt
 6: end for
 7: return $IV_{data}$
```

Read Hidden Data. Algorithm 2 shows the pseudocode for reading a logical hidden sector. *dm-crypt-hidden* first queries the in-memory map to retrieve the starting physical sector number where the data is stored (line 2), then issues a read request to the dm-integrity layer for the data and IVs of f consecutive sectors from that location (line 3). It then decrypts each IV using the hidden key K_h and physical sector number (line 5) and returns the result (line 7).

Write Public Data. The *dm-crypt-public* instance of INVISILINE receives the requests for operations on public data. Algorithm 3 shows the pseudocode for public data writes. INVISILINE provides hidden data non-interference (§ 5.2) by ensuring that hidden data present in a physical sector's IV is not overwritten on public write. The function reads the sector's data and the corresponding IV (line 2), decrypts the IV (line 3), and checks if it holds hidden data (line 4).

If the IV contains hidden data, it extracts the IV offset field (line 5), with value between 0 and $f-1$. If the IV's offset is not zero, the function fetches the first IV (line 7). This is needed since only the first of the f IVs storing a logical hidden sector's data, stores the hidden logical sector number (see Figure 5). The function extracts the logical hidden sector number and queries the ATM table for the mapped physical sector number (line 8 and 10). It then checks to see if the public sector holds the latest version of the hidden data, by comparing the input public sector

Algorithm 3 $Write\_Public\_Data(s_n, s_d)$

```
Require: $s_n$, $s_p$    ▷ write public sector number $s_n$ with data $s_d$
1: $MAGIC := 0xAA$
2: $(Enc_{data}, IV_{data}) := Phy\_read(s_n, 1)$    ▷ Read data and IV
3: $IV_{data} := decrypt(IV_{data}, K_h, s_n)$    ▷ Decrypt hidden data with $K_h$ and $s_n$
4: if ( ( $IV_{data}[15]$ == $MAGIC$ ) then
5:     $offset := IV_{data}[12]$
6:     if ( $offset \neq 0$) ) then
7:         $IV_0 := get\_IV\_for\_sector(s_n - offset)$    ▷ Get first IV
8:         $(mapped\text{-}sector, VerNo, m) := getfrom\_ATM(IV_0)$
9:     else
10:        $(mapped\text{-}sector, VerNo, m) := getfrom\_ATM(IV_{data})$
11:    end if
12:    if ( $mapped\text{-}sector + offset \neq s_n$) ) then
13:        $IV_{data} := get\_random\_bytes(16)$    ▷ pseudorandom IV
14:        $addto\_freelist(s_n)$    ▷ Add $s_n$ to freelist
15:        $removefrom\_dirty\_list(s_n)$    ▷ Remove $s_n$ from dirty list
16:    else
17:        if $(in\_dirty\_list(s_n) == False)$ then
18:            $IV_{data}[13:14] := IV_{data}[13:14] + 1$    ▷ PWC ++
19:            $IV_{data} := encrypt(IV_{data}, K_h, s_n)$    ▷ Encrypt hidden
20:            $addto\_dirty\_list(s_n)$    ▷ Add $s_n$ to dirty list
21:        end if
22:    end if
23: else
24:    $IV_{data} := get\_random\_bytes(16)$    ▷ Generate pseudorandom IV
25:    $addto\_freelist(s_n)$    ▷ Add $s_n$ to freelist
26: end if
27: $Enc_{data} := encrypt(s_d, K_p, IV_{data})$    ▷ Encrypt public data
28: $Phy\_write(s_n, 1, Enc_{data}, IV_{data})$    ▷ Write public data, IV
```

number with the mapped physical sector number (line 12). If there is no match (i.e., the latest version of the hidden data is mapped elsewhere) the function erases the old version of the hidden data by generating and storing a pseudorandom IV (line 13) and using it to encrypt the public data (line 27). INVISILINE then adds the physical sector number to the *freelist* (line 14), to be used during a future hidden write. It then removes it from the dirty list, if present (line 15): the sector no longer contains the latest hidden data.

If the IV stores the latest version of the data for the logical hidden sector number, INVISILINE increments its PWC field (line 18), and adds the physical sector to the dirty list (line 20). It then re-encrypts the IV (line 19), and uses it for encrypting the public data (line 27). The encrypted public data along with the refreshed IV are stored through the dm-integrity layer (line 28). Note that in this case, the *freelist* is not updated. As discussed above, the dirty list allows INVISILINE to increment the PWC only once per session.

Read Public Data. INVISILINE uses the the vanilla dm-crypt code to read public data: it reads the physical data along with the IV, decrypts the data using K_p and returns the result.

Reconstructing the ATM Table. INVISILINE stores the explicit ATM table, that maps logical hidden sectors to physical sectors, in memory. INVISILINE also persists this information, by storing it along with the hidden data. Upon session start, INVISILINE needs to use this information to reconstruct the in-memory ATM map: *dm-crypt-hidden* cannot read and write hidden data before the map is fully reconstructed.

INVISILINE uses multiple threads to parallelize this task. Each thread issues a special request to the underlying dm-integrity layer to read only the metadata sectors (sectors that only contain the IVs of data sectors), without the corresponding public data. It then decrypts IVs and uses the result's 13th and 16th byte to identify those that store hidden data (see Figure 5). When an IV does not store hidden data, the thread skips reading the next f physical sectors.

However, when a decrypted IV stores hidden data, the thread reads the IV's offset to check if it is the first IV in the sequence of f IVs that store a logical hidden sector. This is because only the first IV in the sequence records the data's logical hidden sector number, see Figure 5.

