Artifice: Data in Disguise

Austen Barker, Yash Gupta, Sabrina Au, Eugene Chou, Ethan L. Miller, Darrell D. E. Long
University of California, Santa Cruz
{atbarker, ygupta, scau, euchou, elm, darrell}@ucsc.edu

***Abstract*—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 and similar credentials. For some users this creates a need for hidden volumes that provide plausible deniability or the ability to deny the existence of sensitive information. Plausible deniability directly impacts groups such as democracy advocates relaying information in repressive regimes, journalists covering human rights stories in a war zone, or NGO workers hiding food shipment schedules from violent militias. All of these users would benefit from a plausibly deniable data storage system. Previous deniable storage solutions only offer pieces of an implementable solution. We introduce Artifice, the first tunable, operationally secure, self repairing, and fully deniable storage system.**

With Artifice, hidden data blocks are split with Shamir Secret Sharing to produce a set of obfuscated carrier blocks that are indistinguishable from other pseudo-random blocks on the disk. The blocks are then stored in unallocated space and possess a self-repairing capability and rely on combinatorial security. Unlike preceding systems, Artifice addresses problems regarding flash storage devices and multiple snapshot attacks through comparatively simple block allocation schemes and operational security. To hide the user's ability to run a deniable system and prevent information leakage, Artifice stores its driver software separately from the hidden data.

I. 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. In some situations, the possession of an encrypted file or disk can expose a user to coercive cryptanalysis tactics, or worse [1]. In such situations, such as crossing the border of a country with a repressive regime, it becomes 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 of exfiltrating 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 [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 encapsulated by an open public file system 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, a variety of 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 has demonstrated distinctive "tells" that enable an adversary knowledgeable of their design to quickly discover them. Some systems, such as StegFS [4], do not disguise data accesses with deniable operations, enabling an adversary to compare two images of the disk and find the hidden volume with a *multiple snapshot attack* [5]. On-the-fly-encryption systems that include hidden volumes like TrueCrypt [6] fail to adequately address information leakage to a public volume through user programs [7]. While they hide data and disguise accesses to a hidden volume, oblivious RAM (ORAM) based systems such as HIVE [8] or Datalair [9] incur significant performance penalties for both hidden and public volumes – presenting a "fingerprint" detectable through basic benchmarking techniques. Lastly, no existing approach successfully addresses deniability for the existence of the software or driver necessary for accessing a deniable storage system.

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.

In this paper, 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 deniablility 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 [10] to generate pseudo-random *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 pseudo-random 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 don't cause irreparable harm.

Artifice's metadata locations are algorithmically generated from a passphrase that must be supplied to find and use the hidden volume. Without the correct passphrase, 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.

II. The Problem of Plausibly Deniable Storage

The most likely scenario for the use of a deniable storage system entails the adversary gaining unfettered access to a device for a short period of time. An example of which would be inspection of a device at a border crossing. Unlike previous work, we assume the existence of a significantly stronger adversary, more in line with the capabilities of a intelligence or law enforcement agency, that would likely be encountered by users of a plausibly deniable system.

A. Adversary Model

We assume the adversary can confiscate the user's device and perform any *static* forensic analysis that they deem warranted. This adversary is capable of taking multiple static snapshots of the device at different points in time and comparing those snapshots in an attempt to discover the existence of hidden data. The adversary can also install malware on the user's operating system, so long is it is not firmware based like a bootkit. Similarly the adversary can monitor the user's interactions with external network infrastructure for suspicious behavior. That said, they cannot continuously monitor the user's actions at all times. Should the adversary discover a suspect aspect of the user's device such as an undisclosed partition, hidden information, or suspicious software, they may force the user to reveal a password, encryption key, or other sensitive information possibly through the threat of legal penalties [3] or the use of a rubber-hose attack [1]. Lastly we must assume that the adversary has knowledge of deniable storage systems and their capabilities. As such, the security of a system should not rely on the secrecy of its design or "security through obscurity" [11].

While we assume that our adversary is relatively powerful we must also keep in mind that even a sophisticated intelligence agency will possess considerable but ultimately finite resources to carry out an attack. As a result, they will not escalate to more involved methods, such as forensic analysis, snapshot attacks, or coercive tactics, unless they believe it is probable that a user is running a deniable system.

