Artifice: A Deniable Steganographic File System

Austen Barker Staunton Sample Yash Gupta Anastasia McTaggart Ethan L. Miller
Darrell D. E. Long
University of California, Santa Cruz

Abstract

The challenge of deniability for sensitive data can be a life or death issue depending on location. Plausible deniability directly impacts groups such as democracy advocates relaying information in repressive regimes, journalists covering human rights stories in a war zone, and 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. Artifice is the first tunable, operationally secure, self repairing, and fully deniable steganographic file system.

Artifice operates through the use of a virtual block device driver stored separately from the hidden data. It uses external entropy sources and erasure codes to deniably and reliably store data within the unallocated space of an existing file system. A set of data blocks to be hidden are combined with entropy blocks through erasure codes to produce a set of obfuscated carrier blocks that are indistinguishable from other pseudorandom blocks on the disk. A subset of these blocks may then be used to reconstruct the data. Artifice presents a truly deniable storage solution through its use of external entropy and erasure codes, while providing better durability than other deniable storage systems.

1 Introduction

As everyday use of strong 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. In such situations it becomes necessary for the user to establish full plausible deniability.

This encourages individuals to resort to extreme methods to exfiltrate data from dangerous or restricted environments. In 2011, a Syrian engineer smuggled a micro SD card out in an arm wound in order to exfiltrate information about atrocities in Hama [14]. It is also becoming increasingly common for nations and law enforcement to legally obligate disclosure of encryption keys [24].

Since carrying encrypted files or dedicated hardware is inherently suspicious, a deniable storage system must co-exist with an open public file system to maintain *plausible deniability*. It is highly suspicious if there are visible drivers or firmware, unconventional partitioning schemes, unusable space in a file system, or unexplained changes to disk free space. The hidden file system must therefore operate in such a way that the encapsulating file system and operating system are unaware of the hidden file system's existence, even when faced with a detailed forensic examination. A deniable system must therefore meet four requirements: effectively hide existence of the data, disguise hidden data accesses, have no impact on the behavior of the public system, and hide the software used to access the hidden data.

Existing deniable storage systems only address a subset of these requirements. Some systems such as StegFS [15] do not disguise data accesses with deniable operations, enabling an adversary to compare two images of the disk and find the hidden volume in a *multiple snapshot attack*. While it hides data and disguises accesses, HIVE [3] significantly slows accesses to public volumes. No existing approach successfully addresses deniability for the existence of *both* the data and the storage system itself.

Artifice is a deniable steganographic storage system that seeks to provide functional plausible deniability for both the data and the Artifice system. When the user needs to access the hidden data, they boot into an Artifice-aware operating system, such as a patched Linux Live USB. Booting into a separate OS provides effective isolation from the host OS. Unlike previous systems this does not leave behind suspicious drivers on the user's machine. A user's data blocks are combined with *entropy blocks* using non-systematic erasure codes to generate *carrier blocks*. The carrier blocks are then stored in the unallocated space of the public file system. As the public file system cannot be aware of Artifice's existence, Artifice must protect itself from overwrite by public operations. The aforementioned erasure codes lend Artifice overwrite

tolerance and enable self repair whenever the user boots the Artifice aware OS. Carrier blocks are produced such that they have high entropy, so they cannot be distinguished from other unallocated spaces that are filled with random data, possibly through use of an additional disk encryption system. Entropy blocks are pseudo-random information sourced either from user files, such as DRM protected media, or generated with a pseudo-random number generator. Artifice's location is generated algorithmically from a pass-phrase that must be supplied before it can be discovered. Without the correct pass-phrase, an Artifice instance should be undetectable. Artifice addresses the issue of multiple snapshot attacks through deniably shuffling blocks under the guise of a deniable operation, such as defragmentation, or through operational security measures.

Developing tools that enable privacy are *vital* for nascent democracies, peaceful dissent, and the free dissemination of information. Artifice is designed to provide a safe, deniable means to store information for journalists, activists, and international human rights observers.

