Barker *et al.* *J Surveill Secur Saf* 2021;2:42-65
DOI: 10.20517/jsss.2020.22

Journal of Surveillance, Security and Safety

Original Article

Open Access

Rethinking the adversary and operational characteristics of deniable storage

Austen Barker, Yash Gupta, James Hughes, Ethan L. Miller, Darrell D. E. Long

Baskin School of Engineering, University of California, Santa Cruz, CA 95064, USA.

Correspondence to: Austen Barker, Baskin School of Engineering, University of California, Santa Cruz, 1156 High Street, Santa Cruz, CA 95064, USA. E-mail: atbarker@ucsc.edu

How to cite this article: Barker A, Gupta Y, Hughes J, Miller EL, Long DDE. Rethinking the adversary and operational characteristics of deniable storage. *J Surveill Secur Saf* 2021;2:42-65. https://dx.doi.org/10.20517/jsss.2020.22

Received: 8 Jul 2020 **First Decision:** 7 Dec 2020 **Revised:** 21 Jan 2021 **Accepted:** 22 Feb 2021 **Published:** 27 May 2021

Academic Editor: Xiaofeng Chen **Copy Editor:** Xi-Jun Chen **Production Editor:** Xi-Jun Chen

Abstract

Aim: With the widespread adoption of disk encryption technologies, it has become common for adversaries to employ coercive tactics to force users to surrender encryption keys. For some users, this creates a need for hidden volumes that provide plausible deniability, the ability to deny the existence of sensitive information. Previous deniable storage solutions only offer pieces of an implementable solution that do not take into account more advanced adversaries, such as intelligence agencies, and operational concerns. Specifically, they do not address an adversary that is familiar with the design characteristics of any deniable system.

Methods: We evaluated existing threat models and deniable storage system designs to produce a new, stronger threat model and identified design characteristics necessary in a plausibly deniable storage system. To better explore the implications of this stronger adversary, we developed Artifice, the first tunable, operationally secure, self repairing, and fully deniable storage system.

Results: With Artifice, hidden data blocks are split with an information dispersal algorithm such as Shamir Secret Sharing to produce a set of obfuscated carrier blocks that are indistinguishable from other pseudorandom blocks on the disk. The blocks are then stored in unallocated space of an existing file system. The erasure correcting capabilities of an information dispersal algorithm allow Artifice to self repair damage caused by writes to the public file system. Unlike preceding systems, Artifice addresses problems regarding flash storage devices and multiple snapshot attacks through simple block allocation schemes and operational security measures. To hide the user's ability to run a deniable system and prevent information leakage, a user accesses Artifice through a separate OS stored on an external Linux live disk.

Conclusion: In this paper, we present a stronger adversary model and show that our proposed design addresses the primary weaknesses of existing approaches to deniable storage under this stronger assumed adversary.

Keywords: Deniable storage, steganography, file systems

1. INTRODUCTION

As everyday use of encryption for personal data storage becomes more common, adversaries are forced to turn to alternative means to compromise the confidentiality of data. As a result, the possession of an encrypted file or disk can expose a user to coercive cryptanalysis tactics, or worse[1]. In such situations, e.g. crossing the border of a country with a repressive regime, it is necessary for the user to establish plausible deniability - the ability to deny the existence of sensitive information.

The heightened risk inherent to carrying encrypted information pushes individuals to extreme methods to exfiltrate data from dangerous or restricted environments. For example, in 2011, a Syrian engineer smuggled a micro SD card hidden inside of a self-inflicted wound in order to expose information about atrocities in Hama[2]. It is also increasingly common for nations and law enforcement to legally obligate disclosure of encryption keys when requested by authorities under the authority of key disclosure laws[3]. These alarming trends highlight the need for dependable deniable storage technologies to better safeguard at-risk individuals.

Since carrying encrypted files or dedicated hardware can be inherently suspicious, a deniable storage system must be hidden within a publicly visible file system or storage device to maintain plausible deniability. It is highly suspicious if there are visible drivers or firmware, unconventional partitioning schemes, excess unusable space in a file system, or unexplained changes to the disk's free space. To avoid suspicion, the hidden volume must operate in such a way that the encapsulating file system and operating system are entirely unaware of the hidden file system's existence, even when faced with a detailed forensic examination.

Over the course of two decades, various systems have been designed in an attempt to address this problem. In the process of navigating the compromises inherent to plausibly deniable storage, each of these systems demonstrates distinctive "tells" that enable a skilled adversary with knowledge of their design to quickly discover them. Some systems, such as StegFS[4], do not disguise hidden data writes as deniable operations, enabling an adversary to compare two images of the disk and find the hidden volume in a *multiple snapshot attack*[5]. On-the-fly-encryption systems that include hidden volumes such as TrueCrypt[6] fail to adequately address information leakage to a public volume through common programs such as word processors[7]. While they hide data and disguise writes to a hidden volume, oblivious RAM (ORAM)-based[8,9] systems such as HIVE[10] or Datalair[11] incur significant performance penalties for both hidden and public volumes — presenting a "fingerprint" detectable through basic benchmarking techniques. No existing approach successfully addresses deniability for the existence of the software or driver necessary for accessing a deniable storage system. Previous work also rarely or inadequately discusses operational security characteristics that would contribute towards a concept of operations. Given the realistic assumption that a skilled adversary knows these weaknesses, using these existing solutions would put a user at significant risk.

These demonstrated weaknesses require a deniable storage system to meet a series of requirements: effectively hide existence of the data, prevent information leakage to the public elements of a system, protect against overwrite by user behavior, disguise changes to the physical storage media, and hide or disguise the means used to access the hidden volume.

We take a step toward the goal of applying deniable storage systems to safeguard users by addressing the above requirements with Artifice, a block device that provides functional plausible deniability for both hidden data and the Artifice driver itself. To access a hidden volume, the user boots into a separate, Artifice-aware operating system through a Linux live USB drive which provides effective isolation from the host OS. Unlike earlier systems, this does not leave behind suspicious drivers on the user's machine and mitigates the impact of malware and information leakage to the public volume.

A user's writes to the Artifice volume are split through an *information dispersal algorithm (IDA)* such as Shamir Secret Sharing[12] to generate pseudorandom *carrier blocks*. The carrier blocks are then stored in the unallocated space of the public file system, which is also assumed to be full of pseudorandom blocks due to whole drive encryption, a secure deletion utility, or similar means.

As the public file system cannot be aware of Artifice's existence, Artifice must protect itself from damage due to overwrites by public operations. IDAs provide Artifice overwrite tolerance through the inclusion of redundant carrier blocks and enable a self repair process whenever the user boots the Artifice-aware OS. The overwrites still occur, but they do not cause irreparable harm.

Artifice's metadata locations are algorithmically generated through hashing a pass-phrase that must be supplied to find and use the hidden volume. Without the correct pass-phrase, an Artifice instance is indistinguishable from the rest of the free space on a disk. Unlike previous approaches, Artifice addresses the unique challenges posed by modern flash devices through the careful management of TRIM operations. Many systems aim to address the problem of multiple snapshot attacks in which an adversary deduces the existence of a hidden volume through comparison of multiple images of the disk taken at different times. Artifice tackles this issue through writing hidden blocks under the guise of a suitable deniable operation, such as defragmentation and routine file deletion, or through operational security measures.

In summary, Artifice provides a plausibly deniable storage system that effectively hides data in the free space of an existing file system while ensuring the integrity of the hidden data, defends against information leakage and malware, and hides the user's ability to run a deniable storage system - all while performing well enough for everyday tasks without affecting the behavior of the public system.

2. RELATED WORK

Over the past few decades, there have been several attempts at creating a plausibly deniable storage system (shown in Table 1). While these existing systems all claim to provide plausible deniability, they commonly possess easily detectable traces or behaviors. Such characteristics can betray the existence of the file system itself or the user's capability of running a plausibly deniable system.

Anderson *et al.*[13] were the first to propose a steganographic file system and described two possible approaches. The first approach consists of a set of cover files filled with random information, of which a subset is combined with hidden files using an additive secret sharing scheme. The exact subset of the cover files that must be XOR'd together to reveal the hidden data is determined by a user's passphrase. This approach is more in line with classical steganographic systems that rely on embedding hidden information in the lower order bits of large files such as audio and video. While this approach is conceptually simple, the presence of unexplained random cover files is suspicious and a relatively large number of cover files is required to provide computational security.

The second construction hides data within the unallocated space of another file system. Although the proposal lacked an implementation of the two ideas, most deniable storage systems follow the second approach.

McDonald and Kuhn implemented Anderson *et al.*'s second scheme as a Linux file system based on `ext2` known as StegFS[4]. StegFS uses a block allocation table to map encrypted data to unallocated blocks with the additional capability of nesting hidden volumes so that the user can reveal some hidden data in the hope of satiating an adversary. This system does not adequately address the issue of a multiple snapshot attack.

Pang *et al.*[14] implemented their variant of StegFS that improved reliability by removing the risk of data loss in the hidden file system when the open file system writes data. However, this version contains a bitmap which exposes the existence and maximum size of the hidden volume.

Mnemosyne[15] replaces StegFS's simple replication technique with Rabin's Information Dispersal Algorithm[16] in order to provide greater durability and improve write amplification across nodes. However, since it is designed as a peer-to-peer system, it does not hold up to our adversary model.

The on-the-fly-encryption (OTFE) system TrueCrypt[6] also provides the capability of running a hidden file system within the free space of an ordinary encrypted volume. Its approach is similar to StegFS in that each nested file system has a single key, which grants access to the hidden data. Since such approaches coexist with the public operating system, challenges arise concerning information leakage through programs that access the hidden volume[7]. Additionally they do not defend against multiple snapshot attacks.

Datalair[11] and HIVE[10] combine a hidden volume with ORAM[8,9] techniques to obscure the volume's existence and disguise access patterns. Accesses to hidden data are disguised among random accesses to a public volume. Theoretically, this prevents an adversary from successfully carrying out a multiple snapshot attack. In practice, ORAM and similar techniques incur significant performance penalties that severely impact the usability of the hidden and public volumes. In the case of HIVE, throughput for both public and hidden sequential operations is slowed to approximately 1 MB/s[11]. Random disk write patterns and unexplained slow performance compared to the raw disk can possibly be viewed as suspicious.