Several of the previous deniable storage systems support the feature of multiple levels that correspond with the sensitivity of the data [4], [12], [13]. The intention being that under coercion the user could reveal one, less sensitive, level of the system while keeping others secret. We assume this stratagem does not hold if the adversary has knowledge of a deniable storage system's capabilities as the presence of one level will raise suspicion about the presence of additional levels.

B. Design Requirements

Considering this adversary, we can derive the following series of design requirements for a deniable storage system.

1) **Render hidden information indistinguishable from free space:** 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.
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.* demonstrated that the system was plagued by information leakage through both the features of the Windows operating system and applications such as Microsoft Word [7]. It has also been made apparent by Troncoso *et al.* 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 [5].
3) **Mitigate the effects of overwrites by the public file system:** In normal operations the user will be primarily interacting with the public volume. When accessing the public system there should be (i) no trace of the hidden volume's existence, such as metadata structures or access credentials, visible to an observer and (ii) normal write behavior. This requirement, intended to prevent information leakage to an adversary also prevents the public system from knowing where the hidden data is stored and therefore avoid accidentally overwriting it. A deniable storage system hiding within unallocated space must therefore provide some reasonable protection from overwrite.

TABLE I
PLAUSIBLY DENIABLE SYSTEMS

System	Overwrite Protection	Indistinguishability	Information Leakage Resistance	Deniable Changes	Deniable Software
Veracrypt [14]	✓	✓	-	-	-
StegFS (McDonald) [4]	Probabilistic	✓	-	-	-
StegFS (Pang) [12]	✓	-	-	-	-
Hive [8]	✓	✓	-	ORAM	-
Datalair [9]	✓	✓	-	ORAM	-
DEFY [13]	✓	✓	-	-	-
Mobiflage [15]	✓	✓	-	-	-
Mobipluto [16]	✓	✓	-	-	-
Ever Changing Disk [17]	Probabilistic	✓	-	User behavior	-
PD-DM [18]	✓	✓	-	ORAM	-
Artifice	Probabilistic	✓	✓	User Behavior	✓

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 [8], [9], [12], [18] or with hiding hidden data accesses within other random information [17].
5) **Hide the user's ability to run a deniable storage system:** Since deniable storage systems inherently possess significant drawbacks it is unlikely for the average user to keep a copy 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 [8], [9], [18]. 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 satisfying these requirements inevitably result in multiple design compromises. For instance, to protect against hidden data overwrite or to disguise access patterns, performance is inevitably affected by IO amplification from reading, writing, or analyzing more blocks per operation.

III. 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. Artifice obfuscates data and provides protection from accidental overwrite using an IDA such as Shamir Secret Sharing [10] to generate a set of pseudo-random 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 implying 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.

A. Obfuscation and Redundancy

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 either Shamir Secret Sharing [10] or non-systematic Reed-Solomon erasure codes [19] to generate carrier blocks. Both of which provide an (n, k) scheme where at least k carrier blocks out of a set of n are needed to reconstruct the original data.

Carrier block overwrite will occur through operations performed by the public system. Artifice treats overwrites by the

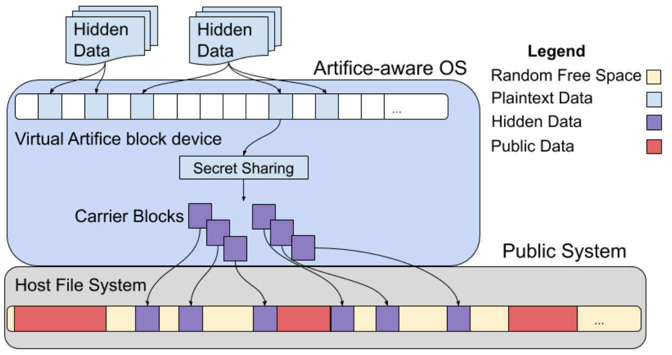


Fig. 1. System overview of Artifice. The Artifice kernel module resides in a separate operating system contained on removable media. The public system includes the public file system that Artifice hides in and the public OS. Free space in the public file system should be filled with pseudo-random blocks.

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.

Unlike previous systems, Artifice relies on combinatorial security as well as encryption [4], [8], [9], [13], [18]. Without knowledge of which carrier blocks correspond to what data blocks 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 $O\left(\binom{N}{k}\right)$ 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 then we can approximate that out of the 2^{168} block combinations, on the order of 2^{27} combinations will yield a valid data block.