If the IV offset is 0, INVISILINE reads the logical hidden sector number and the $VerNo$. It then uses the corresponding physical sector number to insert a (logical hidden sector number, physical sector number, $VerNo$, $d = 0$, $m = 0$) tuple to the in-memory ATM, where d and m are the deletion and modification bits, respectively. If an entry already exists for this logical hidden sector, the ATM entry's physical sector and $VerNo$ values are updated only if its $VerNo$ value is smaller than the one in the newly decrypted IV.

If the IV's offset is not zero, the thread issues a request for the first IV (at current physical sector number - offset) and applies the above procedure. Once an entry has been added to the translation table, the thread skips the subsequent f physical sectors.

Appendix B discusses support provided by INVISILINE to delete logical hidden sectors (via the TRIM command) from the hidden device. Appendix C further presents INVISingle, an optimization for single-snapshot adversaries that can inspect the user device only once.

6. Analysis

IsExecutable. Let $HideCount(\mathcal{O}_k^1)$ be a function that returns the number of unique sectors in the adversary's hidden data write requests $O_h^j = (hs_j, hd_j) \in \mathcal{O}_k^1$ in the k-th round. Further, let $PubCount(\mathcal{O}_k^1, \cup_{l=1}^{k-1}\mathcal{O}_l^1)$ be a function that returns the number of unique sector numbers sn_i that appear in the adversary's public writes $O_p^i = (sn_i, data_i) \in \mathcal{O}_k^1$ in round k, that do not appear in any of the previous rounds.

Then, the $IsExecutable(\cup_{l=1}^{k}\mathcal{O}_l^1)$ predicate with input all public and hidden writes generated by the adversary $\mathcal{A}$ up until and including round k, returns true only if $HideCount(\mathcal{O}_k^1) \leq f \cdot PubCount(\mathcal{O}_k^1, \cup_{l=1}^{k-1}\mathcal{O}_l^1)$. That is, the number of unique hidden data writes in the k-th round need to be at most equal to the storage factor f times the number of unique sectors written with public data in round k, that were not written to in any of the previous rounds.

INVISILINE Provides Plausible Invisibility. We adapt the P-INV game of § 3.3 to the INVISILINE system defined in § 5.3: The device $\mathcal{D}$ stores data in sectors and IVs associated with data sectors. Let $\mathcal{D}[i, k]$ denote the data stored on $\mathcal{D}$ for sector i after round k of the P-INV game. Thus, if sector i has been written with public data in any of the

k rounds, $\mathcal{D}[i,k] = (IV_i, E_{K_p}(IV_i, data_i))$, where IV_i is an initialization vector, and public data $data_i$ is encrypted with the key K_p and IV. Otherwise, $\mathcal{D}[i,k] = (R_1, R_2)$, for pseudorandom R_1 and R_2 used to initialize the device (§ 2).

During the k-th round (session) the adversary sends $O_k^0 = PW_k$ and $O_k^1 = PW_k \cup HW_k$, with public data write requests in PW_k of format $O_p^i = (sn_i, data_i)$ and hidden data write requests in HW_k of format $O_h^j = (hs_j, hd_j)$ chosen such that the above defined $IsExecutable(\cup_{l=1}^{k}\mathcal{O}_l^1)$ returns true. The $isExecutable$ function ensures that each hidden write in the adversary's set HW_k can be stored in the IVs of the public sectors written in the set PW_k. Thus, the only sectors that change in round k are the ones referenced by the adversary in the set PW_k. For the purpose of this proof, we make the simplifying assumption that the storage factor $f = 1$, i.e., each hidden data write fits in a single IV.

At the end of the k-th round (session) the adversary captures a snapshot of the storage device, i.e., all $\mathcal{D}[i,k]$ tuples, $i = 1..n$. Let $val(IV_i, l)$ denote the value of IV for sector i at the end of round l in the P-INV game. There are two main cases, based on the value of $\mathcal{C}$'s bit b:
(Case 0): When $b = 0$, i.e., the challenger only writes public data using OTSS, $val(IV_i, k)$ is random for all sectors i where public data has been written in the current round.

When $b = 1$, i.e., the challenger uses INVISILINE to write both public and hidden data. More specifically, for each hidden write in $\mathcal{O}_k^1$, INVISILINE first performs f public writes from $\mathcal{O}_k^1$. At the end of round k, the value of any IV_i on $\mathcal{D}$ corresponding to a sector s_i referenced in $\mathcal{O}_k^1$ can be one of the following:
(Case 1): $val(IV_i, k)$ is pseudorandom generated. This happens for public data write requests $O_p^i = (s_i, data_i)$ where no hidden data was embedded into the IV of sector number s_i in any of the $1..k$ rounds. INVISILINE emulates OTSS to encrypt the public data with a pseudorandom IV.
(Case 2): $val(IV_i, k) = E_{K_h}(s_i, [hd_j, VerNo_j = 0, PWC_i = 0, Magic])$, where the sector number s_i corresponding to IV_i is used for initialization value. This occurs after a hidden write $O_h^j = (hs_j, hd_j) \in \mathcal{O}_k^1$ where $ATM[hs_j] = null$, i.e., this is the first time the adversary wrote data at hidden sector hs_j, and the value of IV_i after round $k-1$, $val(IV_i, k-1)$ was pseudorandom.