Recently, Chen *et al.*[17] published PD-DM, a device mapper based approach aimed at addressing the poor performance of ORAM dependent systems[10,11]. Although it significantly improves read performance, write performance still suffers, and it presents the same distinctive performance characteristics that would betray the existence of a hidden volume.

Mobiflage[18], a deniable storage system for mobile devices, maintains a partition of the disk containing random data within which hidden data can possibly be stored. It relies on the ambiguity of whether or not hidden data are present to provide deniability while ignoring that the presence of a disk partition containing unexplained random information is suspicious.

DEFY[19] is a log structured deniable file system designed for host controlled flash devices and is based on WhisperYAFFS[20]. DEFY does not adequately protect against hidden data overwrite unless hidden volumes are constantly mounted and is limited to use on a type of raw flash device called memory technology devices (MTDs).

Zuck *et al.*[21] proposed the ever-changing disk (ECD)[21], a firmware design that splits a device into hidden and public volumes where hidden data are written alongside pseudorandom data in a log structured manner. Although the design makes significant progress towards solving the problem of hidden data overwrite and mitigating multiple snapshot attacks, the lack of deniability for the exposed partitioning scheme and proposed custom firmware are a vulnerability.

None of the previously described systems hide a user's capability of running a plausibly deniable system, prevent information leakage, or address malicious software installed by an adversary. One proposed system, a previous version of Artifice[22], attempts to address these concerns but lacks a well defined threat model, a better description of its information dispersal techniques, does not address operational security, and needs a more thorough evaluation.

3. THE PROBLEM OF PLAUSIBLY DENIABLE STORAGE

Many existing deniable storage systems' faults originate with the assumptions made in their adversary model or in some cases an absence of one. We believe that it is critical to the design and operational considerations of deniable storage to better define and explore the possible threats against such systems. Most often overlooked is the fact that the adversary does not have to prove the existence of a deniable volume to compromise the user. Since the adversary is often assumed to apply coercive tactics to access visible obfuscated data, they could also apply the same technique to compromise a deniable volume. In the case of encrypted data, the obfuscated bytes are readily apparent, but, in the case of a deniable volume, there must be some indication that the user is running such a system to warrant escalating to coercion. As a result, we assume attacks against a deniable storage system do not have to prove the existence of a hidden volume but only convince the adversary that there is a sufficient probability that the user is hiding something. Since the ultimate goal of a hidden volume is to avoid coercive tactics, we must then avoid arousing the adversary's suspicion.

Some examples of previous work assume that the user's ability to utilize a deniable storage system is innocuous if that system is bundled with other widely used software such as an operating system[18,23]. Since plausibly deniable storage is a response to scrutiny of disk encryption systems, it follows that the adversary would also view a deniable storage system with similar, if not more, suspicion than they would an encrypted volume. Without this assumption, even sophisticated multiple snapshot resistant approaches would be vulnerable to a significantly simpler adversary. While this more limited adversary would have to know what driver software or system characteristics to look for, we must also assume that the adversary has knowledge of the deniable system's design or else we risk relying on "security through obscurity", which should be avoided[24]. From this, we propose that a user's ability to run a plausibly deniable storage system cannot be considered normal and as such any secure deniable storage system must take efforts to disguise a user's ability to run said system in addition to any hidden data.

The second flawed assumption is that the adversary will not install malware or take other steps to gain more information about that device than what they can obtain through static snapshots[14,17–19,23]. It has been observed that officials have installed malware on users' devices when they cross a border[25]. We propose that a deniable storage system must assume that malware can be installed on the user's device and that it should be capable of defending against it. Since malware can potentially provide a significant amount of information to the adversary about the user's activities, we must assume that an adversary that can install malware on the user's device must be capable of carrying out continuous traffic analysis to detect a hidden volume.

3.1 Adversary model

We consider our adversary to likely be an agency or organization that wishes to tightly control the flow of information inside and across their domain's borders. We assume that a likely and typical use case for a deniable storage system entails the adversary gaining unfettered access to a device for a short period of time in the presence of the user, such as officials inspecting a device at a border crossing. This does not preclude the possibility of the adversary gaining access to the user's device and installing malicious software or compromising the device's firmware without the user's knowledge in what is commonly known as an "evil maid" attack[26].

From these points, we propose the following adversary model and operational assumptions for a deniable storage system. We must assume that the adversary has knowledge of deniable storage systems, their designs, and capabilities. We consider any part of the user's machine that interfaces with the disks or other storage devices to be part of the attack surface. Unique among deniable storage is that we must consider that the user is also open to attacks by our adversary, not just the device. As a result, we have to include the user in the attack surface.

We assume the adversary can confiscate the user's device and perform any *static* forensic analysis that they deem warranted. While accessing the device, the adversary has the ability to install and run applications, modify files, manipulate disk sectors, and make copies of the disk. This includes the possibility of a multiple snapshot attack where the adversary has been able to analyze the device at multiple points in time. The adversary can also install operating system level malware (rootkit) or malicious application on the user's device for the purpose of gathering more information about the user's activities and device characteristics. Similarly, the adversary can monitor the user's interactions with public network infrastructure for suspicious behavior. That said, they cannot continuously monitor the user's actions at all times. If an adversary is able to continuously monitor the user, then there is no safe time to access a hidden volume. Should the adversary discover a suspicious aspect of the user's device such as an undisclosed partition, hidden information, or suspicious software, they may coerce the user to reveal a password, encryption key, or other sensitive information possibly. This coercion can include methods such as the threat of legal penalties[3] or the use of a rubber-hose attack[1]. As the adversary can easily verify any access credentials provided by the user under coercion, we assume that coercion is a very effective means of compromising a hidden volume.

While we assume that our adversary is relatively powerful, we must also consider that even a sophisticated intelligence agency will possess considerable but ultimately finite resources to carry out an attack. From this, we can assume the adversary would have a procedure for escalating the rigor and corresponding resource cost of their tactics. For example, they would move from relatively simple techniques such as disk capacity checks and looking for suspicious software to more involved techniques such as forensic analysis and snapshot attacks. The eventual result of this escalation is coercive tactics and/or detaining the user. While this model is useful in determining requirements for a deniable storage system, we cannot predict how suspicious the adversary will initially be of a given user and at which level of rigor the adversary will start examining their device.

3.2 Deniability of random information

From the beginning, deniable storage systems have relied on hiding obfuscated information in and amongst other pseudorandom data existing on a user's device. A common problem that all these systems face is providing a deniable reason for the existence of pseudorandom information on the disk.

If we hide information when using a whole disk encryption system, then at first glance one will not be able to tell the difference between the public file system, unallocated space, and the hidden data. As we assume our adversary treats encrypted information as suspicious, it would also mean that all pseudorandom data on the disk is suspect until the adversary further examines it. When the adversary gains access to the encryption key, it will reveal the free space, which should still be decryptable unless the key has been changed since the data were deleted. In this case, our hidden information cannot be decrypted with this key and will stand out. Ideally, we need some way to render free space pseudorandom without providing a means for the adversary to recover and decrypt blocks. To do this, we can use a secure deletion utility to encrypt data in the free space of a file system and throw away the key, providing a deniable reason for the presence of undecryptable pseudorandom information.

3.3 Multiple levels of deniability

Many previous deniable storage systems support multiple levels of hidden information where the lower is the level, the more sensitive is the information[4,14,19]. The intended purpose is so that when subject to coercion the user could reveal less sensitive levels of the deniable system while keeping others secret. This is similar to running a whole disk encryption system alongside a deniable system where the user would surrender the encryption key to an adversary while not revealing the presence of the deniable system. The multi-level approach assumes the adversary will deduce the existence of a hidden volume. While such an approach may satiate an ignorant or less determined adversary, if they possess knowledge of the storage system's capabilities, then we have to assume they will press the user further for access to further hidden levels.

As we must assume that our adversary knows of a deniable system's design and capabilities, it becomes clear that multiple levels of deniability should not be used as a primary defensive measure. That is not to say that the capability is not worthwhile but that it should be relied on as a fallback if other methods of protecting the user have failed.

4. DESIGN REQUIREMENTS

Considering our assumed adversary as previously described, we can derive the following series of design requirements for a deniable storage system.

1. **Render hidden information indistinguishable from deniable information.** As most deniable storage devices rely on hiding information within the free space of some other volume, it is essential to make such information indistinguishable from the rest of the free space. A deniable system that separates public and hidden volumes with a unique partitioning scheme cannot fulfill this requirement as it is possible for the adversary to compare the usable and advertised capacities of the device and investigate any discrepancies.
2. **Prevent information leakage.** Since deniable storage systems coexist with the public operating system, challenges arise concerning information leakage through programs that access the hidden volume. In the case of TrueCrypt, Czeskis *et al.*[7] demonstrated that the system was plagued by information leakage through both the features of the Windows operating system and applications such as Microsoft Word. It was also made apparent by Troncoso *et al.*[5] that, should an adversary install malware on a device to continuously leak disk traffic information, then it is possible to reveal the existence and location of hidden files.
3. **Mitigate the effects of overwrites by the public file system.** Under normal use the system does not know about the data hidden in unallocated space, so any writes have some probability of overwriting clandestine data. A deniable storage system hiding within unallocated space must therefore provide some reasonable protection from overwrite.
4. **Disguise changes to the free space of the file system.** Foremost among the concerns for deniable storage systems is the multiple snapshot attack in which the adversary is able to capture images of the disk and infer the existence of a hidden volume through analysis of the changes. Most approaches attempt to disguise accesses to the hidden volume with either many random accesses to the disk[10,11,14,17] or with hiding hidden data accesses within other random information[21].
5. **Hide the user's ability to run a deniable storage system.** As discussed above, it is unlikely for the average user to keep a deniable volume driver on their devices and thus possession of such software would be considered suspicious. The presence of a driver implies that the user's device contains a hidden volume. Detecting the driver software is perhaps the least computationally intensive manner for the detection of a deniable system, as it only involves inspection of the storage software stack, device firmware, behavioral characteristics, or partitioning scheme. In the case of some systems, there is a significant performance impact for both the hidden and public volumes such that it would be simple to infer the existence of a deniable system[10,11,17]. While it is possible to hide such software through the use of a rootkit or other malware, we cannot rely on this technique as it is "security through obscurity".