2 Background

Plausibly deniable storage is often considered a solved problem, but there has yet to be a working and truly deniable storage solution. While existing solutions all claim to provide plausible deniability, they all possess easily detectable elements or dependencies. These tells can betray the existence of the file system itself or the user's capability of running a plausibly deniable system.

Anderson, *et al.* were the first to propose a steganographic file system [1]. Their second proposed scheme chooses carrier blocks in the unallocated space of the open file system within which the blocks of the hidden file system are stored. Although this work lacked an implementation, many deniable file systems follow this approach.

McDonald and Kuhn implemented a variant of Anderson's second design as a Linux file system based on `ext2`, known as StegFS [12]. StegFS uses a block allocation table to map encrypted data to unused blocks. They argue that because no information can be deduced from the table, they felt no need to continue to obfuscate the existence of their system. The authors argue that while StegFS is detectable, the data will be safe in a lower level. Yet, it is structured such that opening a level of StegFS also opens up all of the lower levels. Another criticism of StegFS is that it attempts to mitigate overwrites by replicating files. While this scheme attempts to ensure that at least one copy of a file survives, it is not space-efficient.

Pang, *et al.* [15] 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 that exposes the existence and maximum size of the hidden volume.

Goldreich [8] and Zhou [25] propose systems to obfuscate access patterns to shared data using Oblivious RAM (ORAM). However, both do not conceal the *presence* of data, only access patterns.

Datalair [5] and HIVE [3] combine a hidden volume with ORAM-like techniques to obscure the volume's existence and thwart the multiple snapshot attack. Accesses to hidden data are disguised amongst random accesses to a public volume. In theory, 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. Random disk write patterns and unexplained slow performance compared to the raw disk can possibly be viewed as suspicious.

DEFY [16] is a log structured deniable file system designed for flash devices based on WhisperYAFFS [19]. DEFY does not adequately protect against hidden data overwrite unless hidden volumes are constantly mounted and requires all file system metadata to be stored in memory.

On-the-fly-encryption (OTFE) is the basis for several approaches to a hidden file system; examples include Rubberhose [2], FreeOTFE [18], TrueCrypt [22], and VeraCrypt [13]. These approaches are 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 [6].

Recently Zuck *et al.* proposed the Ever-Changing Disk (ECD) [26], a firmware design that splits a device into hidden and public volumes where hidden data is 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 deniability of the partitioning scheme and proposed custom firmware could be a vulnerability.

None of the previously described systems hide a user's capability of running a plausibly deniable system from an adversary or address malicious software installed by an adversary.

3 Security and Adversary Model

The security of Artifice lies in the inability of the adversary to distinguish carrier blocks from random data. Each carrier block is derived from a set of data blocks and a set of entropy blocks, but a set of carrier blocks alone is insufficient to generate the original data blocks. Even if the adversary obtains the correct entropy blocks, Artifice relies on combinatorics to protect the data; the attacker does not know *which* entropy blocks were used to encode each data block. The set of carrier blocks and entropy blocks needed to rebuild a single Artifice block could be algorithmically derived from a long key, a pass-phrase, or even by a random value stored as part of the i-node of which the data block is a part. An attacker would then have to examine all possible random values, generate a

set of all possible entropy blocks, and attempt to reconstruct the data block for each value, which is a computationally infeasible task [11]. Without the entropy blocks, there is no key that will decrypt a carrier block, eliminating a common technique for detecting a deniable storage system.

We assume our adversary can monitor all of the user's network communication. The adversary can confiscate the user's computer and perform any *static* forensic analysis they want. The user could be absent and will not be aware of such attacks beforehand, yet, the user may know that such a confiscation has taken place. The adversary can easily break weak encryption if needed. The adversary can force the user to reveal a password or key if they discover any encrypted or undisclosed partition, hidden data, or suspicious software. The adversary *cannot* monitor the user continuously. As such Artifice does not defend against an adversary that has access to the data path and can observe operations on the hidden volume in real time.

