Preserving Hidden Data with an Ever-Changing Disk

Aviad Zuck
Technion – Israel Institute of Technology
aviadzuc@cs.technion.ac.il

Udi Shriki
Technion – Israel Institute of Technology
ujshriki@gmail.com

Donald E. Porter
The University of North Carolina at Chapel Hill
porter@cs.unc.edu

Dan Tsafrir
Technion – Israel Institute of Technology
dan@cs.technion.ac.il

ABSTRACT

This paper presents a storage system that can hide the presence of hidden data alongside a larger volume of public data. Encryption allows a user to hide the contents of data, but not the fact that sensitive data is present. Under duress, the owner of high-value data can be coerced by a powerful adversary to disclose decryption keys. Thus, private users and corporations have an interest in hiding the very presence of some sensitive data, alongside a larger body of less sensitive data (e.g., the operating system and other benign files); this property is called *plausible deniability*. Existing plausible deniability systems do not fulfill all of the following requirements: (1) resistance to multiple snapshot attacks where an attacker compares the state of the device over time; (2) ensuring that hidden data won't be destroyed when the public volume is modified by a user unaware of the hidden data; and (3) disguising writes to secret data as normal system operations on public data.

We explain why existing solutions do not meet all these requirements and present the Ever-Changing Disk (ECD), a generic scheme for plausible deniability storage systems that meets all of these requirements. An ECD stores hidden data inside a large volume of pseudorandom data. Portions of this volume are periodically migrated in a log-structured manner. Hidden writes can then be interchanged with normal firmware operations. The expected access patterns and time until hidden data is overwritten are completely predictable, and insensitive to whether data is hidden. Users control the rate of internal data migration (R), trading write bandwidth to hidden data for longevity of the hidden data. For a typical 2TB disk and setting of R, a user preserves hidden data by entering her secret key every few days or weeks.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Database and storage security**;

HotOS '17, May 08-10, 2017, Whistler, BC, Canada

ACM ISBN 978-1-4503-5068-6/17/05…$15.00
https://doi.org/10.1145/3102980.3102989

ACM Reference format:
Aviad Zuck, Udi Shriki, Donald E. Porter, and Dan Tsafrir. 2017. Preserving Hidden Data with an Ever-Changing Disk. In *Proceedings of HotOS '17, Whistler, BC, Canada, May 08-10, 2017,* 6 pages.
https://doi.org/10.1145/3102980.3102989

1 INTRODUCTION

Preventing unwanted access to sensitive information is a significant concern for many users. Whether it is users wishing to protect private data on their mobile device, political activists under an oppressive regime, or corporations wishing to protect their employees' devices from industrial espionage, the ability to protect data on storage devices—even if the devices fall into the wrong hands—is a timely concern. One possible solution is encrypting data using a key known only to the storing user [8, 10, 16]. However, recently it has been demonstrated that potent adversaries are capable of legally coercing users into divulging the key or password to their protected data [1, 14, 17].

A particularly prevalent limitation of using encryption alone is that users cannot hide the *presence* of the high-value data. Take, for instance, a human rights activist documenting various human rights infringements in a country ruled by an oppressive regime. Before the activist crosses the border, she could encrypt her hard drive using a tool such as BitLocker [8]. However, the fact that encrypted data is present on the hard drive will be apparent upon inspection at the border. An intelligence officer can then detain the activist until either the laptop's encryption is hacked, or even coerce the activist until the decryption key or password is surrendered.

In the above example, the activist wants to hide the presence of some data on the device, even if she is coerced into decrypting a primary volume with less sensitive data, such as a music collection or benign personal photos. This property is called *plausible deniability*.

A plausible deniability solution (PDS) should satisfy three requirements, necessary to make it more secure, as well as practical and easy to use. In order to illustrate these points, consider a simple PDS that uses free space to store hidden data [15, 20]. The first requirement is that the overall layout of the disk should appear innocuous and similar to a disk without hidden data. In our example, hidden data should be encrypted and all free blocks should be initialized with random data with a comparable bit distribution to the ciphertext of hidden data. The second requirement is resistance to multiple snapshot attacks, where a potent adversary compares the state of the device over time in order to detect telltale signs of data hiding. In the prior example, a border agent may take a snapshot of the

disk on entry to the country and upon exit notice unexplained changes to the free space on the device. In a multiple-snapshot-resistant PDS, the adversary can collect an arbitrary number of disk snapshots over time, and not discern the presence of hidden data. The third requirement is preserving the integrity of hidden data stored alongside publicly-visible data even when the user accesses only public data. In the simple example of storing hidden data in free space, if the hidden volume is not in use, the PDS should either avoid allocating this space or migrate the hidden data to another location. The underlying challenge for not destroying hidden data is that the metadata tracking the placement of hidden data needs to be hidden when not in use, lest the presence of hidden data be revealed. In pursuit of this goal, systems generally must expose this risk of hidden data loss to users in some fashion. Some require users to reason about the probability that a public write will destroy data [3] or to try to avoid potentially damaging block allocations [15]. Thus, a subordinate goal is to expose a simple and intuitive risk parameter to users.