There is no previous system that satisfies all of these requirements. In particular, none hide a user's capability of running a plausibly deniable system or address malicious software installed by an adversary.

Efforts to satisfy these requirements inevitably result in design compromises. For instance, to protect against

Table 1. Plausibly deniable systems

System	Overwrite protection	Indistinguishability	Information leakage resistance	Deniable changes	Deniable software
Veracrypt[27]	✓	-	-	-	-
StegFS (McDonald)[4]	Probabilistic	✓	-	-	-
StegFS (Pang)[14]	✓	-	-	Dummy accesses	-
Hive[10]	✓	-	-	ORAM	-
Datalair[11]	✓	-	-	ORAM	-
DEFY[19]	✓	-	-	-	-
Mobiflage[18]	✓	-	-	-	-
Mobipluto[23]	✓	-	-	-	-
Ever Changing Disk[21]	Probabilistic	✓	-	User behavior	-
PD-DM[17]	✓	-	-	ORAM	-
Artifice	Probabilistic	✓	✓	Operational security	✓

hidden data overwrite or to disguise access patterns, performance is inevitably affected by IO amplification from reading, writing, or analyzing more blocks per operation.

The overarching goal of any deniable storage system is not only to provide confidentiality but to also allow the user to deny the data's existence. There will inevitably be anomalies on the disk that the adversary will investigate in an attempt to find a clue that points to the existence of a hidden volume. These requirements are meant to minimize the number of anomalies visible to the adversary while also equipping the user to provide deniable reasons for those that remain.

5. DESIGN

To address the previously discussed design requirements, we designed *Artifice*, a plausibly deniable virtual block device built on the Linux device mapper kernel interface (shown in Figure 1). Artifice obfuscates data and provides protection from accidental overwrite using an IDA such as Shamir Secret Sharing[12] to generate a set of pseudorandom shares or *carrier blocks* from a user's data blocks. These carrier blocks provide *combinatorial security* where an adversary must select the correct blocks out of the free space to reconstruct a data block. Adding redundant carrier blocks enables Artifice to repair itself when it is inevitably damaged by the public file system. This IDA-based approach and flexible block allocation allows the user to configure Artifice for use with a variety of public file systems and mitigate the effectiveness of a multiple snapshot attack.

Unlike previous approaches that require driver software to be installed on the user's device, a user accesses Artifice by booting a separate live Linux installation on a USB drive containing the Artifice driver. Isolating the driver from the public operating system prevents information leakage and protects the Artifice volume from most malware. Separating the hidden data from the driver software prevents the adversary from inferring the existence of the data from the existence of the Artifice software. It is also important to note that the user does not need to possess a copy of the bootable USB drive at all times, and it would be advantageous for them to not be carrying it on their person when under the scrutiny of an adversary.

5.1. Information dispersal algorithms

Artifice addresses the problems of obfuscation and hidden data overwrite with a single step through the use of an IDA. Artifice is designed to utilize Shamir Secret Sharing[12] or systematic erasure codes combined with an all or nothing transform[28] to generate carrier blocks (shown in Figure 2). These algorithms provide an (n, k) scheme where at least k carrier blocks out of a set of n are needed to reconstruct the original data and no data are recovered if fewer than k blocks are available.

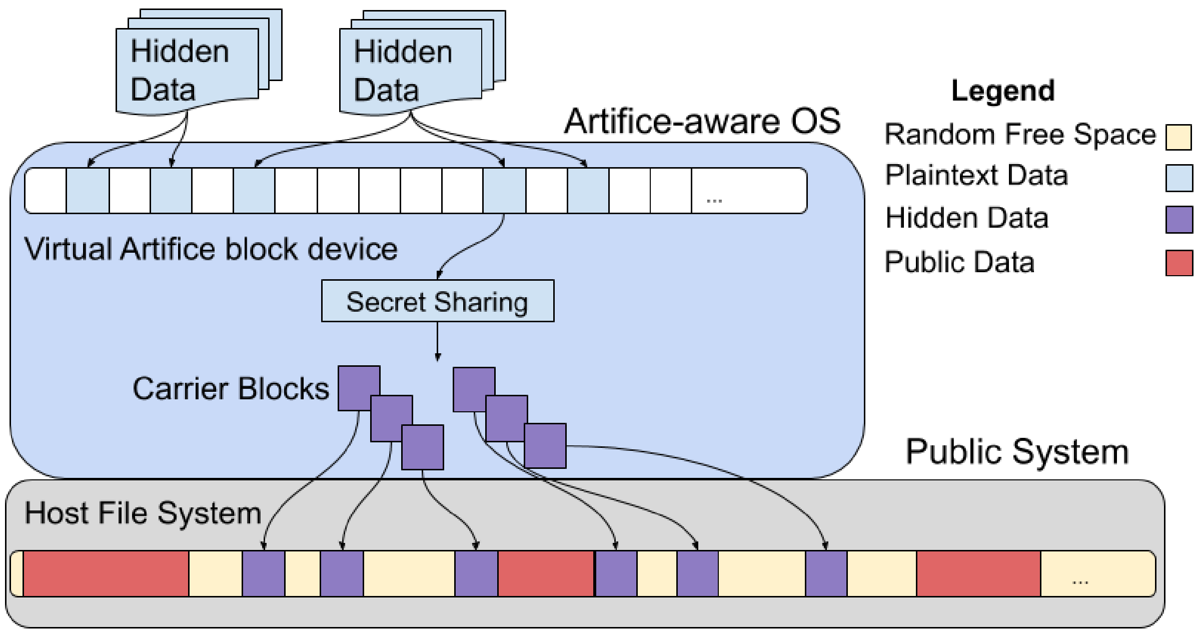


Figure 1. System overview of Artifice. The Artifice kernel module resides in a separate operating system contained on removable media. The public system includes the public file system that Artifice hides in and the public OS. Free space in the public file system should be filled with pseudorandom blocks.

Carrier block overwrite will occur through operations performed by the public system. Artifice treats overwrites by the public file system as erasures or lost shares. Reconstructing the original data block through an IDA provides Artifice with the redundancy needed to tolerate accidental overwrites and the ability for the system to repair itself through reconstructing and remapping lost carrier blocks.

The first IDA that Artifice can use is Shamir Secret Sharing[12]. This algorithm secret splits a single data block into a set of n carrier blocks of which $k \leq n$ are required to reconstruct the original data block. As long as $k < n$, the scheme can tolerate at least $n - k$ carrier block overwrites per set. In addition to obfuscation and redundancy, Shamir's scheme also provides a guarantee of information theoretic security. This prevents the adversary from gleaning information about the data block without picking a full set k of the correct carrier blocks out of the public file system's unallocated space.

The downside to utilizing Shamir's secret sharing scheme is that it does not provide any advantage in space efficiency for a given level of redundancy over a simple encryption and replication scheme. This is not a significant concern for Artifice as Shamir's scheme provides us with a simple design and a strong security guarantee when used in conjunction with encryption.

To address the drawbacks of Shamir's scheme, Artifice can also utilize a significantly more efficient IDA called AONT-RS[28]. This algorithm combines an all or nothing transform[29] with a systematic Reed–Solomon erasure code[30] to provide a threshold scheme similar to Shamir's approach but more space efficient. With AONT-RS, the message M is encrypted with a randomly generated key K to generate the ciphertext C. AONT-RS appends a known integrity checking "canary" value to the data prior to encryption. With Artifice, we modify this and instead store a hash of the M in Artifice's metadata. The encrypted data are then hashed to produce h, which is XOR'd with K to produce a difference d. The difference d is appended to the encrypted data, which are then split into a set of k blocks, with the number k being our reconstruction threshold. To generate the set of redundant blocks m, we encode our encrypted data using Reed–Solomon codes to produce m parity blocks.

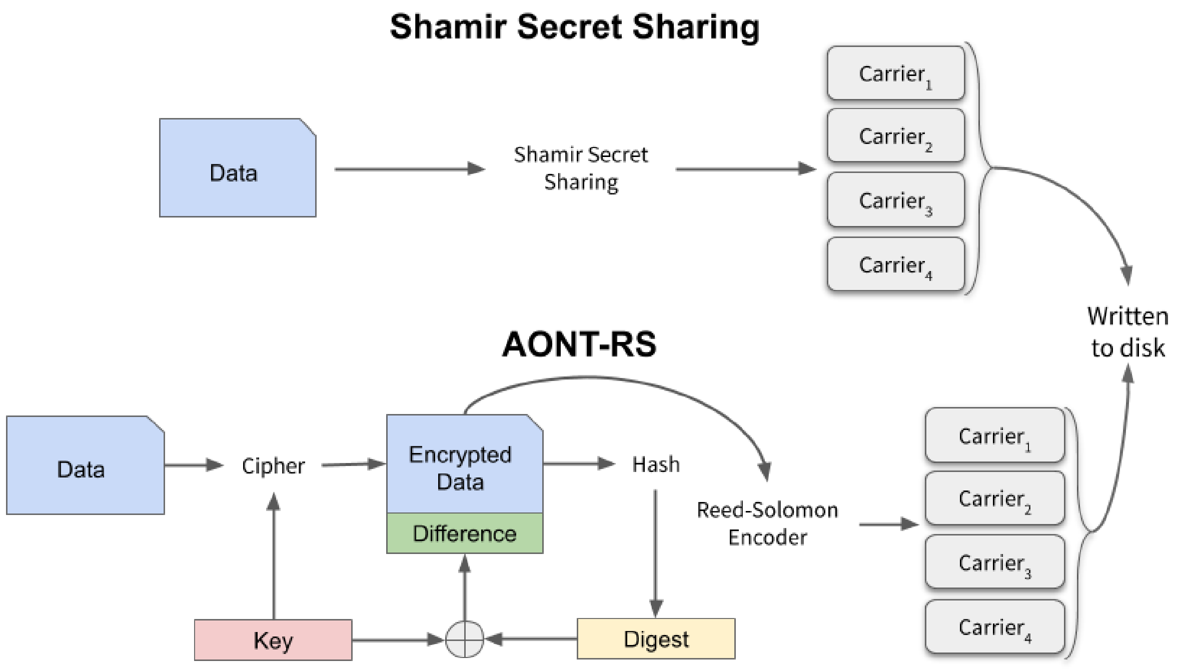


Figure 2. The information dispersal algorithms available for use with Artifice.