In order to address gaps in Artifice's security model, a user must take care to follow proper operational security procedures while using Artifice. For instance, if a device is confiscated from a user it must be assumed that malware or spyware has been installed on the device or that the adversary has obtained an image of the disk. While the threat of malware is partially mitigated by using a seperate OS to access Artifice, it does not defend against the presence of firmware level malware such as a Bootkit [7]. A user must also keep their device on their person at all times, such that they are aware of any access by the adversary. Should the adversary gain access to the device without direct observation from the user, it would be ideal to replace the device.

4 Design

Artifice presents a virtual block device via the Linux Device Mapper framework that encodes user data blocks into carrier blocks using entropy blocks supplied from a source such as DRM-protected media files stored in a public file system.

Artifice combines entropy blocks and data blocks with an (n,k) error correcting scheme such as Reed-Solomon codes [17] 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. Carrier block overwrite will occur through operations performed by the unwitting file system in the intervening time since the hidden data was written. The use of erasure codes helps to alleviate this problem as Artifice treats overwrites by the public file system as bad data and rebuilds the "missing" blocks during

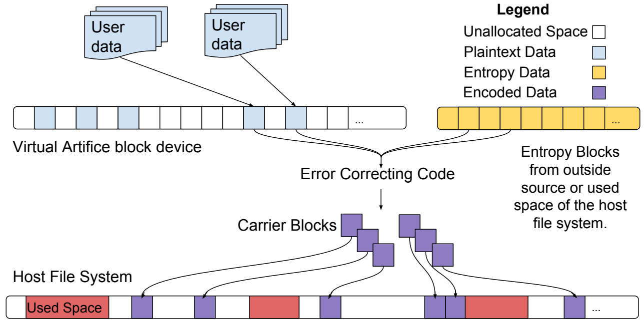

Figure 1: Creation of carrier blocks from data and entropy.

reconstruction. This approach presents a more space efficient and tunable method for addressing hidden data overwrite than previous work such as the replication scheme in StegFS [12]. Additionally, unlike previous systems this scheme relies on combinatorial security as opposed to encryption. Without knowledge of which carrier and entropy blocks correspond to what data blocks an adversary must attempt to reconstruct each permutation of possible carrier and entropy blocks.

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 4 carrier blocks after encoding. Since the two unencoded 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 or security as desired. The larger the number of carrier blocks out of a set required to rebuild the data, the more secure the system. Conversely, a smaller number of blocks required for reconstruction provides more data resiliency in the face of overwrites. One can also increase the number of carrier blocks to provide more resiliency at the cost of space efficiency and performance. Whereas decreasing the number of carrier blocks can improve performance.

Artifice reads data by identifying the carrier blocks and entropy blocks associated with the logical location through a metadata structure called the Artifice Map (shown in Figure 2). The Artifice Map stores metadata that provides a mapping from logical data blocks to physical carrier block locations along with a hash of the logical data block. The map is protected by the same entropy/erasure code scheme described earlier and stored alongside the data blocks. Whether a carrier block has been overwritten is determined by a checksum stored in the map. If a carrier block has been overwritten it is considered an erasure in our encoding scheme. Another checksum of the original data block is stored in the map to verify that the data block was rebuilt successfully. A hash of a user specified passphrase is used to determine the location of a superblock which provides general information such as the location of the Artifice Map. This superblock is replicated and encrypted with a unique key derived from the passphrase

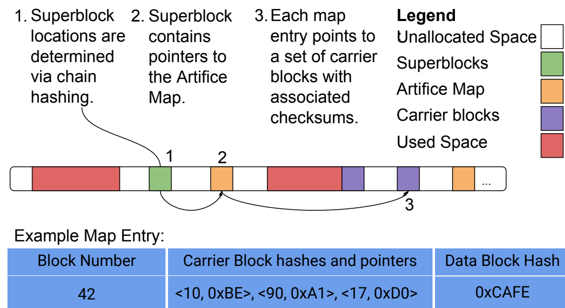

Figure 2: Design and operation of the Artifice Map

to protect against overwrite with each location defined by the a hash of the previous replica's location.