(Case 3): $val(IV_i, k) = E_{K_h}(s_i, [hd_j, VerNo_j + 1, PWC_i = 0, Magic])$, see Figure 6 (middle). This occurs for a hidden data overwrite $O_h^j = (hs_j, hd_j) \in \mathcal{O}_k^1$, where hidden sector hs_j was written by the adversary in any of the rounds $1..k-1$, and $ATM[hs_j] = (s_j, VerNo_j, d_j, m_j)$. $val(IV_i, k-1)$ was pseudorandom, i.e., this is the first time the IV_i of sector $\mathcal{D}[i]$ is used to hide data.
(Case 4): $val(IV_i, k) = E_{K_h}(s_i, [hd_j, VerNo_j, PWC_i + 1, Magic])$, see Figure 6 (bottom). This occurs for an adversary access $O_p^i = (s_i, data_i) \in \mathcal{O}_k^1$ that overwrites the public data at the sector s_i corresponding to IV_i, i.e., $val(IV_i, k-1) = E_{K_h}(hd_j, VerNo_j, PWC_i, Magic)$.

The adversary has access to all $\mathcal{D}[i,l]$ values, $\forall i = 1..n, l = 1..k$. The use of sector numbers s_i as initialization vectors for hidden data-storing IVs ensures that the adversary cannot correlate hidden data-storing IVs from different sectors Thus, to gain an advantage in the game, the adversary needs to distinguish between Case 0 (bit b = 0) and either of the Cases 1 .. 4 (bit b = 1).

Case 1 is by definition identical to Case 0. In Case 2 vs. Case 0 and Case 3 vs. Case 0, the adversary needs to distinguish $E_{K_h}(s_i, [hd_j, 0, 0, Magic])$ and $E_{K_h}(s_i, [hd_j, VerNo_j + 1, PWC_i, Magic])$ respectively, from the output of a pseudorandom generator. An adversary with non-negligible advantage in distinguishing between these cases, can be used to break the semantic security of INVISILINE's encryption function (i.e., AES, § 3.1), i.e., by building a polynomial time algorithm with non-negligible advantage in distinguishing the output of AES from a pseudorandom value with the same length.

Further, in Case 4 vs. Case 0, the adversary needs to distinguish $val(IV_i, k) = E_{K_h}(s_i, [hd_j, VerNo_j, PWC_i + 1, Magic])$ from a pseudorandom value and from $val(IV_i, k-1) = E_{K_h}(s_i, [hd_j, VerNo_j, PWC_i, Magic])$. Given that PWC_i changes in at least one bit, an adversary with non-negligible advantage in distinguishing Case 4 from Case 0 will break the indistinguishability of encryptions of INVISILINE's encryption function (i.e., AES, § 3.1).

Since plausible invisibility implies plausible deniability and OTSS readability (§ 3.3), INVISILINE also provides plausible deniability and OTSS readability.

Data Non-interference. INVISILINE provides non-interference by construction. The $Write\_Public\_Data$ function (Algorithm 3) ensures that hidden data is not erased when the public data stored in the corresponding public sector is updated (see Hidden Data non-interference § 5.2). It achieves this by incrementing the PWC field of the cleartext IV (at most once per session) before re-encrypting the updated hidden data-storing IV, which is then used to re-encrypt the updated public data. The public data non-interference (§ 3.2) holds since INVISILINE provides OTSS readability which in turn implies accessibility of all public data.

Security Non-interference. INVISILINE prevents security interference (§ 5.2) by ensuring that the IVs used for encrypting public data are not reused. First, since each of the f IVs of a logical hidden sector includes a different offset (see Figure 5), each IV is different, even when repetitions occur within the hidden data. Second, since each IV includes a $VerNo$ value, all the IVs that store the different versions of a logical hidden sector are different, even if the hidden data is the same in all the versions. Third, the use of public sector numbers as initialization vectors to encrypt hidden data-storing IVs ensures that hidden data-storing IVs from different sectors will be different even if the hidden data, version number and offset are the same.

Thus, each hidden-data storing IV is generated by encrypting material that differs in at least one bit from any other hidden-data storing IV on the disk. An adversary that can distinguish with non-negligible advantage an INVISILINE IV storing hidden data from a pseudorandom string of the same length or from an IV with either the same version number, same offset or same value, can be used to break the semantic security of INVISILINE's encryption function (i.e., AES, § 3.1).

Flash-Based Data Duplication Attack. Any updates to data in SSDs are written to new locations for wear leveling. INVISILINE encrypts public sectors at least twice during a session. First with a pseudorandom IV and then with encrypted hidden data as IV. An adversary with access to raw flash data will find multiple copies of the same public data encrypted with different IVs, to guess that INVISILINE was used to store hidden data. However, this is indistinguishable from usage with vanilla dm-crypt for e.g., when updating two sectors containing same data or when reverting to a previous version of sector data.

Further, and importantly, this paper assumes an adversary that cannot access the raw data on the user device (§ 3.3). On modern high-density SSDs access to raw data is virtually impossible without significant SSD manufacturer support (e.g., by injecting access primitives into the firmware). Modern SSDs have flash chips BGA soldered with hundreds of high density pins, making it virtually impossible for reliable access to raw pages, even if they are not destroyed during de-soldering attempts. Assuming SSD manufacturer collusion would break the security model, where the adversary can only access storage snapshots: the manufacturer firmware in effect is an online witness to all I/O on the device.

7. Implementation

INVISILINE was implemented as a Linux device mapper dm-crypt target. It exposes two logical block devices, one for hidden and one for public data. All changes were made on the Linux 5.19 kernel image.