We then have the set of n carrier blocks where $n = m + k$.

To reconstruct the data, Artifice decodes set of carrier blocks using Reed–Solomon to resolve any erasures and recover the ciphertext. C is then hashed to reproduce h, and the key is recovered from XOR'ing the hash with the appended value d. The data can then be decrypted and the integrity of the result verified using our stored hash. The AONT-RS implementation used with Artifice uses material derived from the user's passphrase as an initialization vector during the encryption step.

Instead of each share being the same size as the original data block, they can be a fraction of the size of the original secret similar to Rabin's IDA[16]. This combined with the efficiency of Reed–Solomon allows significantly lower space consumption versus a similarly resilient secret sharing scheme. For example, to store i blocks of s bytes with a threshold of k in our Reed–Solomon/entropy approach, we must have $m \geq k - e$ carrier blocks and therefore store $(m \times s \times i)/(k-e)$ bytes. To store the same number of blocks with our secret sharing scheme, we must store $i \times s \times (m+k)$ bytes. In AONT-RS, we only have to store $(i \times s \times (m+k))/k$ bytes. However, unlike secret sharing, it only provides a computational guarantee as opposed to an information theoretic guarantee.

There is of course a trade off among resilience, security, and performance, but using an IDA allows for a measure of tunability. The larger is the threshold k required to rebuild the data, the more secure is the system as the number of possible block combinations has increased. Conversely, a smaller k in proportion to n provides more data resiliency in the face of overwrites as we have more redundant blocks that Artifice can afford to lose. One can also increase the number of carrier blocks to provide more resiliency at the cost of space efficiency and performance due to increased write amplification and extra CPU intensive operations. Conversely, decreasing the number of carrier blocks can improve performance.

5.2. Combinatorial security

The use of an IDA provides Artifice with combinatorial security on top of more traditional computational security provided by encryption. Without knowledge of which carrier blocks correspond to what data blocks

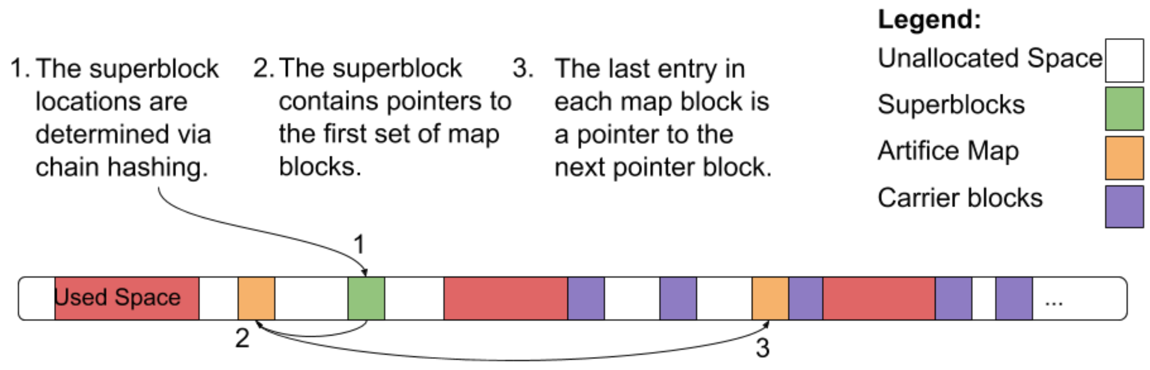

1. The superblock locations are determined via chain hashing.
2. The superblock contains pointers to the first set of map blocks.
3. The last entry in each map block is a pointer to the next pointer block.
Legend:
Unallocated Space
Superblocks
Artifice Map
Carrier blocks
Used Space
1
2
3
...

Figure 3. The process of locating Artifice superblocks through chain hashing a user's passphrase.

and with carrier blocks indistinguishable from other free space on the disk, an adversary must attempt to reconstruct every combination of possible carrier blocks.

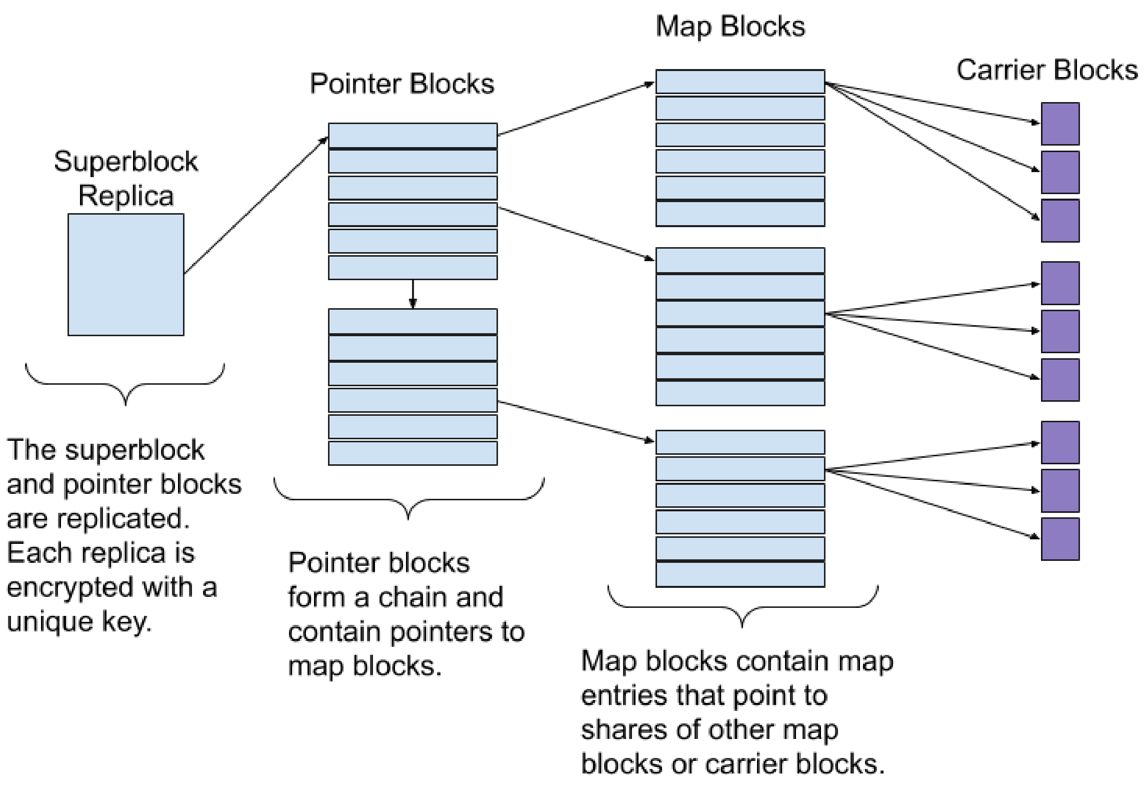
If we assume that the adversary cannot determine which unallocated blocks contain hidden data, then the time needed for a brute force attempt to reconstruct the Artifice volume is on the order of $O(\frac{n!}{k!(n-k)!})$, where N is the total number of unallocated blocks on the disk. In this case, the threshold k can be considered constant or limited to a small range of realistic values, so the computational complexity can be simplified to $O(N^k)$. While polynomial time does not necessarily provide a strong security guarantee on its own, the number of blocks N can be quite large. In the case of a 1 TB disk with 512 GB of free space, there are 2^{27} 4 KB unallocated blocks. If we assume each data block is divided into a set of 12 carrier blocks with a reconstruction threshold of 8, then there are $(2^{27})^8$ or 2^{216} possible combinations for reconstructing each data block.

Even if an adversary is able to determine which blocks contain hidden data, a brute force attack is still infeasible. Using the same secret sharing parameters as the previous example and assuming our Artifice volume has footprint of 8 GB or 2^{21} 4 KB carrier blocks, then there are still $(2^{21})^8$ or 2^{168} possible block combinations for the adversary to attempt. Each data block can be reconstructed through $\binom{12}{8} = 495$ different combinations. With this and the approximate size of the volume, we can approximate that, out of the 2^{168} block combinations, on the order of 2^{27} combinations will yield a valid data block.

5.3. Artifice map

Artifice reads data by identifying the carrier blocks and entropy blocks associated with the logical block address through a metadata structure called the Artifice Map (shown in Figure 4). The Artifice Map is a multi-level tree that stores mappings from logical data blocks to physical carrier block locations. The map is made repairable in the face of overwrites by our IDA scheme and is stored alongside the carrier blocks in unallocated space.

A hash of a user specified passphrase is used to determine the location of a superblock which provides general information about the metadata structures and the locations of the Artifice Map carrier blocks (Figure 3). The superblock is replicated to protect against overwrite, and each replica is encrypted with a different unique key derived from the passphrase. The possible block offset of each replica is derived from a hash of the previous replica's possible location. If a specified location is in use by a valid public block, then the next possible location is used for that replica. A possible location is confirmed to contain a superblock replica if the decrypted replica starts with a known value. If a possible location for a superblock is found to not begin with the expected value when mounting an Artifice volume, then the driver moves to the next location and repeats the process until a replica is found. The number of these replicas written to the disk is defined by the user. Artifice should be configured to generate more replicas of the superblock than shares of the data blocks to provide a better


Map Blocks
Carrier Blocks
Pointer Blocks
Superblock Replica
The superblock and pointer blocks are replicated. Each replica is encrypted with a unique key.
Pointer blocks form a chain and contain pointers to map blocks.
Map blocks contain map entries that point to shares of other map blocks or carrier blocks.