Artifice has multiple ways to acquire high entropy data such as from a user's DRM-protected media files. The presence of which on a publicly visible file system is not suspicious. In addition to finding sources of entropy, Artifice must also disguise its carrier blocks in order to thwart a forensic examination of the hard drive. Ideally the host operating system would provide some form of whole drive encryption such that every block, used or unallocated, would appear random and a carrier block would be indistinguishable from any other block. Even if the user is coerced into providing the decryption key for the drive, it only reveals which blocks are unallocated and does not reveal the existence of the Artifice instance. Without entropy blocks the carrier blocks are just random data like all other unallocated blocks. Alternatively, Artifice can make use of secure drive wiping software to fill unallocated space with random data, providing a place to store carrier blocks. Finally, self destruct is trivial through discarding the passphrase and/or entropy source. Without this information data retrieval is combinatorically infeasible and normal public operations will eventually overwrite Artifice.

Since our adversary can confiscate a device for analysis at any moment, Artifice must be resistant to a multiple snapshot attack. Similar to previous work [3, 5, 26], Artifice can be rendered resistant to such attacks through making extra writes during either repair cycles or normal disk operations. However, a deniable reason and pattern for these writes is required. One possibility is to cover writes with public file system deletions where the newly freed blocks are immediately occupied by Artifice blocks. Another possibility is to shuffle Artifice and public file system data during a defragmentation operation.

Any activity in the open file system has the potential to destroy carrier blocks in Artifice. A second challenge is ensuring Artifice can repair extensive damage caused by normal file system writes in higher levels. One approach is altering the number of carrier blocks and the number of entropy blocks needed to reconstruct. Increasing the overall number of carrier blocks and decreasing the number of blocks required to reconstruct will further mitigate the effects of overwrites. Alternatively we can make smarter choices in the selection of carrier block locations so that an operation in the open file system is less likely to affect more than one carrier block in a set. A combination of these techniques is used to tune Artifice to provide maximum resiliency. The simplest solution Artifice can employ is to randomly scatter carrier blocks around the drive because clustering them together is *more* likely to result in data loss from higher-level file system activity. 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.

5 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} [9]. 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, Artifice will have destruction of its constituent blocks as the norm. As the public file system operates, it will write to believed free blocks, some of which may be Artifice carrier blocks. For instance, assume that each Artifice Map entry represents m carrier blocks, two data blocks, and $k-2$ entropy blocks for a total of n blocks per codeword. The drive being used is 1 TB, where 512 GB is left free for Artifice to hide in. We use an Artifice instance with 5 GB of utilizable space, and about 6.4% of the Artifice block device used for storing the Artifice map. It is assumed that the user writes 5 GB of data per day and mounts Artifice at least once a day to repair any overwritten blocks. Recall from section 3 that we require k out of n blocks to reconstruct our data. We can calculate the survival probability of the Artifice metadata with the following where t is time in days, p is the probability that a carrier block is overwritten, and $Size(s,m,k)$ is the size of the Artifice metadata.

$$\Pr_{\text{Survival}}(k,m) = \left(\sum_{i=0}^{m} p^i \binom{k+m}{i} (1-p)^{k+m-i}\right)^{\text{Size}(s,m,k)\cdot t}$$

We can perform a similar calculation to determine the survival probability for the entirety of an Artifice instance where $Size(s,m,k)$ is instead the effective size of the entire instance when accounting for write amplification. Figure 3 shows the survival probability of our example instance over the course of a year with a repair cycle run each day for both the metadata and the entire instance. 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.