Figure 3 shows the architecture of INVISILINE. The dm-crypt kernel module in INVISILINE supports two components namely *dm-crypt hidden* and *dm-crypt public* based on how the device mapper target has been instantiated. When cryptsetup is invoked with *open* action for a physical device, two dm-crypt logical devices (hidden and public) and one dm-integrity logical device gets created. For e.g., if *test* is provided as the mapping name with cryptsetup for a physical device, the device mapper dm-crypt hidden device is created at */dev/mapper/test_pd*, and dm-crypt public device is created at */dev/mapper/test*. The dm-integrity logical device is created at */dev/mapper/test_dif*. Both *dm-crypt hidden* and *dm-crypt public* operate on the same underlying dm-integrity device and share data structures and keys, allowing each to operate on the other's data and metadata.

The dm-integrity layer, mapped at */dev/mapper/test_dif* is transparent to whether the data operations coming from the dm-crypt layer pertain to hidden or public data. Separate file systems, e.g., EXT4, can be mounted on the hidden and public block devices to store files and directories.

INVISILINE uses *cryptsetup*, run with *luksFormat*, to specify the AES-XTS block cipher with *random* ivmode. The *integrity* option is set to *none* to ensure that integrity metadata only stores 16 bytes of persistent IV per sector.

OP	dm-crypt	INVISingle Pub	INVISingle Hidden	INVISILINE Pub	INVISILINE Hidden
Read	12.29	13.47	48.88	11.89	76.12
Write	19.39	43.34	81.76	65.62	112.73
Random Read	18.41	21.25	104.71	17.14	137.19
Random Write	18.3	45.60	111.17	67.12	172.34

TABLE 2. LATENCY (IN MS) COMPARISON OF INVISILINE FOR SINGLE AND MULTI SNAPSHOT ADVERSARIES AGAINST VANILLA DM-CRYPT. WE OBSERVE HIGH LATENCY (112MS) FOR HIDDEN WRITES IN MULTI-SNAPSHOT CASE COMPARED TO OTHER APPROACHES AS IT CONSTITUTES READING AND DECRYPTING MULTIPLE PUBLIC SECTORS AND THEN RE-ENCRYPTING THEM WITH NEW HIDDEN DATA.

OP	dm-crypt	INVISingle Pub	INVISingle Hidden	INVISILINE Pub	INVISILINE Hidden
Read	71.4	62.5	19.96	66.7	12.78
Write	45.5	22.5	11.12	14.43	8.73
Random Read	47.6	42.2	9.30	50.75	7.49
Random Write	47.6	21.05	8.69	13.8	5.74

TABLE 3. COMPARISON OF IOPS (1000 I/O OPERATIONS PER SEC) OF INVISILINE FOR SINGLE AND MULTI SNAPSHOT ADVERSARIES AGAINST DM-CRYPT. INVISILINE AND INVISINGLE WRITE HIDDEN DATA WITH INVISIBILITY AND PLAUSIBLE DENIABILITY AT 19% AND 24% RESPECTIVELY OF THE VANILLA DM-CRYPT IOPS.

8. Evaluation

This section evaluates INVISILINE and compares it against the vanilla dm-crypt/dm-integrity and INVISingle. The experiment platform is a laptop running Intel Core i7 with 6 cores, 16GB DRAM and 512GB SSD NVMe storage. The experiments were performed using the flexible I/O tester (fio) [38], on a 40GB device file, with 512B sector size, on sequential and random workloads. All measurements reported are averages over 5 independent experiments.

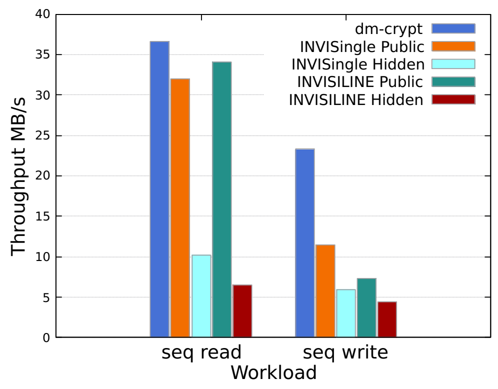


Figure 7. Throughput for sequential workload in INVISingle is higher when compared to INVISILINE because of INVISILINE IO and CPU overhead.

Hidden Device Size. The size of the hidden device depends on the size of the physical device and the storage factor f. If the size of the physical device is s, then the size of the hidden device is $\approx s/f$. Thus, for a 1TB physical device, INVISILINE ($f = 52$) generates a 19.23GB hidden device, while INVISingle($f = 40$) generates a 25GB hidden device.

Figure 7 shows the INVISILINE and INVISingle sequential read and write throughput for hidden data, and the vanilla dm-crypt throughput for sequential public data. Figure 8 shows the INVISILINE and INVISingle read and write throughput for hidden data at random logical hidden

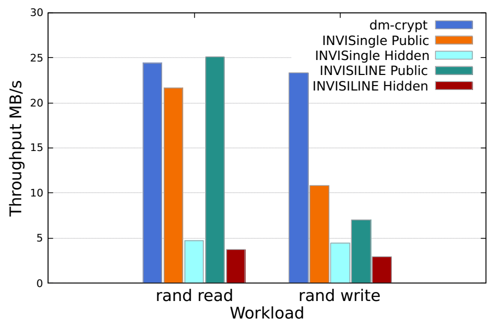


Figure 8. Throughput for random workload is marginally lower compared to sequential operations owing to non sequential access.