Example Map Entry:

Artifice block number.	Carrier block hashes and pointers	Data block hash
0	<10, 0xBE>, <90, 0xA1>, <17, 0xD0>	0xCAFE

Figure 4. The design of the Artifice Map.

probability of survival. Should a user need to revoke the passphrase, the superblocks are re-encrypted with a new passphrase and moved to new locations determined by the new passphrase.

Each entry in the map contains a set of carrier block pointers, checksums of each carrier block, and a hash of the original data block that is used to verify the reconstruction succeeded. In the case of an encoding scheme that requires an external entropy source, identifying information about that entropy source is also included. These entries are arranged into *map blocks*.

To support information dispersal for the map blocks, a multi-level approach is required. Additional levels of map blocks are used to track the shares of the next lowest level of map blocks. This technique reduces the size of the top level of map blocks. The location of the top level of map blocks are referenced by a set of *pointer blocks*. Both this top level of map blocks and pointer blocks are replicated and encrypted in a similar manner to the superblock. Information about the pointer blocks is stored in the superblock. When Artifice is running, map and pointer blocks are reconstructed and a working set is cached in memory. As the map is modified by new writes, it is periodically flushed to the disk.

5.4. Block operations and self-repair

Disk operations in Artifice occur on block IO requests similarly to other device mappers such as dm-crypt[31]. However, instead of encrypting data blocks, the blocks are split into carrier blocks on writes and reconstructed on reads. Artifice differs from most device mappers as it possesses the ability to self repair.

To determine what space on the disk is unallocated, Artifice identifies the type of the public file system and parses its metadata to return a list of unallocated block addresses. When Artifice is initialized, the user provides their passphrase, which is used to determine the current locations of the superblock replicas. The superblock then provides enough information to reconstruct and load the Artifice Map into memory.

This list of block addresses and the reconstructed Artifice Map are used to construct an allocation bitvector, at which point Artifice can carry out self repair and normal block device operations. As it is only a block device, the user would then format the Artifice device with a file system of their choice and mount it as they would any other disk. Additional data integrity protections can be provided by this file system.

Artifice differs from most device mappers in its self repair processes. Self repair occurs whenever Artifice is mounted or when prompted by the user and starts with a scan of the Artifice Map and the list of unallocated blocks from the public file system. Whether a carrier block has been overwritten is determined through the carrier block checksum stored in the map entry. If a carrier block has been overwritten, it is considered an erasure in our encoding scheme. The missing carrier blocks are reconstructed and remapped to new locations on the disk. Should additional problems arise, the checksum of the original data block is stored in the map to verify that the data block was rebuilt successfully.

The self repair process is best executed whenever Artifice is initialized and can run in the background while other disk accesses are in progress. Any additional error correction, crash recovery, or integrity checks are left to the file system that Artifice is formatted with.

Lastly, Artifice provides a trivial method for self destruction. As the deletion of carrier blocks is a normal consequence of public volume writes, one can simply discard the passphrase. Without the passphrase, information retrieval is computationally infeasible and normal public operations will overwrite the hidden data over time. For additional security, Artifice can also delete the superblock replicas to erase any chance of finding the hidden volume.

5.5. Public file system considerations

To effectively combat hidden data overwrite and the multiple snapshot attack, Artifice must be able to tailor its behavior to the file system that it is hiding in. In the case of data overwrite, Artifice should react to the block allocation and write behaviors of the public file system. This takes the form of identifying "busy" locations in the public file system, such as journaling regions, and avoiding them, allowing Artifice to place data in free blocks that are the least likely to be overwritten in the near future. In the case of log structured file systems[32], it is best to place only one carrier block of out a set in each segment.Thus, the impact of garbage collection on any individual segment is minimized. Lastly, a user must avoid activities such as large data writes and SSD TRIM operations. These pose a significant risk to the carrier blocks regardless of efforts to protect them as both can overwrite or remap a large number of blocks. When multiple Artifice volumes are present within the same free space, overwrites can also come from these other instances. The end result of overwritten blocks is the same, but the user must be aware that storing multiple hidden volumes increases the risk of data loss.

Additionally, there is the problem of a deniable reason for large amounts of pseudorandom information in the free space of a file system. The naive method is to fill the free space of a disk with pseudorandom bits prior to initializing a hidden volume[6]. The drawback of this approach is that filling the unallocated portions of the disk with unexplained pseudorandom information could be considered suspicious. Even though using Artifice in conjunction with a disk encryption system would make the free space appear random at first glance, we must assume that the adversary would be in possession of the key to the publicly visible encrypted data. As a result, such data would be readable and any obfuscated carrier blocks could possibly be discovered. To avoid this, we need to render the free space undecryptable. This can be achieved through a secure deletion utility that deletes data by encrypting it and throwing away the key. If a user were to use such a program to securely wipe a drive prior to using it with a hidden volume, then there would be a pool of random blocks for Artifice to hide in. The drawback is that, as the user writes and deletes public data, the pool of deniably random blocks would shrink. To solve this, a public file system that performs secure deletion through encryption by default would be the ideal environment for Artifice as it replenishes the pool of deniably random blocks whenever the user deletes public data.

6. OPERATIONAL SECURITY

Operational security is an often overlooked aspect of deniable systems. Exploring the procedures and operational circumstances surrounding the deployment of such a storage system is critical and very likely to be the deciding factor in the actual security of the system. In the case of Artifice, we must consider the deniability of the driver software, the security of the passphrase, physical security of the device, and the measures that should be taken if any of the components necessary to use Artifice are compromised. These measures are summarized in Table 2.

As mentioned previously, Artifice aims to provide deniability for the storage system's driver, as the presence of a relatively uncommon and suspicious program on the user's device would imply the existence of a deniable volume. Keeping the volume and software on separate devices assists in this but is still heavily reliant on the user's practices. We assume that the deniability of the software is tied to its probability of being present on an average user's device. The more common and innocuous the software is, the less suspicious it is to our adversary. Due to the many drawbacks inherent to deniable storage systems, we can assume that a user will not have it installed unless they intend to maintain a hidden volume. However, if a deniable storage system were to be included in some other common software package, such as the Linux kernel, then the user would have a deniable reason for the driver's presence on their device.

In an ideal scenario, the user would ensure that they do not possess a copy of Artifice on a live disk when the adversary is most likely to inspect their device. This assumes the user has discarded their original live disk and made arrangements to obtain a copy once the present danger has passed to access or repair the volume. To carry out this more secure procedure, the user in possession of the hidden volume may need to coordinate with multiple other individuals. When this is not practical the user could fall back on classic steganographic techniques such as hiding data in the lower order bits of images to hide the software on the live disk. This approach would require less overhead as the driver is significantly smaller than the hidden volume. Should the user carry the live disk on their person without additional measures to hide the software and it falls into the hands of the adversary, we must be concerned about the adversary escalating efforts to monitor the user's actions and we must assume the adversary knows that the user is possibly in possession of a hidden volume.

Artifice does not persistently store any material pertaining to the passphrase on a device. To further improve security, multiple factors can be utilized alongside the passphrase to locate and secure an Artifice volume. If a passphrase is compromised by an adversary, Artifice provides an easy means for revoking and replacing a set of credentials. When a new passphrase is chosen, Artifice will determine new locations for the superblock and pointer blocks. These will be remapped to those new locations and encrypted using the new passphrase. The old superblock replicas and pointer blocks will be overwritten with random information. The adversary gaining access to the live disk is irrelevant as the adversary could at this point know the user has an Artifice

Table 2. Possible scenarios and remedies for the compromise of different Artifice components.

✓: The item is still in the user's possession or not compromised by the adversary.
×: The item has been lost or compromised.
-: Irrelevant to the scenario.

Passphrase	Device	Live Disk	Loss of Security	Description	Remedy
×	✓	-	no	Passphrase is compromised by the adversary.	The existing passphrase should be revoked and replaced with new credentials.
✓	×	✓	no	Device containing the volume is compromised.	Change the passphrase, deniably scramble information on the disk, or destroy the device.
✓	✓	×	no	Disk containing the driver is compromised.	The user should be concerned about the adversary escalating efforts to monitor the user's actions.
✓	✓	✓	no	Surveilled by the adversary while accessing Artifice.	The user should be concerned about the adversary escalating efforts to monitor the user's actions. Having multiple volumes could help.
×	×	×	yes	All components are compromised.	There is no possible remedy that will protect the user or the hidden information.

volume. If the user is able to change the passphrase prior to the adversary gaining access to the storage volume, then the adversary will be unable to find Artifice using the old passphrase.

Should the adversary gain physical or remote access to the user's device, we must assume that they have installed malware to monitor the user's actions and taken a snapshot of the disk. Isolating Artifice to a separate operating system helps protect against leakage through OS-level malware. Artifice's model cannot help protect against hardware or firmware level malware, and, if such tampering is assumed to be present, then the safest option is to replace the device. If this is not possible, then the user could scramble information on the disk through an operation such as disk defragmentation or system cleanup utilities and change the passphrase. Ultimately, it is advisable for the user to keep their device with them at all times to protect against the adversary gaining access without their knowledge.

If all the components of the system are compromised, there is little that can be done to maintain security. However, if the user possessed multiple hidden volumes and only the passphrases for a subset of these are compromised, then they may be able to satiate the adversary by revealing one of the volumes. It is unknown if this would be a viable strategy because we assume the adversary would be aware of this capability and may continue pressing the user for more information.

7. DESIGN ISSUES FOR SOLID STATE DRIVES

Solid state drives (SSD) create a set of different issues for Artifice versus traditional hard drives. The logical block store that the flash translation layer (FTL) presents to the operating system allows the SSD to relocate physical pages so that garbage collection can reclaim pages invalidated by more recent writes independently of the operating system. It is necessary for the SSD to create free flash blocks (encompassing a moderate but fixed number of pages) that can be erased and made available to future writes. Erased blocks are usually not available via the logical interface as they are not mapped into the logical address space. The FTL will mark Artifice blocks as written which creates an opening for detecting Artifice through forensic analysis. Alternatively, the FTL may unknowingly erase hidden data as part of opaque and non-standardized garbage collection operations if it is unaware of Artifice's presence.