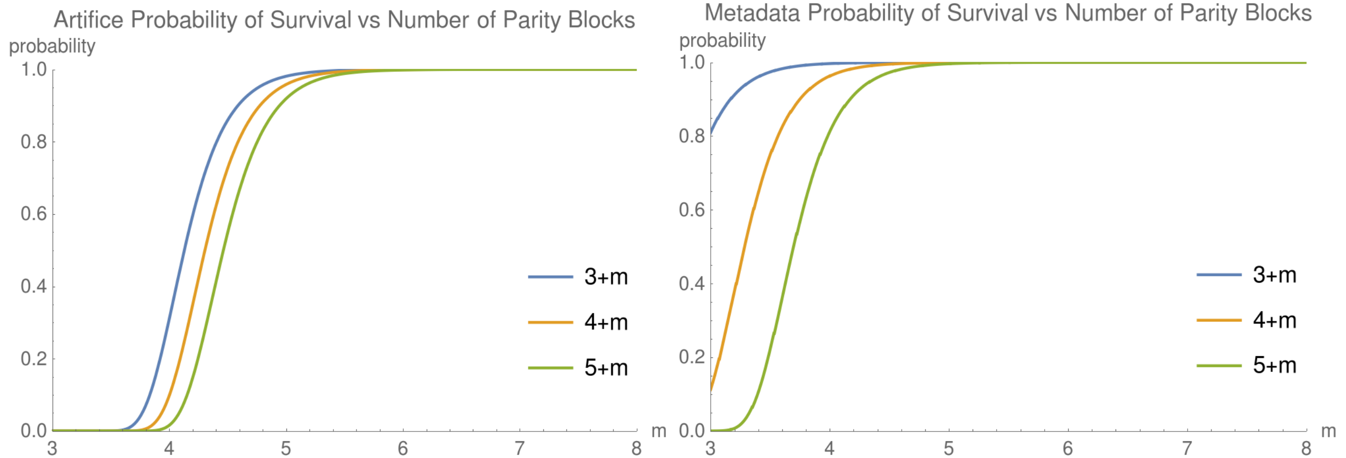

Figure 3: Probability of survival for Artifice data in a variety of configurations where *m* is the number of carrier blocks.

6 Multiple Snapshot and Disguising Accesses

A multiple snapshot attack is a significant problem that most recent deniable storage systems attempt to defend against [3, 5, 6, 16, 26]. While this is theoretically a serious threat to a system, it is easily mitigated through through relatively simple countermeasures.

The first solution is proper operational security. When an adversary gains access to the device, without the user supervising, it must be assumed to be compromised whether through imaging the disk or newly installed malware. The easiest and most reliable response is to replace either the device or the disk. With any data already contained in an Artifice instance copied to the new device, there is then nothing for the adversary to meaningfully compare to the initial snapshot. Although such a scenario is ideal, it will likely not always be practical for a user to replace a device.

Another, more practical, approach is to move enough blocks in the free space not occupied by Artifice that it prevents the adversary from confirming its existence with a reasonable doubt. With a mechanical disk this can be done by keeping the public file system fragmented for the first snapshot and defragmenting *through* the Artifice aware OS such that Artifice has the opportunity to move its blocks and avoid total overwrite. Another approach would be to shuffle public and hidden file system blocks with each write to Artifice, masking accesses to hidden data with dummy writes [26]. The pattern of these dummy operations must reflect changes to the public file system. Therefore it would be advisable to maintain a pool of unimportant public files that could be moved or deleted to provide a deniable reason for changes on the disk.

7 Multiple Levels

It is commonly assumed in the realm of deniable encryption that a person who is coerced can reveal some set of verifiable truths while keeping others secret [2, 12]. The same method of hiding data within the unallocated space of an unaware file system can be applied when the host file system is another instance of Artifice. The user is then able to nest an arbitrary number of instances within one another so that under coercion a user is able to reveal the contents of one Artifice instance while insisting that is all that exists. For example, a journalist might have one set of sources in the first level file system while protecting another more sensitive set of sources. Although the effectiveness of such an approach is questionable if an adversary has prior knowledge of Artifice and its inner workings.

In order to implement nesting, Artifice must also understand its own metadata structures and the metadata of higher level instances. The primary challenge lies in obtaining the metadata of higher level instances in order to determine which blocks are free. To solve this problem Artifice uses the deterministic hash function, CRUSH [23]. It is essential that a lower level be able to view the metadata of higher level instances without exposing the lower level.