There is of course a trade off between resilience, security, and performance but using an IDA allows for a measure of tunability. The larger the threshold k required to rebuild the data, the more secure 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. Whereas decreasing the number of carrier blocks can improve performance.

In its default configuration Artifice uses Shamir Secret Sharing [10] as an IDA to split data blocks into sets of pseudo-random carrier blocks. Artifice 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. So long as $k < n$ then 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.

Artifice can also use an erasure code as an information dispersal and obfuscation algorithm that provides improved space efficiency. In such an approach Artifice combines pseudo-random appearing entropy blocks and a pool of data blocks with Reed-Solomon codes to produce a set of carrier blocks.

For example if we have $d \leq k$ data blocks and $e = k - d$ entropy blocks, after encoding we are left with $m = n - k$ carrier blocks and the e entropy blocks. The data blocks are then discarded and the carrier blocks are stored in the unallocated space of the file system. The entropy blocks are then stored in a known, external location. If $m < e + d$, then we require entropy blocks in order to reconstruct the original data. Whereas if $m \geq e + d$ then we do not require entropy blocks to reconstruct. For example, if we have $d = 2$ data blocks and $e = 3$ entropy blocks resulting in $k = 5$, and assuming that $n = 9$, we arrive at a set of $m = 4$ carrier blocks after encoding. Since the two plaintext data blocks are discarded and not written to disk, we are left with seven blocks that can be used to reconstruct the original data. Out of this set of $n - d$ blocks, only k are needed to reconstruct the original data. The numbers m, d, and e can each be adjusted by the user to provide more resiliency, performance, or security as desired.

Artifice has multiple ways to acquire high entropy data from deniable sources such as a user's DRM (Digital Rights Management) protected media files. The presence of which on a publicly visible file system is not suspicious. So long as $m < d + e$, without the entropy blocks, the original data is unrecoverable.

With an erasure coding based scheme it is possible to map multiple data blocks to a single pool of carrier blocks and provide improved space efficiency over secret sharing. Although we must weigh the advantage of improved space efficiency against the additional complexity and inconvenience of requiring additional entropy data.

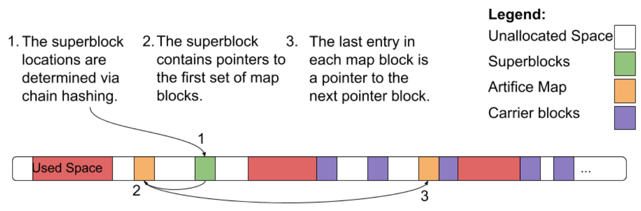
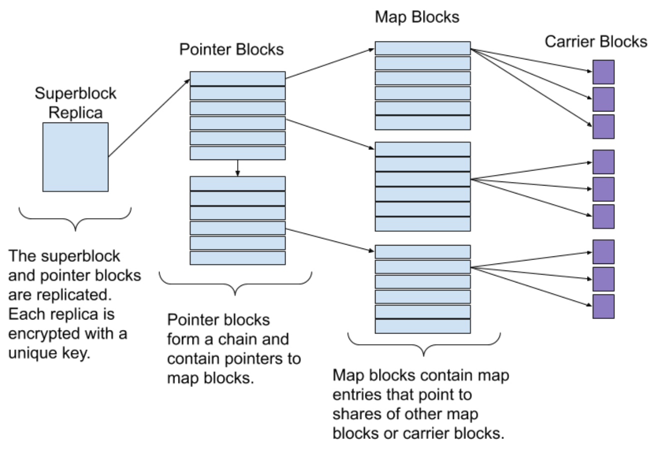

1. The superblock locations are determined via chain hashing.
2. The superblock contains pointers to the first set of map blocks.
3. The last entry in each map block is a pointer to the next pointer block.
Legend:
Unallocated Space
Superblocks
Artifice Map
Carrier blocks
Used Space
1
2
3
...

Fig. 2. The process of locating Artifice superblocks through chain hashing a user's passphrase.


Map Blocks
Pointer Blocks
Carrier 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

Fig. 3. The design of the Artifice Map.

B. 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 Fig. 3). 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 for a user configurable number of redundant superblocks which provide general information about the metadata structures and the locations of the Artifice Map carrier blocks (Fig. 2). The superblock is replicated to protect against overwrite and each replica is encrypted with a different unique key derived from the passphrase. The location of each replica is defined by a hash of the previous replica's location. Artifice should be configured to generate more replicas than shares of a data block to provide a better probability of survival.