This layer of abstraction presents a hurdle for deniable storage systems. Most that seek to address these challenges either work on raw flash devices[19] or are intended to operate as drive firmware[21,33,34]. Since custom firmware would be suspicious and raw flash devices are still relatively uncommon, Artifice must attempt to address these challenges through other means.

7.1. TRIM

Most modern file systems support the TRIM function, which notifies an SSD that certain blocks are no longer in use by the host, and thus need not be copied to new locations during garbage collection. Ideally, for the public file system, the hidden data would be TRIMmed, therefore marked as unallocated by the SSD, and would treat garbage collection operations as another form of accidental overwrite. However, only one kind of TRIM (Non-deterministic TRIM) allows the possibility of accesses to the original data after a block has been subject to TRIM. When reading from TRIMmed blocks, the SSD couldreturn either the original data or some other information if the block has been subjected to garbage collection.

The other two types of TRIM are far more damaging for a deniable storage system: deterministic read after TRIM (DRAT) and Deterministic Read Zero after TRIM (RZAT). Both will return some consistent pre-defined value for any logical block address that has been TRIMmed. In this case, it would be necessary for a deniable storage system to leave all of its blocks listed as allocated on the SSD and therefore vulnerable to forensic analysis. Additionally, deterministic TRIM will cause most, if not all, free space on the device to appear uniform, eliminating the ability for a deniable storage system to hide within pseudorandom free space.

The challenge posed by TRIM is somewhat mitigated by the fact that most operating systems utilize *periodic TRIM*, where the operating system will periodically send a TRIM command for all blocks deleted after the previous TRIM operation[35,36]. This is viewed as preferable to *continuous TRIM* where a TRIM command is sent to the disk each time a file is deleted. The common use of periodic TRIM allows for a small region of accessible unTRIMmed free space to exist on an SSD between TRIM invocations. The size and lifespan of this region would be dependent on the periodic TRIM settings and how much data the user has deleted since the last TRIM operation. If this region is of sufficient size, then it would be possible to deniably write data to this region.

There is a more reliable method of mitigating the effects of TRIM: it is common to disable the TRIM command if using a drive encryption system as it could leak the locations of the unallocated blocks and reveal the possible size of stored data[6,18,37]. In the case of a deniable storage system, disabling TRIM is an ideal choice as we need not worry about hiding data in blocks that would be altered by TRIM - effectively causing the SSD to behave as a mechanical disk from the perspective of Artifice and the public file system.

7.2. Opportunities in host controlled SSDs and zoned namespaces

Parallel to the development of FTL based SSDs, there has been work on FTL-less flash devices such as open channel SSDs. A related recent development has been the move towards the adoption of Zoned Namespaces[38]. This new approach breaks an SSD up into a series of sequentially written and host controlled "zones" that are written sequentially similar to a log structured file system. Zoned block device support is already included in the Linux kernel through software such as dm-zoned and F2FS[39,40]. Zones behave similarly to segments in a log structured file system or erase blocks on a flash device. Artifice could extend its block allocation functionality to support hiding data in zones already containing pseudorandom or encrypted information. The hidden blocks could be claimed to be deleted data that have not yet been garbage collected by the file system or dm-zoned. As virtual block devices in Linux can layer atop one another, it is also possible to tune Artifice to leverage specific behaviors of dm-zoned to more effectively hide information in a zoned storage device.

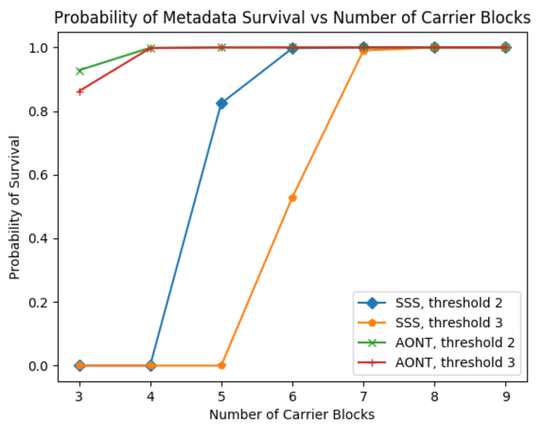
While Zoned Namespaces and other host controlled flash devices present ideal options for bypassing the challenges posed by FTL controlled disks, they have not yet achieved wide adoption and some standards such as Zoned Namespaces have not yet been integrated into commonly available hardware.

8. THE MULTIPLE SNAPSHOT TRADE-OFF

In what is commonly called a multiple snapshot attack, the adversary has been able to view the contents of the user's device at two or more points in time and compare the differences between these snapshots in an attempt to expose hidden volumes. Most recent deniable storage systems place significant emphasis on defending against this sort of attack[10,11,17,19,21]. Efforts to provide a provable guarantee against a multiple snapshot attack inevitably weaken the system against other far simpler attacks as they require constantly running software to disguise accesses. These approaches primarily compromise disguising the user's ability to run a deniable storage system, which we assume is less resource intensive for an adversary to determine. Most previous multiple snapshot resistant systems rely on making accesses to public and hidden volumes indistinguishable from one another through a number of random decoy accesses. If this approach is taken in with Artifice, the adversary would be able to see write patterns that may be abnormal for a given public file system and therefore hard to plausibly deny. Additionally, these approaches do not mitigate information leakage or seek to defend against malware, the later of which could be used to carry out continuous traffic analysis[5], a more powerful attack in which the adversary is capable of viewing the order and timing of individual disk operations as opposed to static snapshots. In light of these trade-offs, Artifice instead prioritizes hiding the user's capability of running a plausibly deniable system, defense against malware, and information leakage resistance and primarily places the onus on the user to avoid multiple snapshot scenarios through operational means.

Artifice's primary defense against a multiple snapshot attack is through operational procedures. Avoiding the scenario of a multiple snapshot attack is the most foolproof way to defeat it. When an adversary gains access to the device, without the user supervising, the user must assume that either a snapshot has been taken or malware has been installed. The easiest and most reliable response is to replace either the whole device or the disk or to deniably scramble the contents of the disk, rendering the previous snapshot meaningless. With any data already contained in the public and hidden volumes copied to the new device, there is then nothing for the adversary to meaningfully compare to the initial snapshot. In the case of a mechanical disk, a defragmentation operation between two snapshots would render the first meaningless and provide a deniable reason for the changes. Only then would hidden data be written to the hidden volume. Although relying on operational security is ideal, it will not always be practical for a user to take such relatively drastic measures.

While operational procedures would be a relatively reliable method of defending against multiple snapshot attacks, alternative means are also possible with Artifice. One possible method is to write data to a portion of the disk where the contents change frequently. This reduces the problem to selecting suitable blocks for storing hidden data at the cost of incurring more overwrites. Artifice will be limited to writing new data only to blocks that have been freed by the public file system after the most recent opportunity for the adversary to take a snapshot. To accomplish this Artifice stores in its metadata an allocation bit vector describing which blocks are in use. When Artifice is next initialized, the current state of the disk would be compared to the previous state. Since these "hot" regions on the disk change frequently, there would be a deniable reason for changes in the free space. There are some serious limitations to this approach. The first is that using a secure delete program or key revocation technique is essential. Otherwise, recently changed pseudorandom blocks that cannot be decrypted by a user's key would be inherently suspicious. The second is that hiding data in frequently changed sections of the disk increases the probability of overwrite and would negatively impact the survivability of the hidden data. Artifice would then need to store larger sets of carrier blocks to provide a reasonable probability of survival. Lastly, for this approach to be feasible, the user must be sure to delete a sufficient amount of data to provide free blocks to write to. It is also unknown whether such attempts at mimicking genuine disk operations hold up to rigorous analysis.



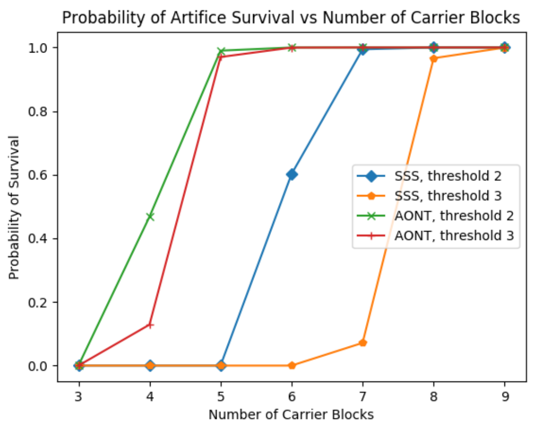


Figure 5. Probability of survival for Artifice metadata in a variety of configurations using both AONT-RS (AONT) and Shamir Secret Sharing (SSS). Probabilities are calculated assuming 512 GB of free space, 5 GB written between repair cycles, 5 GB Artifice volume, and over the course of 365 repair cycles.

9. SURVIVABILITY

Conventional systems are predominantly designed for use with highly reliable devices. Traditional magnetic drives have an uncorrectable error rate on the order of 10^{-13}–10^{-15} [41]. If a block can be read at all, it is extremely unlikely to be incorrect after normal error correction techniques are carried out by the disk. Blocks that are marginal can be remapped by the drive or the file system. Failed blocks are typically protected through error correcting codes or replication.

In contrast, a deniable storage system would have constant destruction of data blocks as a normal behavior. Normal public system operations will overwrite some Artifice carrier blocks. Without a constantly running mechanism to prevent the public file system from overwriting carrier blocks, the survival of the hidden information is probabilistic. Although this may appear as a problematic situation, it is relatively simple to reliably ensure the survival of a small hidden volume hiding in a large area of unallocated space.