8 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. This process occurs 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 not available via the logical interface as they are not mapped into the logical address space. This layer of abstraction and garbage collection presents problems for Artifice.

The FTL may mark Artifice blocks as written which creates an opening for detecting Artifice through forensic analysis. Also the FTL may erase hidden data as part of normal garbage collection operations if it is unaware of Artifice's presence.

Modern operating systems support the TRIM function, which notifies the SSD which flash blocks are no longer in use by the file system, allowing the FTL to correctly handle deletions. Specifically, it notifies the SSD that there is no need to keep copies of pages during garbage collection.

Common forensic analysis of an SSD only uses SATA commands and *sees* the drive as the OS does. Access to raw flash is possible, but not standard in practice because of variation in SSD designs [10]. Many SSDs implement Deterministic Read After Trim (DRAT), in which the SSD returns the same data for blocks that have been TRIMmed, regardless of whether the block survives in flash. 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 [4, 20, 22]. These characteristics make disabling TRIM an ideal choice as we need not worry about hiding carrier blocks among zeroed data or carrier blocks being wiped by garbage collection.

The Artifice file system can still write to blocks in the free space of the open system. No TRIM command would be issued for these and Artifice can read them through normal means. It is yet unclear whether or not these blocks would be considered abnormal. To our knowledge, there are no accumulated statistics on the frequency of zeroed or random blocks after TRIM, and we will attempt to gain a better understanding of this issue.

9 Performance Considerations

First and foremost, Artifice is not intended to be a high performance system; its goal is the protection of users in hostile environments. That said, adequate performance is essential to the real-world use of a deniable storage system. The largest overhead will be the additional processing erasure codes require and the write amplification. There are many traditional ways of speeding up access to files that also apply to the Artifice file system. Despite these methods, reading blocks from seemingly random 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. Excess reads will therefore present less of an issue. Contrarily, using erasure codes to generate the carrier blocks will inevitably impose excess writes and CPU overhead. Traditional buffering techniques can be used to mitigate these delays, though care must be taken to avoid correlation of the carrier blocks and their associated data blocks. Simple methods applied in traditional file systems, such as contiguous allocation, are not applicable as they introduce correlations that would make Artifice insecure. Technological advances

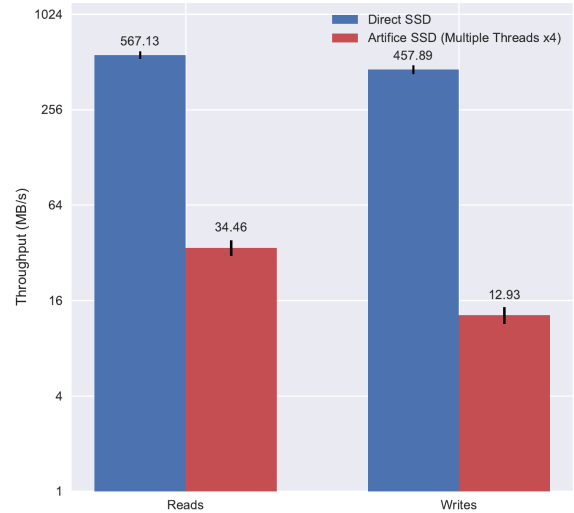

Figure 4: Read and write performance of the Artifice prototype block device versus raw disk.

such as SSDs and on-bus non-volatile memories (NVM) make contiguity less important. As seen in figure 4 the current Artifice prototype with a rudimentary disk scheduling sheme and configured to write four carrier blocks per data block on a commodity SSD provides performance on par with USB 2.0 flash drives [21]. This performance is sufficient for most tasks including compressed 1080p video playback and is a significant improvement over recent ORAM based systems [3, 5].