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 superblocks. Information about the pointer blocks is stored in the superblocks. 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.

C. Block Operations and Self-Repair

Disk operations in Artifice occur on block IO requests similarly to other device mappers such as dm-crypt [20]. Although instead of encrypting data blocks, the blocks are split into carrier blocks on writes and reconstructed on reads.

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 locations of superblock replicas. The superblocks then provide 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, such as `ext4`, and mount it as they would any other disk. Additional data integrity protections can be provided by this user specified 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 normal, one can simply discard the passphrase. Without the passphrase information retrieval is combinatorically 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.

D. Public File System Considerations

In order 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 [21], it is best to place only one carrier block of out a set in each segment. So that the impact of garbage collecting 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.

Additionally there is the problem of a deniable reason for large amounts of pseudo-random information in the free space of a file system. The naive method is to fill the free space of a disk with pseudo-random bits prior to initializing a hidden volume [6]. The drawback of this approach is that filling the unallocated portions of the disk with unexplained pseudo-random information could be considered suspicious. Therefore it would be advisable for a user to run the public system with a secure deletion utility or a similar producer of pseudo-random information to provide a deniable reason for pseudo-random free space.

Should Artifice be used alongside an encrypted file system it must also be able to understand the algorithms used and the key. Similarly, to enable nested Artifice volumes we must be able to treat Artifice itself in the same manner as a public file system. To access a given Artifice instance we must have the passphrase(s) for all levels above it.

E. Operational Security of the Artifice Driver

Even though Artifice provides deniability for the its driver by storing it on a seperate device, there are security concerns that arise. 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 arrangments 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. Additional options to avoid exposing the driver include downloading it through secure means like TOR [22] or an HTTPS secured website and carrying it on an easily concealed MicroSD memory card. 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.

IV. 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 may 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 [13], or are intended to operate as drive firmware [17]. Since custom firmware would be suspicious and raw flash devices are still relatively uncommon, Artifice must attempt to address these challenges through other means.

A. TRIM

Most modern file systems support the TRIM function, which notifies an SSD which 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 a possibility of accesses to the original data after a block has been subject to TRIM. When reading from TRIMmed blocks the SSD could either return 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 pseudo-random 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 [23], [24]. 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 is not sufficient to fully solve the problem that TRIM presents.

Fortunately, 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], [15], [25]. 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 like a mechanical disk from the perspective of Artifice and the public file system.

B. 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 [26]. 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 [27], [28]. 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 deleted blocks of zones that have not yet been garbage collected by the file system or dm-zoned if those zones already contained pseudo-random or encrypted information. As virtual block device 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 like Zone Namespaces have not yet been integrated into commonly available hardware.

V. Multiple Snapshot and Disguising Accesses

A multiple snapshot attack is a significant problem that most recent deniable storage systems attempt to defend against [8], [9], [13], [17], [18]. 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. Artifice instead prioritizes hiding the user's capability of running a deniable system while still providing some methods for defending against a multiple snapshot attack which rely on providing deniable reasons for the changes in a disk's free space.

The first solution is proper operational security. 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 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.

Another approach 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 limitations to this approach. First is that using a secure delete program or key revocation technique is essential. Otherwise pseudo-random blocks that cannot be decrypted by a user's key would be inherently suspicious. Second is that hiding data in frequently changed sections of the disk increases the probability of overwrite. 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 prior to provide free blocks prior to writing to an Artifice volume. Due to the stricter requirements placed on the user's behavior, such measures should not be carried out unless there is a high possibility of an adversary carrying out a multiple snapshot attack.