Recall from Section 3 that we require k carrier blocks out of a set of n to reconstruct our original data when using secret sharing. By calculating the probability that we will lose no more than $n-k$ blocks, we can determine the probability of survival for an Artifice volume. We assume $n-k$ is the number of redundant shares, s is the logical size of the Artifice volume, $\mathrm{Size}(s,n,k)$ is the number of blocks at risk of being overwritten, p is the probability that a given carrier block is overwritten, and t is the is time in days. We can perform a similar calculation to determine the survival probability for the entirety of an Artifice instance where Size() is instead the effective size of the entire instance when accounting for write amplification.

$$\Pr_{\text{Survival}}(k,n) = \Big(\sum_{i=0}^{n-k} p^i \binom{n}{i} (1-p)^{n-i}\Big)^{\mathrm{Size}(s,n,k)\cdot t}$$

With this function, we can evaluate the probability of survival for a given number of carrier blocks. We assume that the drive in use has 512 GB of unallocated space and an Artifice instance of 5 GB. It is assumed that the user writes 5 GB of data between each time Artifice is initialized to start a repair cycle that rebuilds any overwritten blocks.

Figure 5 shows the survival probability of our example instance over the course of one year assuming a repair

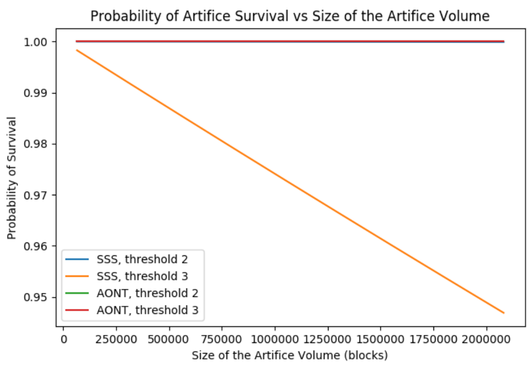


Figure 6. Probability of survival with varying Artifice volume sizes ranging from 256 MB to 4 GB, 5 GB of writes between repair operations, and 512 GB of unallocated space.

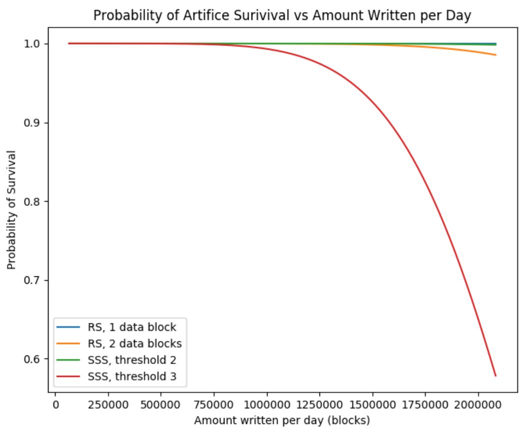


Figure 7. Probability of survival with varying sizes of writes between Artifice invocations from 256 MB to 8 GB, 512 GB of unallocated space, and a 5 GB Artifice volume.

cycle run each day for both the metadata and the entire Artifice instance with a variety of different encoding techniques and numbers of carrier blocks. From these calculations, we can see that there is a number of carrier blocks for each configuration where the probability of survival asymptotically approaches one which depends on the reconstruction threshold k. We can also observe that using AONT-RS can provide better reliability due to improved error correction capabilities and a smaller footprint on the disk. On the other hand, Shamir Secret Sharing would usually require one additional carrier block to provide a similar level of reliability.

We can also model survivability with respect to the size of the Artifice volume, the size of the unallocated space, and the amount written to the public file system between repair operations. For these figures, we assume that each data block corresponds to a set of eight carrier blocks. As shown in Figure 6, the smaller is the Artifice volume, the higher is the probability of survival with overall marginal decreases in reliability even in the case of Shamir Secret Sharing with a threshold of three blocks which lags behind the other configurations. Overall, we can observe a linear relationship between the size of the Artifice volume and reliability. In the case of the amount written to the public volume between repair operations [Figure 7], we can observe an exponential decrease in reliability after approximately 4 GB. Finally, when regarding the amount of free space available to Artifice [Figure 8], we can see that 256 GB of unallocated space provides a promising probability of survival for our Artifice instance.

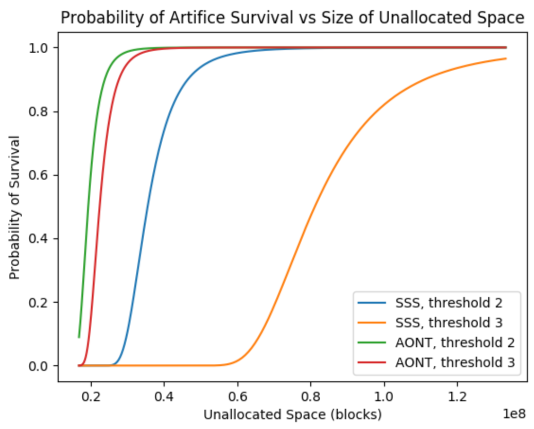


Figure 8. Probability of survival with varying sizes of unallocated spaces from 64 GB to 512 GB, 5 GB of writes between repair operations, and a 5 GB Artifice volume.

The last metric we must consider when evaluating the survivability of an Artifice instance is the write amplification and metadata overhead of our information dispersal scheme. In the case of Shamir Secret Sharing, the metadata overhead is minimal as we must only track the offset of each block and checksums to detect whether a block has been overwritten, although we see significant write amplification as the size of our plaintext data is multiplied by the number of shares. In our AONT-RS scheme, the write amplification is improved such that the size is only amplified by a factor of number of carrier blocks/number of data blocks.

This shows that Artifice can sustain severe damage, as long as the user: (1) maintains a certain percentage of the encapsulating file system free for Artifice to occupy; and (2) regularly mounts Artifice to carry out self repair. It should be noted that these figures do not specifically take into account the probability of overwrite from additional sources such as garbage collection on an SSD utilizing non-deterministic TRIM operations. Although Artifice cannot escape the probabilistic block overwrite behaviors that arise as a result of our stronger adversary model, it is possible with the right configuration with respect to the user's circumstances to effectively nullify the issue.

10. EVALUATION

To demonstrate and test its viability, we implemented Artifice as a loadable kernel module intended to be run from a Linux live flash drive. Artifice uses the device-mapper framework to present the user with a virtual block device. Artifice maps block IO operations from logical data blocks to secret split carrier blocks that are written into the free space of an existing file system. As with most device-mapper targets, Artifice can be layered with other device mappers such as dm-crypt and dm-zoned to modify its behavior. The current implementation of Artifice is easily extensible to support multiple public file system types and currently supports `EXT4` and `FAT32` with planned support for `NTFS` and `APFS`.

For obfuscation and redundancy, Artifice includes a variant of the libgfshare[42] Shamir Secret Sharing library ported for use in the Linux kernel. The current implementation operates on 4 KB logical blocks as it is a common block size for file systems such as `EXT4` and because it is the default Linux page size. Block checksums use a modified version of the cityhash library[43] and the passphrase is hashed with `SHA256`.

10.1 Performance considerations

In general, the performance of a deniable storage system is a low priority. Artifice is *not* a high performance system; its ultimate goal is protecting the user and only requires sufficient throughput to process small amounts

Table 3. Performance comparison of public volume performance with artifice configured with AONT-RS and Shamir Secret Sharing (SSS)

	Public Volume		Artifice AONT		Artifice SSS	
	Throughput	*CPU*	*Throughput*	*CPU*	*Throughput*	*CPU*
Read	795.399 MB/s	59.970%	124.556 MB/s	8.313%	46.946 MB/s	3.000%
	±31.580	±2.419	±3.157	±0.264	±0.986	±0.210
Write	523.758 MB/s	30.731%	82.512 MB/s	17.072%	66.965 MB/s	13.410%
	±12.591	±1.186	±0.812	±0.224	±0.792	±0.356

of information. That said, performance must be sufficiently fast so that Artifice does not become a dangerous hindrance to the user as is the case with some previous systems[10,11,17]. We consider this threshold to be the performance of a small removable storage device such as a USB flash drive. The largest source of overhead is write amplification from writing multiple carrier blocks for each individual data block. To a lesser extent, performance is impacted by the additional processing that secret sharing or Reed–Solomon requires when compared to a scheme that only utilizes encryption. Performance oriented Shamir Secret Sharing and erasure code implementations can be achieved through the use of vector instructions[44] and fast Fourier transforms[45] to accelerate Galois field operations. Despite these optimizations, reading multiple blocks from disparate locations will hinder performance.

Fortunately, the use of magnetic hard drives is rapidly decreasing and with them painfully long seek times. SSDs impose no significant seek penalty and have high read performance. Scattered blocks on an SSD pose less of a performance hindrance.

Contrarily, writing redundant carrier blocks will inevitably impose excess writes and CPU overhead. Traditional buffering techniques can be used to mitigate these delays. Simple methods applied in traditional storage devices, such as contiguous allocation, are not applicable as they introduce correlations that would render Artifice vulnerable to multiple snapshot attacks and an increased risk of accidental overwrite.

It is also important that a deniable storage system does not impact the performance of the public system as is the case with some approaches aimed at tackling the multiple snapshot attack[10,11,17].

Our test machine was equipped with an AMD R7-2700 CPU, 32 GB of RAM, and a 1 TB HP EX920 SSD. To better model an average laptop computer, we ran all benchmarks on a Virtualbox virtual machine with four processor cores, 4GB of RAM, and a virtual disk formatted with FAT32 containing 100 GB of free space. This virtual machine was running Ubuntu 18.04 with kernel v4.15.0. Our Artifice volume was 16 GB and formatted with EXT4. For our benchmark, we used bonnie++ version 1.97 without any additional flags.

As shown in Table 3, our Artifice implementation using a relatively slow secret sharing library running on a commodity SSD provides performance on par with USB 2.0 flash drives[46] and thoroughly surpasses the write throughput of recent competing systems[10,11,17] without compromising the performance of the public volume. As the current bottleneck is a naive secret sharing implementation, further improvements can be made by leveraging processor vector instructions or fast Fourier transforms, as previously discussed. Even without those improvements, Artifice's performance is sufficient for most basic tasks including compressed 1080p video playback.