10 Conclusion

Artifice will offer the first operationally secure, tunable, and self repairing deniable storage system. Unlike previous systems, Artifice provides a means to both deny the existence of data but also the Artifice software itself. Artifice is designed to be used in a hostile environment and provides functional avenues for defeating a multiple snapshot attack. Through its use of external entropy and erasure codes, it protects data through combinatoric security without the use of encryption. These features offer those in harm's way a method to not only protect themselves, but the free flow of information in restricted environments. Those in need of a deniable storage system, aid workers, democracy advocates, or journalists, will be able to entrust their lives to a usable system like Artifice.

Acknowledgments

We would like to thank Thomas Schwarz for assistance with the survivability calculations and our paper shepherd Rishab Nithyanand. 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] Ross Anderson, Roger Needham, and Adi Shamir. The Steganographic File System. In David Aucsmith, editor, *International Workshop on Information Hiding*, pages 73–82, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[2] Julian Assange, Ralf P. Weinmann, and Suelette Dreyfus. Rubberhose. https://web.archive.org/web/20100915130330/http://iq.org/~proff/rubberhose.org/.

[3] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward Robust Hidden Volumes Using Write-Only Oblivious RAM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*, pages 203–214, New York, NY, USA, 2014. ACM.

[4] M. Broz and V. Matyás. The TrueCrypt On-Disk Format–An Independent View. *IEEE Security Privacy*, 12(3):74–77, May 2014.

[5] Anrin Chakraborti, Chen Chen, and Radu Sion. DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries. *Computing Research Repository (CoRR)*, abs/1706.10276, 2017.

[6] Alexei Czeskis, David J. St. Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce 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)*, pages 7:1–7:7, Berkeley, CA, USA, 2008. USENIX Association.

[7] Jake Edge. Thwarting the Evil Maid. *lwn.net*, 2015.

[8] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996.

[9] Jim Gray. Empirical Measurements of Disk Failure Rates and Error Rates. Technical report, December 2005.

[10] Yuri Gubanov and Oleg Afonin. White Paper: SSD Forensics 2014: Recovering Evidence from SSD drivers: Understanding TRIM, Garbage Collection, and Exclusions. Technical report, Belkasoft, 2014.

[11] A. Kiayias and M. Yung. Cryptographic Hardness Based on the Decoding of Reed–Solomon Codes. *IEEE Transactions on Information Theory*, 54(6):2752–2769, June 2008.

[12] Andrew D McDonald and Markus G Kuhn. StegFS: A steganographic file system for Linux. In *International Workshop on Information Hiding*, pages 463–477. Springer, 1999.

[13] Mounir Iddrassi. Veracrypt. https://www.veracrypt.fr/en/Home.html.

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

[15] H. Pang, K. Tan, and X. Zhou. StegFS: a steganographic file system. In *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, pages 657–667, March 2003.

[16] Timothy Peters, Mark A. Gondree, and Zachary N. J. Peterson. DEFY: A Deniable, Encrypted File System for Log-Structured Storage. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[17] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[18] Sarah Dean. FreeOTFE. https://web.archive.org/web/20130305192701/http://freeotfe.org/docs/Main/version_history.htm.

[19] SignalApp. Github: WhisperYAFFS. https://github.com/signalapp/WhisperYAFFS/wiki.

[20] Adam Skillen and Mohammad Mannan. On Implementing Deniable Storage Encryption for Mobile Devices. In *20th Annual Network & Distributed System Security Symposium*, February 2013.

[21] Zachary Throckmorton. USB 3.0 Flash Drive Roundup. *Anandtech*, 2011.

[22] Truecrypt Foundation. Truecrypt. http://truecrypt.sourceforge.net/.

[23] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, Tampa, FL, November 2006. ACM.

[24] Wikipedia contributors. Key disclosure law — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 14-May-2019].

[25] Xuan Zhou, HweeHwa Pang, and Kian Tan. Hiding Data Accesses in Steganographic File System. In *Proceedings 20th International Conference on Data Engineering*, pages 572–583, April 2004.

[26] Aviad Zuck, Udi Shriki, Donald E. Porter, and Dan Tsafrir. Preserving Hidden Data with an Ever-Changing Disk. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS '17)*, pages 50–55, New York, NY, USA, 2017. ACM.