sectors, and the vanilla dm-crypt read and write throughput for public data at random addresses. Tables 2 and 3 also show the latency and IOPS values for INVISILINE, INVISingle and vanilla dm-crypt, for both sequential and random workloads. In the following we discuss performance separately for operations on public and hidden data.

Public Data Operations. INVISILINE and INVISingle achieve similar throughput for sequential (34 MB/s) and random (24 MB/s) public reads when compared to vanilla dm-crypt. Public writes in dm-crypt are higher than in INVISingle. This is because public writes first read the sector (along with IV) to determine if hidden data is present. This incurs additional overhead in terms of IO and CPU. Sequential public writes in INVISingle (11.5 MB/s) are 56% faster than in INVISILINE (7.37 MB/s), whereas random public writes are 44% faster in INVISingle (10.8 MB/s vs 7.5 MB/s). This is because (1) the storage factor in INVISILINE is larger than in INVISingle, and (2) INVISILINE imposes additional I/O and CPU overhead to determine if the hidden data stored in a sector's IV is the latest version.

Hidden Data Operations. Hidden operations in INVISILINE achieve a throughput of 6.55 MB/s for sequential reads and 4.46 MB/s for sequential writes. Hidden operations impose higher I/O and CPU overhead since they involve a factor of $f = 52$ physical sectors for every logical sector. Hidden reads decrypt the IVs of physical sectors, while hidden writes decrypt the data of physical sectors and encrypt it back with hidden data as IV. INVISILINE imposes additional overhead to maintain (1) the *freelist*, (2) the mapping table (ATM) and (3) the dirty sector list. Each logical hidden sector is mapped to a different set of contiguous physical sectors. Thus, even when accessing contiguous logical hidden sectors, separate I/O requests for each set of physical sectors have to be issued to the underlying layer.

The INVISingle optimization improves the sequential read throughput by 55% (10.21 MB/s) and the sequential write throughput by 34% (5.98 MB/s), when compared to INVISILINE. This is due to its lower storage factor, $f = 40$. Further, INVISingle can perform a single IO request for all the contiguous logical sectors. Random reads and random writes access random logical sectors and therefore produce lower throughput than the sequential counterparts.

Hidden ATM Table: Size and Reconstruction Overhead. Each entry in the hidden ATM table takes 12 bytes to store the logical hidden sector, the physical sector number, the version number, and the deletion and the modified bits. For a disk of size D, the maximum size of the ATM table is $12 \times D/(f \times 512)$. That is, for a 1TB disk and a storage factor f of 52, the table can grow up to 450.7MB.

At the start of each session, INVISILINE scans the integrity metadata sectors from the underlying device to reconstruct the ATM table (§ 5.3). In order to avoid the overhead of reading both the sector data and its metadata, INVISILINE uses a modified dm-integrity module to retrieve only the metadata region without reading the actual data sectors. INVISILINE spawns multiple threads (12 threads for 6 core CPU) for reading IVs in parallel and also avoids reading each IV in the metadata region. In our experiments it took 1.03s to scan 81,184,760 IVs for a 40 GB disk and fully reconstruct the ATM. For a 1TB disk this scales to a start-up overhead of 26.36s at the beginning of each session.

9. Discussion and Limitations

False Positives. The magic value of IVs (Figure 5 and 9) allows INVISILINE and INVISingle to determine if an IV stores hidden data. To reduce false positives (FPR = 2^{-8}) in case of match, INVISILINE also checks whether the IV offset has a value between 0 and $f - 1$. Further, it fetches another IV in the sequence and verifies that it has the magic value set, an appropriate offset (0 or 1 respectively), and the same PWC as the first IV. This reduces the false positive rate to less than 2^{-34}.

Overflowing PWC and VerNo. If a write operation would lead to a $VerNo$ or PWC counter overflow, INVISILINE returns an IO error. While theoretically this difference between INVISILINE and dm-crypt could be exploited by an adversary to break invisibility, the attack would require significant time: assuming an average of ten sessions per day, it will take INVISILINE 17.95 years to overflow the $VerNo$ and PWC counters. This exceeds twice the expected lifetime of most laptops and servers [39], [40].

Hidden Data Can Be Accidentally Erased. If the disk is unmounted and INVISILINE is uninstalled, the user can still read public data if the vanilla dm-crypt is installed. However, any write operations to the public data using vanilla dm-crypt may overwrite the hidden data permanently. That is, data hidden in IVs overwritten by dm-crypt cannot be recovered even if INVISILINE is re-installed.

10. Conclusion

We introduced INVISILINE, the first *invisible* disk encryption system. INVISILINE achieves invisibility, even in the presence of multi-snapshot adversaries, by hiding data in IVs used to encrypt public data, in a manner compatible with dm-crypt, a popular disk encryption software. We formally defined invisibility for plausibly-deniable storage. INVISILINE was implemented and can securely and invisibly hide 19GB of data on a 1TB disk, with hidden data throughput exceeding multiple MB/s.

References

[1] “Cloud act,” ”https://www.justice.gov/dag/page/file/1152896/download”.

[2] “Promoting Public Safety, Privacy, and the Rule of Law Around the World: The Purpose and Impact of the CLOUD Act,” Department of Justice, https://www.justice.gov/opa/press-release/file/1153446/download, 2019.

[3] “The CLOUD Act Data Access Agreement – 10 Things That U.S. Telecommunications Companies Need to Know Now,” Wiley Law,https://www.wiley.law/alert-The-CLOUD-Act-Data-Access-Agreement-10-Things-That-US-Telecommunications-Companies-Need-to-Know-Now, 2022.

[4] 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.