VI. 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} to 10^{-15} [29]. If a block can be read at all it is extremely unlikely to be incorrect. Blocks that are marginal can be remapped by the drive, or by 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, $\text{SizeShamir}(s, n, k)$ 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 SizeShamir() is instead the effective size of the entire instance when accounting for write amplification.

$$\underset{\text{Surv.}}{\Pr}(k,n) = (\sum_{i=0}^{n-k} p^i \binom{n}{i} (1-p)^{n-i})^{\text{SizeShamir}(s,n,k)\cdot t}$$

In the case of our Reed-Solomon scheme we must also account for the entropy blocks, e, and the possibility of multiple data blocks, d, mapping to a single set of carrier blocks m. In this case we can lose up to $m-d$ blocks out of $e+m$ stored. It is important to note that unlike the secret sharing approach, the reconstruction threshold is dependent on the number of carrier blocks. The number of vulnerable blocks is given as $\text{SizeRS}(s, m, e, d)$.

$$\underset{\text{Surv.}}{\Pr}(e,d,m) = (\sum_{i=0}^{m-d} p^i \binom{e+m}{i} (1-p)^{e+m-i})^{\text{SizeRS}(s,m,e,d)\cdot t}$$

With these two functions 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. In the case of our Reed-Solomon scheme we assume that our code word contains one entropy block and either one or two data blocks. 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.

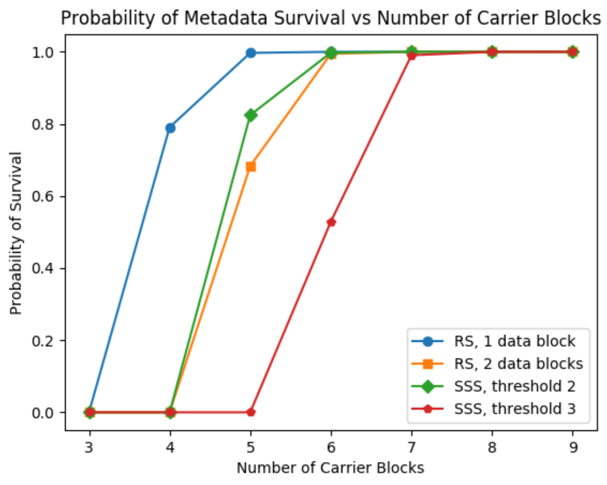
Fig. 4 shows the survival probability of our example instance over the course of 365 repair cycles for both the metadata and the entire Artifice instance with a variety of different encoding techniques and numbers of carrier blocks. From these calculations we can see that there is a number of carrier 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 a Reed-Solomon erasure code can provide better reliability due to improved error correction capabilities and a smaller footprint on the disk at the cost of additional operational overhead due to the required entropy blocks. On the other hand Shamir Secret Sharing would usually requires 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 Fig. 5, the smaller the Artifice volume the higher 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 (Fig. 6) we can observe an exponential decrease in reliability after approximately 4 GB. Finally, when regarding the amount of free space available to Artifice (Fig. 7) we can see that 256 GB of unallocated space provides a promising probability of survival for our Artifice instance.

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 Reed-Solomon scheme the write amplification is improved such that the size is only amplified by a factor of $\frac{\text{\# of carrier blocks}}{\text{\# of data blocks}}$.

Additional gains to survivability could be obtained through Artifice identifying frequently overwritten sections of the disk over multiple sessions. These regions can then be avoided by Artifice's block allocation function. Although there is a trade off with our defense against multiple snapshot attacks as a deniable reason for changes to an infrequently used portion of the disk may be difficult to provide.

This shows that Artifice can sustain severe damage, as long as the user i) maintains a certain percentage of the encapsulating file system free for Artifice to occupy, and ii) 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



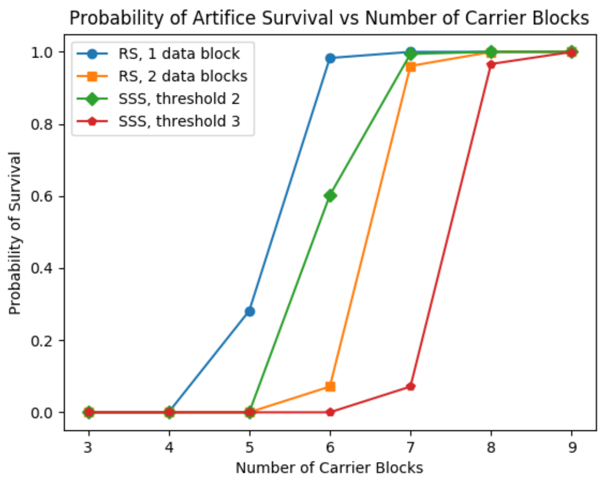


Fig. 4. Probability of survival for Artifice metadata in a variety of configurations using both Reed-Solomon (RS) and Shamir Secret Sharing (SSS). Probabilities are calculated assuming 512 GB of free space, 5 GB written between repair cycles, 5 GB Artifice volume, and over the course of 365 repair cycles.