11. CONCLUSION

Artifice is an operationally secure deniable block device that addresses the problem of hiding a users capability of running a deniable storage system. The use of combinatorial security in addition to more conventional techniques, self repair functionality, and comparatively simple solutions to the challenges posed by multiple snapshot attacks and flash devices results in a system that addresses the challenges posed by a more realistic and knowledgeable adversary. We demonstrate that this system can be tuned to survive hidden data overwrite from public file system operations, while also resulting in significantly improved performance and usability when compared to previous designs. A deniable storage system such as Artifice provides a much needed tool to ensure the continued free flow of information in suppressed or surveilled environments.

DECLARATIONS

Acknowledgments

We would like to thank Thomas Schwarz for assistance with survivability calculations and modeling.

Authors contributions

Made substantial contributions to the threat model, design, and evaluation: Barker A, Gupta Y, Miller E, Long DDE

Provided significant assistance in the development of an operational security model: Hughes J

Availability of data and materials

Not applicable.

Financial support and sponsorship

This work was supported by National Science Foundation grant numbers (IIP-1266400, IIP184545), and award number (CNS-1814347), and by the industrial partners of the Center for Research in Storage Systems (CRSS). James Hughes did not receive financial support.

Conflicts of interest

All authors declared that there are no conflicts of interest.

Ethical approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Copyright

REFERENCES

1. Wikipedia contributors. Rubber-hose Cryptanalysis - Wikipedia, The Free Encyclopedia; 2020. Available from https://en.wikipedia.org/wiki/Rubber-hose_cryptanalysis. [Last accessed on 27 January 2020].
2. Mull J. How a Syrian Refugee Risked His Life to Bear Witness to Atrocities. Toronto Star Online 2012 March. Available from: https://www.thestar.com/news/world/2012/03/14/how_a_syrian_refugee_risked_his_life_to_bear_wit ness_to_atrocities.html. [Last accessed on 11 February 2020]
3. Wikipedia contributors. Key Disclosure Law - Wikipedia, The Free Encyclopedia; 2020. Available from: https://en.wikipedia.org/wiki/Key_disclosure_law. [Last accessed on 27 January 2020].
4. McDonald AD, Kuhn MG. StegFS: A Steganographic File System for Linux. In: International Workshop on Information Hiding. Springer; 1999. pp. 463-77.
5. Troncoso C, Diaz C, Dunkelman O, Preneel B. Traffic Analysis Attacks on a Continuously-Observable Steganographic File System. In: International Workshop on Information Hiding. Berlin, Heidelberg: Springer Berlin Heidelberg; 2007. pp. 220-36.

6. Truecrypt Foundation. Truecrypt; 2020. Available from: http://truecrypt.sourceforge.net. [Last accessed on 11 February 2020]
7. Czeskis A, Hilaire DJS, Koscher K, Gribble SD, Kohno T, et al. Defeating Encrypted and Deniable File Systems: TrueCrypt V5.1a and the Case of the Tattling OS and Applications. In: Proceedings of the 3rd Conference on Hot Topics in Security (HOTSEC '08). Berkeley, CA, USA: USENIX Association; 2008.
8. Goldreich O. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87). ACM; 1987. pp. 182-94.
9. Goldreich O, Ostrovsky R. Software Protection and Simulation on Oblivious RAMs. *J ACM* 1996;43:431-73.
10. Blass EO, Mayberry T, Noubir G, Onarlioglu K. Toward Robust Hidden Volumes Using Write-Only Oblivious RAM. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14). New York: ACM; 2014. pp. 203–14.
11. Chakraborti A, Chen C, Sion R. DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries. Computing Research Repository (CoRR) 2017;abs/1706.10276. Available from: http://arxiv.org/abs/1706.10276. [Last accessed on 11 February 2020]
12. Shamir A. How to Share a Secret. *Communications of the ACM* 1979;22:612-13.
13. Anderson R, Needham R, Shamir A. The Steganographic File System. In: Aucsmith D, editor. International Workshop on Information Hiding. Berlin, Heidelberg: Springer Berlin Heidelberg; 1998. pp. 73-82.
14. Pang H, Tan K, Zhou X. StegFS: A Steganographic File System. In: Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405); 2003. pp. 657-67.
15. Hand S, Roscoe T. Mnemosyne: Peer-to-Peer Steganographic Storage. In: Peer-to-Peer Systems. Berlin, Heidelberg: Springer Berlin Heidelberg; 2002. pp. 130-40.
16. Rabin MO. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *J ACM* 1989;36:335–48.
17. Chen C, Chakraborti A, Sion R. PD-DM: An Efficient Locality-preserving Block Device Mapper with Plausible Deniability. *Proceedings on Privacy Enhancing Technologies* 2019;2019:153-71.
18. Skillen A, Mannan M. On Implementing Deniable Storage Encryption for Mobile Devices. In: 20th Annual Network & Distributed System Security Symposium; 2013. Available from: https://spectrum.library.concordia.ca/975074/. [Last accessed on 11 February 2020]
19. Peters T, Gondree MA, Peterson ZNJ. DEFY: A Deniable, Encrypted File System for Log-Structured Storage. In: 22nd Annual Network and Distributed System Security Symposium, (NDSS). The Internet Society; 2015.
20. SignalApp. Github: WhisperYAFFS; 2011. Available from: https://github.com/signalapp/WhisperYAFFS/wiki. [Last accessed on 11 February 2020]
21. Zuck A, Shriki U, Porter DE, Tsafrir D. Preserving Hidden Data with an Ever-Changing Disk. In: Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17). New York: ACM; 2017. pp. 50-5.
22. Barker A, Sample S, Gupta Y, McTaggart A, Miller EL, et al. Artifice: A Deniable Steganographic File System. In: 9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19). Santa Clara: USENIX Association; 2019.
23. Chang B, Wang Z, Chen B, Zhang F. MobiPluto: File System Friendly Deniable Storage for Mobile Devices. In: Proceedings of the 31st Annual Computer Security Applications Conference. ACSAC 2015. New York: ACM; 2015. pp. 381-90.
24. Kerckhoff A. La Cryptographie Militaire. *Journal des Sciences Militaires* 1883;IX.
25. Zhong R. China Snares Tourists' Phones in Surveillance Dragnet by Adding Secret App 2019 July. Available from: https://www.nytimes.com/2019/07/02/technology/china-xinjiang-app.html. [Last accessed on 11 February 2020]
26. Edge J. Thwarting the Evil Maid. LWNnet 2015 July. Available from: https://lwn.net/Articles/651021/. [Last accessed on 11 February 2020].
27. Idrassi M. Veracrypt; 2020. Available from: https://www.veracrypt.fr/en/Home.html. [Last accessed on 11 February 2020].
28. Resch JK, Plank JS. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In: Proceedings of the 9th USENIX Conference on File and Stroage Technologies. FAST'11. USA: USENIX Association; 2011. p. 14.
29. Rivest RL. All-or-nothing encryption and the package transform. In: Biham E, editor. Fast Software Encryption. Berlin, Heidelberg: Springer Berlin Heidelberg; 1997. pp. 210-18.
30. Reed IS, Solomon G. Polynomial Codes Over Certain Finite Fields. Journal of the Society for Industrial and Applied Mathematics 1960;8:300-4.
31. Broz M. dm-crypt; 2020. Available from: https://gitlab.com/cryptsetup/cryptsetup. [Last accessed on 11 February 2020]
32. Rosenblum M, Ousterhout JK. The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems 1992;10:26-52.
33. Zuck A, Li Y, Bruck J, Porter DE, Tsafrir D. Stash in a Flash. In: 16th USENIX Conference on File and Storage Technologies (FAST 18). Oakland, CA: USENIX Association; 2018. pp. 169-88.
34. Chen C, Chakraborti A, Sion R. INFUSE: Invisible plausibly-deniable file system for NAND flash. Proceedings on Privacy Enhancing Technologies 01 Oct 2020;2020:239-54.
35. Aaron Griffin and others. The Arch Linux Wiki: Solid state drive; 2020. Available from: https://wiki.archlinux.org/index.php/Solid_state_drive. [Last accessed on 11 February 2020]
36. The Debian Project. Debian Wiki: SSD Optimization; 2020. https://wiki.debian.org/SSDOptimization. [Last accessed on 11 February 2020]

37. Broz M, Matyás V. The TrueCrypt On-Disk Format–An Independent View. *IEEE Security Privacy* 2014;12:74–77.
38. Bjørling M. From Open-Channel SSDs to Zoned Namespaces. Boston, MA: USENIX Association; 2019. VAULT-2019: 1st USENIX Conference on Linux Storage and Filesystems.
39. Linux Documentation Maintainers. The Linux Kernel user's and Administrator's Guide: Device Mapper: dm-zoned. Available from: https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-zoned.html. [Last accessed on 11 February 2020]
40. Lee C, Sim D, Hwang J, Cho S. F2FS: A New File System for Flash Storage. In: 13th USENIX Conference on File and Storage Technologies (FAST 15). Santa Clara, CA: USENIX Association; 2015. pp. 273-86.
41. Gray J. Empirical Measurements of Disk Failure Rates and Error Rates; 2005. Available from: https://www.microsoft.com/en-us/research/publication/empirical-measurements-of-disk-failure-rates-and-error-rates/. [Last accessed on 27 January 2020]
42. Silverstone D. Library for Shamir Secret Sharing in Galois Field 2**8; 2006. Available from: https://github.com/jcu shman/libgfshare. [Last accessed on 27 January 2020]
43. Google. CityHash, A Family of Hash Functions for Strings; 2013. Available from: https://code.google.com/p/cityhash. [Last accessed on 14 May 2021].
44. Plank JS, Greenan KM, Miller EL. Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions. In: FAST-2013: 11th Usenix Conference on File and Storage Technologies. San Jose, CA: USENIX Association; 2013.
45. Knuth DE. In: The Art of Computer Programming, volume 2 (3rd ed.): Seminumerical Algorithms. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 1997. p. 505.
46. Throckmorton Z. USB 3.0 Flash Drive Roundup. Anandtech 2011 July. Available from: https://www.anandtech.com/show/4523/usb-30-flash-drive-roundup/3. [Last accessed on 11 February 2020]