[5] R. Westhead. (2012) How a syrian refugee risked his life to bear witness to atrocities. http://www.thestar.com/news/world/2012/03/14/how_a_syrian_refugee_risked_his_life_to_bear_witness_to_atrocities.html.

[6] “Truecrypt,” ”https://truecrypt.sourceforge.net/”.

[7] R. P. W. J. Assange and S. Dreyfus, “Rubberhose:cryptographically deniable transparent disk encryption system,” 1997, ”http://marutukku.org”.

[8] “Shufflecake: plausible deniability for multiple hidden filesystems on Linux,” https://shufflecake.net/.

[9] 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*. ACM, 2014, pp. 203–214.

[10] C. C. Anrin Chakraborti and R. Sion, “DataLair: Efficient block storage with plausible deniability against multi-snapshot adversaries,” *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 3, 2017.

[11] C. Chen, A. Chakraborti, and R. Sion, “Pd-dm: An efficient locality-preserving block device mapper with plausible deniability.” *Proc. Priv. Enhancing Technol.*, vol. 2019, no. 1, pp. 153–171, 2019.

[12] ——, “{PEARL}: Plausibly deniable flash translation layer using {WOM} coding,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1109–1126.

[13] “dm-crypt,” ”https://wiki.archlinux.org/title/dm-crypt”.

[14] A. Houmansadr, C. Brubaker, and V. Shmatikov, “The parrot is dead: Observing unobservable network communications,” in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 65–79.

[15] “Windows bitlocker,” ”https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview”.

[16] “Macos filevault,” ”https://support.apple.com/en-us/HT204837”.

[17] “ecryptfs,” ”https://www.ecryptfs.org/home”.

[18] “Encfs,” ”https://wiki.archlinux.org/title/EncFS”.

[19] “Veracrypt,” ”https://www.veracrypt.fr/en/Home.html”.

[20] “dm-crypt Drive preparation,” https://wiki.archlinux.org/title/Dm-crypt/Drive_preparation.

[21] “cryptsetup,” ”https://gitlab.com/cryptsetup/cryptsetup”.

[22] “Luks,” ”https://gitlab.com/cryptsetup/cryptsetup/-/wikis/LUKS-standard/on-disk-format.pdf”.

[23] “Practical malleability attack against CBC-Encrypted LUKS partitions,” https://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/, 2013.

[24] O. Goldreich, *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[25] ——, *Foundations of cryptography: volume 1, basic tools*. Cambridge university press, 2001.

[26] P. Gasti, G. Ateniese, and M. Blanton, “Deniable cloud storage: sharing files via public-key deniability,” in *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, 2010, pp. 31–42.

[27] A. Skillen and M. Mannan, “On implementing deniable storage encryption for mobile devices,” 2013.

[28] B. Chang, F. Zhang, B. Chen, Y. Li, W.-T. Zhu, Y. Tian, Z. Wang, and A. Ching, “MobiCeal: Towards secure and practical plausibly deniable encryption on mobile devices,” in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 454–465.

[29] A. Zuck, U. Shriki, D. E. Porter, and D. Tsafrir, “Preserving hidden data with an ever-changing disk,” in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 50–55.

[30] A. D. McDonald and M. G. Kuhn, “StegFS: A steganographic file system for Linux,” in *Information Hiding*. Springer, 1999, pp. 463–477.

[31] C. Chen, A. Chakraborti, and R. Sion, “Infuse: Invisible plausibly-deniable file system for nand flash.” *Proc. Priv. Enhancing Technol.*, vol. 2020, no. 4, pp. 239–254, 2020.

[32] C. Chen, X. Liang, B. Carbunar, and R. Sion, “Sok: Plausibly deniable storage,” *CoRR*, vol. abs/2111.12809, 2021. [Online]. Available: https://arxiv.org/abs/2111.12809

[33] R. Anderson, R. Needham, and A. Shamir, “The steganographic file system,” in *Information Hiding*. Springer, 1998, pp. 73–82.

[34] H. Pang, K.-L. Tan, and X. Zhou, “StegFS: A steganographic file system,” in *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 2003, pp. 657–667.

[35] 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*. ACM, 2010, pp. 317–326.

[36] F. Douglis and J. Ousterhout, “Log-structured file systems,” in *COMPCON Spring’89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers*. IEEE, 1989, pp. 124–129.

[37] “A robust flash file system since 2002,” ”https://yaffs.net/”.

[38] “fio flexible i/o tester,” ”https://fio.readthedocs.io/en/latest/”.

[39] “What is the Average Lifespan of a Computer?” HP, https://www.hp.com/in-en/shop/tech-takes/post/average-computer-lifespan, 2022.

[40] N. Cooper, “Server Lifespan: How Long Does It Takes?” https://www.promax.com/blog/how-long-do-servers-last, 2019.

Appendix A. Plausible Deniability

The following adapts the IND-PD (plausible deniability) game defined in [11] to the notations used in this paper (§ 3).

- The challenger $\mathcal{C}$ and adversary $\mathcal{A}$ agree on the of HideSys on device $\mathcal{D}$. $\mathcal{C}$ chooses public and hidden encryption keys K_p and K_h using a security parameter s. $\mathcal{C}$ sends K_p to $\mathcal{A}$.
- $\mathcal{C}$ selects a random bit b.