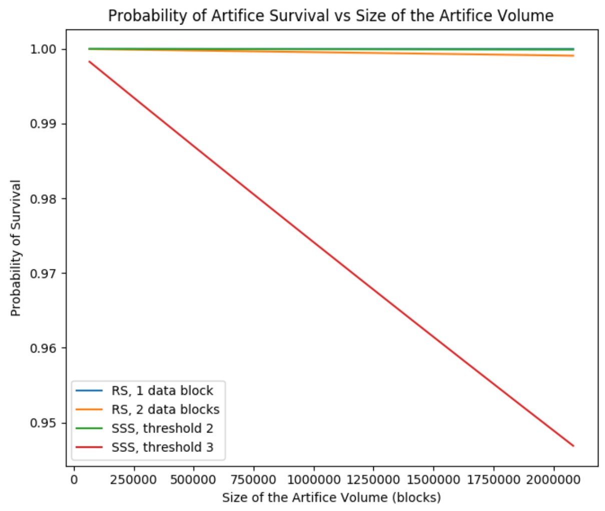


Fig. 5. Probability of survival with varying Artifice volume sizes ranging from 256 MB to 4 GB. 5 GB of writes between repair operations and 512 GB of unallocated space.

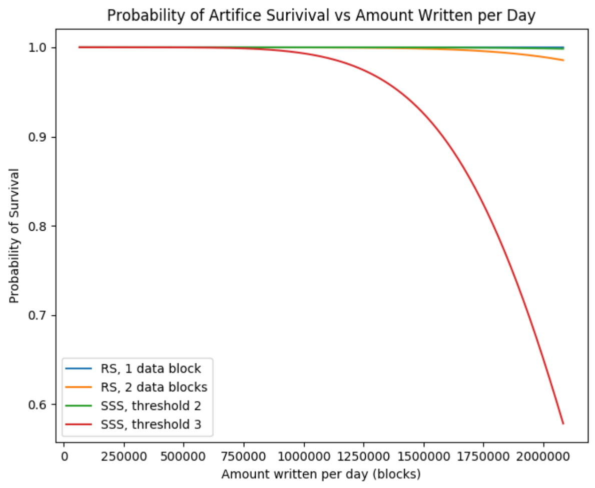


Fig. 6. Probability of survival with varying sizes of writes between Artifice invocations from 256 MB to 8 GB. 512 GB of unallocated space and a 5 GB Artifice volume.

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.

VII. Evaluation

To demonstrate and test its viability we have 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 [30] 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 it is the default Linux page size. Block checksums use a ported version of the `CityHash` library [31] and the passphrase is hashed with `SHA256`.

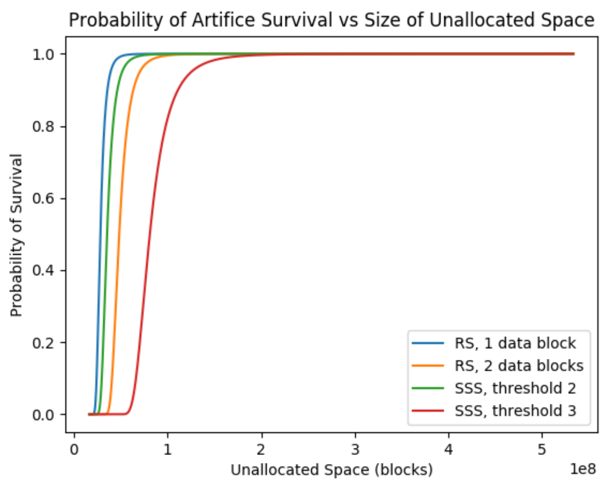


Fig. 7. Probability of survival with varying sizes of unallocated spaces from 64 GB to 1 TB. 5 GB of writes between repair operations and a 5 GB Artifice volume.

A. 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 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 [8], [9], [18]. We consider this threshold to be the performance of a small removable storage device such as a USB flash drive. The largest sources of overhead are the additional processing that secret sharing or Reed-Solomon will require and write amplification from writing multiple carrier blocks for each data block. Performance oriented Shamir Secret Sharing and erasure code implementations can be achieved through the use of vector instructions [32] and Fast-Fourier transforms [33] to accelerate Galois field operations. Despite these methods, reading blocks from scattered 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 [8], [9], [18].

Our test machine was equipped with an i7-4790 CPU, 32 GB of RAM, and a 480GB Intel 660p SSD. To better model an average laptop computer we ran all benchmarks on a Virtualbox virtual machine with 4 processor cores, 4 GB of RAM, and a virtual disk formatted with FAT-32 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 run through the Pilot benchmark framework [34] set to achieve a confidence interval of 95%.

As seen in Table II 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 [35] and thoroughly surpasses the write throughput of recent competing systems [8], [9], [18] 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.

VIII. Related Work

Over the past few decades there have been several attempts at plausibly deniable storage. 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.* [36] were the first to propose a steganographic file system and described two possible approaches. First a system of cover files to camouflage hidden data and the second being hiding 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.* [12] 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 [37] proposes replacing StegFS's simple replication technique with Rabin's Information Dispersal Algorithm [38] in order to provide greater durability and improve write amplification across nodes. Due to its design as a peer-

TABLE II
ARTIFICE PERFORMANCE

	Public Volume		**Artifice**	
	Throughput	***CPU***	***Throughput***	***CPU***
Read	299.285 MB/s±5.527	21.700% ± 0.655	51.876 MB/s ±1.232	9.800% ± 0.362
Write	307.458 MB/s±46.682	22.667% ± 2.930	34.476 MB/s ±2.291	1.700% ± 0.399

to-peer system, however, it does not hold up to our assumed adversary.

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 [9] and HIVE [8] combine a hidden volume with ORAM [39], [40] 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 [9]. Random disk write patterns and unexplained slow performance compared to the raw disk can possibly be viewed as suspicious.

Recently Chen *et al.* published PD-DM [18], a device mapper based approach aimed at addressing the poor performance of ORAM dependent systems [8], [9]. 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 [15], 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 is present to provide deniability while ignoring that the presence of a disk partition containing unexplained random information is suspicious.

DEFY [13] is a log structured deniable file system designed for host controlled flash devices and is based on WhisperYAFFS [41]. 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.* proposed the Ever-Changing Disk (ECD) [17], a firmware design that splits a device into hidden and public volumes where hidden data is written alongside pseudo-random 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.

IX. CONCLUSION

Artifice is an operationally secure deniable block device that addresses the problem of hiding a user's capability of running a deniable storage system. The use of combinatorial security, 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 have demonstrated that this system can easily 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.

ACKNOWLEDGMENTS

We would like to thank Thomas Schwarz for assistance with the survivability calculations. This research was supported in part by the National Science Foundation grant number IIP-1266400, award CNS-1814347, and by the industrial partners of the Center for Research in Storage Systems.

REFERENCES

[1] Wikipedia contributors, "Rubber-hose Cryptanalysis — Wikipedia, The Free Encyclopedia," 2020, [Online; accessed 27-January-2020].

[2] J. Mull, "How a Syrian Refugee Risked His Life to Bear Witness to Atrocities," *Toronto Star Online*, March 2012.

[3] Wikipedia contributors, "Key Disclosure Law — Wikipedia, The Free Encyclopedia," 2020, [Online; accessed 27-January-2020].

[4] A. D. McDonald and M. G. Kuhn, "StegFS: A steganographic file system for Linux," in *International Workshop on Information Hiding*. Springer, 1999, pp. 463–477.

[5] C. Troncoso, C. Diaz, O. Dunkelman, and B. Preneel, "Traffic analysis attacks on a continuously-observable steganographic file system," in *Information Hiding*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 220–236.

[6] Truecrypt Foundation, "Truecrypt," http://truecrypt.sourceforge.net.

[7] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier, “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, pp. 7:1–7:7.

[8] 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 (CCS ’14)*. New York, NY, USA: ACM, 2014, pp. 203–214.

[9] A. Chakraborti, C. Chen, and R. Sion, “DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries,” *Computing Research Repository (CoRR)*, vol. abs/1706.10276, 2017.

[10] A. Shamir, “How to Share a Secret,” *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[11] A. Kerckhoff, “La Cryptographie Militaire,” *Journal des Sciences Militaires*, vol. IX, 1883.

[12] H. Pang, K. Tan, and X. Zhou, “StegFS: A Steganographic File System,” in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, March 2003, pp. 657–667.

[13] T. Peters, M. A. 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*. The Internet Society, 2015.