- $\mathcal{A}$ and $\mathcal{C}$ engage in the following r rounds, where $k = 1..r$ is the current round:
  - $\mathcal{A}$ selects a set of arbitrary writes to public data, $\mathcal{O}_k^0 = \{\mathcal{O}_p^i\}$, and arbitrary writes to hidden data,

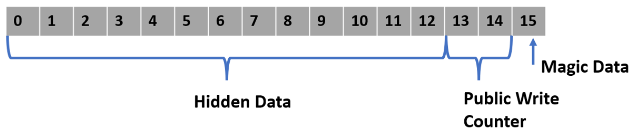


Figure 9. IV format for single snapshot adversary. Two bytes are reserved for PWC and one for Magic data, leaving 13B for hidden data.

$\{\mathcal{O}_h^j\}$, such that there exists an access pattern $\mathcal{O}_k^1 = \mathcal{O}_k^0 \cup \{\mathcal{O}_h^j\}$ where $IsExecutable(\cup_{l=1}^{k}\mathcal{O}_l^1) = True$.
  - $\mathcal{A}$ sends $\mathcal{O}_k^0$ and $\mathcal{O}_k^1$ to $\mathcal{C}$.
  - $\mathcal{C}$ executes $\mathcal{O}_k^b$ on device $\mathcal{D}$.
  - $\mathcal{C}$ sends a snapshot of $\mathcal{D}$ to $\mathcal{A}$.
- $\mathcal{A}$ outputs b', $\mathcal{A}$'s guess for b.

The advantage of adversary $\mathcal{A}$ in the IND-PD game is $Adv(\mathcal{A}) = |P(b' = b) - P(b' \neq b)|$. A system provides *plausible deniability* if any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ has only negligible advantage in the IND-PD game.

Appendix B. Sector Deletion via TRIM

Most SSDs support the TRIM command which allows the OS to inform the device which data blocks are no longer needed and can be deleted. Trimming enables SSDs to efficiently handle garbage collection and wear-leveling. INVISILINE provides support to delete logical hidden sectors (via the TRIM command) from the hidden device. First, it uses the freelist to find an available public write to a single physical sector. It then generates a special *deletion IV* for the sector, that consists of the logical hidden sector number (4B), the $VerNo$ value incremented from the value currently stored in the in-memory ATM table (2B), the PWC (2B) set to 0, and a special magic value indicating deletion (1B). Further, it increments the version number $VerNo$ of the entry in the ATM table corresponding to the deleted logical hidden sector, and marks it as deleted (set it d bit to 1). Subsequently, a read request for a sector whose corresponding d bit is set returns error.

During the ATM table reconstruction at the start of the session, if INVISILINE encounters a *deletion IV* for a logical sector, it marks the sector as deleted in the table by setting its bit d to 1. In future, if there is a write for that sector, it updates the map with the new physical sector number and incremented version number and also resets the deleted bit d. The incremented version number ensures that during map reconstruction, INVISILINE resets the deleted bit if it encounters a higher $VerNo$ value than the one found in the *deletion IV*. This also ensures that the public write would reclaim the *deletion IV* since the map now contains an updated version number for that logical sector.

Appendix C. INVISingle: Single-Snapshot Optimization

We now present INVISingle, an INVISILINE optimization for single-snapshot adversaries, that can inspect the user device only once. Multiple commercial disk encryption tools assume single-snapshot adversaries, e.g., TrueCrypt [6], Rubberhose [7], or Shufflecake [8]. In the following we focus on the INVISingle solution.

A single-snapshot adversary implies at most one user session. In turn, this eliminates the need for the user to generate public writes to store hidden data. Further, it simplifies the requirements for the IV format, eliminates the need of maintaining an ATM table, freelist and dirty list, increases the size of data that can be hidden on the disk, and improves the performance of accessing hidden data. In the following we detail these changes.

IV Format. The IV format used by INVISingle to store encrypted data is illustrated in Figure 9. To prevent public data overwrites from erasing hidden data stored in the corresponding IVs, INVISingle uses the one byte magic value and two byte PWC to indicate if the IV is used to store hidden data. Thus, the factor f used in Algorithms 4 and 5 is $\lceil 512/13 \rceil$ = 40.

Further, since the adversary can only inspect the device once, INVISingle does not need to maintain an ATM table. Instead, it uses a fixed mapping of logical hidden sectors to physical sectors: logical hidden sector i maps to the f physical sectors $i \times f$ to $(i + 1) \times f - 1$.

Algorithm 4 $Write\_Hidden\_Data\_Single(s_n, s_d)$.

```
Require: $s_n$, $s_p$          ▷ write hidden sector number $s_n$ with data $s_d$
 1: $f := 40$
 2: $MAGIC := 0xAA$
 3: $ps\text{-}start_n := f * s_n$        ▷ Compute starting physical sector number
 4: $(Enc_{data}, IV_{data}) := Phy\_read(ps\text{-}start_n, f)$   ▷ Read $f$ sectors
 5: $data := decrypt(Enc_{data}, K_p, IV_{data})$       ▷ Decrypt public data
 6: for $i \in \{0, \ldots, f-1\}$ do
 7:     $IV_{data}[i] := buildIV(s_d, 0, MAGIC)$          ▷ Construct IV
 8:     $IV_{data}[i] := encrypt(IV_{data}[i], K_h, ps\text{-}start_n + i)$  ▷ Encrypt data
 9: end for
10: $Enc_{data} := encrypt(data, K_p, IV_{data})$   ▷ Re-encrypt public data
11: $Phy\_write(ps\text{-}start_n, factor, Enc_{data}, IV_{data})$     ▷ Write data
```

Write Hidden Data. Algorithm 4 shows the pseudocode for writing hidden data for a logical hidden sector. The code is executed by the *dm-crypt-hidden* instance. To ensure non-interference (§ 3.2), on every logical hidden sector update, the write function first extracts the public data from the sectors, and re-encrypts it with the new hidden data as IV. More specifically, the function computes the first physical sector number corresponding to the logical sector number in the request (line 3). It then issues a read request (line 4) to the underlying dm-integrity layer for those sectors. Once *dm-crypt-hidden* receives the data along with the corresponding IVs, it decrypts the public data using the IVs and K_p (line 5). It then constructs each of the f IVs using the hidden data, $PWC = 0$ and magic value (line 7), and encrypts each IV with K_h and the physical sector number as IV (line 8). The function then re-encrypts the public data using resulting IV (line 10), then issues a write request to the dm-integrity layer with the new data (line 11). Thus, at the end of the write request, the hidden data is updated without imposing changes to the corresponding public data.

Algorithm 5 $Read\_Hidden\_Data\_Single(s_n)$

```
Require: $s_n$                                   ▷ read hidden sector number $s_n$
 1: $f := 40$
 2: $ps\text{-}start_n := f * s_n$              ▷ Compute starting physical sector number
 3: $(Enc_{data}, IV_{data}) := Phy\_read(ps\text{-}start_n, f)$   ▷ Read data and IV
 4: for $i \in \{0, \ldots, f-1\}$ do
 5:     $IV_{data}[i] := decrypt(IV_{data}[i], K_h, ps\text{-}start_n + i)$   ▷ Decrypt
 6: end for
 7: return $IV_{data}$
```

Read Hidden Data. Algorithm 5 shows the pseudocode for reading hidden data stored at a logical sector number given as input. The function issues read requests to the corresponding physical sectors. Once it receives the public data and the metadata, it decrypts the metadata using K_h and returns it, discarding the public data.

Write and Read Public Data. Algorithm 6 shows the pseudocode for writing public data s_d at physical sector s_n. To again ensure non-interference, i.e., ensure that the public write does not overwrite its corresponding hidden data with a random IV, the function first reads the data and IV from the underlying layer (line 3) and decrypts the IV using K_h (line 4). To avoid false positives, the function also reads its adjacent IV (at sector $s_n + 1$ or $s_n - 1$) (lines 5-10). It then checks both the IVs for magic data and matching PWC (line 13-15). If the comparison succeeds (the IV holds hidden data), it increments the PWC (line 16) and re-encrypts the IV (line 17). Otherwise, it generates a pseudorandom IV (line 19). The resultant IV is then used to encrypt the public data (line 21). Finally, the function uses dm-integrity to write the encrypted data and IV (line 22). INVISingle uses the the vanilla dm-crypt code to read public data: it reads the physical data along with the IV, decrypts the data using K_p and returns the result.

Invisibility against single-snapshot adversary. INVISingle is a special case of INVISILINE in which the adversary can only take one device snapshot. INVISingle uses a fixed mapping of logical hidden sectors to physical sectors. An IV_i at sector s_i that stores hidden data has value $val(IV_i) = E_{K_h}(s_i, [hd_j, PWC, Magic])$, see Figure 9. The value stored at sector s_i can be either $E_{K_p}(IV_i, data_i)$ if s_i stores public data, orpseudorandom (generated when device $\mathcal{D}$ was initialized before its first use) if no public data was written. In both cases, an adversary with non-negligible advantage in the P-INV game (r = 1) can be used to construct an adversary with non-negligible advantage in distinguishing the output of INVISILINE's encryption function from apseudorandom value. Incrementing PWC on every public write to a sector whose corresponding IV holds hidden data ensures that the IV used for encrypting public data is not re-used. This guarantees the security non-intereference.

Algorithm 6 $Write\_Public\_Data\_Single(s_n, s_d)$

```
Require: $s_n$, $s_p$                    ▷ write public sector number $s_n$ with data $s_d$
 1: $f := 40$
 2: $MAGIC := 0xAA$
 3: $(Enc_{data}, IV_{data}) := Phy\_read(s_n, 1)$            ▷ Get data and IV
 4: $IV_{data} := decrypt(IV_{data}, K_h, s_n)$              ▷ Decrypt hidden data
 5: if $(s_n \bmod f) \neq 0$ then
 6:     $(Enc_{data}, IV_{abj}) := Phy\_read(s_n - 1, 1)$
 7:     $IV_{adj} := decrypt(IV_{adj}, K_h, s_n - 1)$
 8: else
 9:     $(Enc_{data}, IV_{adj}) := Phy\_read(s_n + 1, 1)$
10:     $IV_{adj} := decrypt(IV_{adj}, K_h, s_n + 1)$
11: end if                                                    ▷ Get adjacent IV
12: if (
13:     $IV_{data}[15] = MAGIC$   AND
14:     $IV_{adj}[15] = MAGIC$   AND
15:     $IV_{data}[13:14] = IV_{adj}[13:14]$  ) then
16:     $IV_{data}[13:14] := IV_{data}[13:14] + 1$               ▷ PWC ++
17:     $IV_{data} := encrypt(IV_{data}, K_h, s_n)$              ▷ Encrypt hidden
18: else
19:     $IV_{data} := get\_random\_bytes(16)$     ▷ Generate pseudorandom IV
20: end if
21: $Enc_{data} := encrypt(s_d, K_p, IV_{data})$              ▷ Encrypt public data
22: $Phy\_write(s_n, 1, Enc_{data}, IV_{data})$               ▷ Write back public data
```