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

[15] A. Skillen and M. Mannan, “On Implementing Deniable Storage Encryption for Mobile Devices,” in *20th Annual Network & Distributed System Security Symposium*, February 2013.

[16] B. Chang, Z. Wang, B. Chen, and F. Zhang, “MobiPluto: File System Friendly Deniable Storage for Mobile Devices,” in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC 2015. New York, NY, USA: ACM, 2015, pp. 381–390.

[17] 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 (HotOS ’17)*. New York, NY, USA: ACM, 2017, pp. 50–55.

[18] C. Chen, A. Chakraborti, and R. Sion, “PD-DM: An Efficient Locality-preserving Block Device Mapper with Plausible Deniability,” *Proceedings on Privacy Enhancing Technologies*, vol. 2019, pp. 153–171, 01 2019.

[19] I. S. Reed and G. Solomon, “Polynomial Codes Over Certain Finite Fields,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[20] Milan Broz, “dm-crypt,” https://gitlab.com/cryptsetup/cryptsetup.

[21] M. Rosenblum and J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System,” *ACM Transactions on Computer Systems*, vol. 10, no. 1, p. 2652, Feb. 1992.

[22] R. Dingledine, N. Mathewson, and P. Syverson, “Tor: The second-generation onion router,” in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM04. USA: USENIX Association, 2004, p. 21.

[23] Aaron Griffin and others, “The Arch Linux Wiki: Solid state drive,” https://wiki.archlinux.org/index.php/Solid_state_drive, [Online; accessed 11-February-2020].

[24] The Debian Project, “Debian Wiki: SSD Optimization,” https://wiki.debian.org/SSDOptimization, [Online; accessed 11-February-2020].

[25] M. Broz and V. Matys, “The TrueCrypt On-Disk Format–An Independent View,” *IEEE Security Privacy*, vol. 12, no. 3, pp. 74–77, May 2014.

[26] M. Bjørling, “From Open-Channel SSDs to Zoned Namespaces.” Boston, MA: USENIX Association, Feb. 2019, VAULT-2019: 1st USENIX Conference on Linux Storage and Filesystems.

[27] Linux Documentation Maintainers, “The Linux Kernel user’s and Administrator’s Guide: Device Mapper: dm-zoned,” https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-zoned.html, [Online; accessed 11-February-2020].

[28] C. Lee, D. Sim, J. Hwang, and S. Cho, “F2FS: A New File System for Flash Storage,” in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 273–286.

[29] J. Gray, “Empirical Measurements of Disk Failure Rates and Error Rates,” Tech. Rep., December 2005.

[30] D. Silverstone, “Library for Shamir Secret Sharing in Galois Field 2**8,” https://github.com/jcushman/libgfshare, 2006.

[31] Google, “The CityHash family of Hash Functions,” https://code.google.com/p/cityhash, 2013, [Online; accessed 10-February-2020].

[32] J. S. Plank, K. M. Greenan, and E. L. Miller, “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, February 2013.

[33] D. E. Knuth, *The Art of Computer Programming, volume 2 (3rd ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, p. 505.

[34] Y. Li, Y. Gupta, E. L. Miller, and D. D. E. Long, “Pilot: A framework that understands how to do performance benchmarks the right way,” in *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2016, pp. 169–178.

[35] Z. Throckmorton, “USB 3.0 Flash Drive Roundup,” *Anandtech*, 2011.

[36] R. Anderson, R. Needham, and A. Shamir, “The Steganographic File System,” in *International Workshop on Information Hiding*, D. Aucsmith, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 73–82.

[37] S. Hand and T. Roscoe, “Mnemosyne: Peer-to-Peer Steganographic Storage,” in *Peer-to-Peer Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 130–140.

[38] M. O. Rabin, “Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance,” *J. ACM*, vol. 36, no. 2, pp. 335–348, Apr. 1989.

[39] O. Goldreich and R. Ostrovsky, “Software Protection and Simulation on Oblivious RAMs,” *J. ACM*, vol. 43, no. 3, pp. 431–473, May 1996.

[40] X. Zhou, H. Pang, and K. Tan, “Hiding Data Accesses in Steganographic File System,” in *Proceedings 20th International Conference on Data Engineering*, April 2004, pp. 572–583.

[41] SignalApp, “Github: WhisperYAFFS,” https://github.com/signalapp/WhisperYAFFS/wiki.