Many existing plausible deniability systems do not fully meet the aforementioned requirements (for example, Mobilflage [20], TrueCrypt [15], and StegFS [3] are not resistant to a multiple snapshot attack and . Additionally, all three systems allow hidden data to be overwritten when the system is in public mode. Other systems, such as HIVE [4] and DEFY [18], are resistant to multiple snapshot attacks but do not sufficiently control hidden data loss in public mode, since block allocation for public data writes may unknowingly mistake blocks containing hidden data for free blocks.

In this work we introduce the *Ever-Changing Disk* (ECD), a new design for managing a PDS based on pseudorandom data. Our design allows users to configurably control and meet the aforementioned requirements without compromising the security of hidden data. Like prior PDSes, an ECD also hides data within a large volume of pseudorandom data. However, ECD differs in that hidden and pseudorandom data are not static even in periods when no new data (public or hidden) is written. Instead, hidden and pseudorandom data blocks are constantly relocated and modified to different locations in the system. In other designs, any modifications between subsequent snapshots of the device can potentially undermine security. In the ECD, hidden data security does not depend on data access patterns, public data workloads, or careful system management. The user need only reason about a single configuration parameter, R, that trades hidden data bandwidth for lifespan while data is hidden. Instead, these modifications are easily excused as the result of normal device operations.

The ECD design is not ideal. Without the secret key, data relocations will eventually overwrite hidden data. However, in other systems, the integrity of hidden data is either probabilistic, or relies on careful choices by the file system running in public mode. ECD exposes a simple, intuitive parameter whereby the user can dynamically configure the frequency of automatic data relocations (R), which makes trade-offs in throughput, device lifetime, and frequency of entering the secret key. Ease of use is an essential goal overlooked by prior designs.

2 EXISTING PLAUSIBLE DENIABILITY SYSTEMS

Storing hidden data with plausible deniability in larger public volumes is a long-standing problem. Note that we use the term "public volume" to indicate that the *presence* of the volume is public, not necessarily its contents; the contents of the public volume may still be encrypted. We now detail some prominent prior work.

TrueCrypt [15] stores an additional hidden file system in the free space of a primary public file system. The free space is filled with pseudorandom data when the system is initialized. As a result, the distribution of values in the public volume's free space is comparable, whether the space is truly free or storing hidden, encrypted data.

StegFS [3] is a file system that hides encrypted data in a large volume of pseudorandom blocks. The location of hidden file blocks is determined by the file name. Hidden data is replicated to reduce the risk of being overwritten by updates to the public volume.

Mobilflage [20] is a PDS for mobile devices. Mobiflage uses part of the disk to store random data. An additional hidden volume may be stored within this special partition. Plausible deniability stems from the idea that the random data may or may not include a hidden volume.

HIVE [4], and more recently, DataLair [6], provide plausible deniability by coupling I/O accesses with random data writes, which hides the user's true data accesses. When the user wishes to store hidden data, the random data writes are substituted for hidden data writes. This approach hides suspicious writes in the "noise" of ongoing, random writes. Hidden data is located using a special security construct combined with other in-memory and on-disk mapping data structures.

Finally, DEFY [18] is a log-structured file system for flash that provides plausible deniability. Each data chunk in the public volume is encrypted with a chunk-specific key. The key for each chunk is discarded when it is replaced by a newer chunk. Hidden data is written to a segment that appears in the log as an old, obviated chunk.

3 THE PROBLEM

Building a secure and efficient system for data hiding with plausible deniability entails several challenges. We identify the following as the most challenging and try to address them in this work. Failure to meet these requirements does not formally disqualify a PDS but makes it significantly less secure and practical for users.

Threat Model. We assume the adversary has complete control of the device for periods of time when the device is out of the physical possession of the user. The adversary is capable of inspecting the physical contents of any persistent storage or DRAM on the device, as well as the firmware. We assume the adversary may be able to inspect the device at multiple points in time, and look for correlation in changes across the snapshots. We assume the adversary does not install malware that continues inspection after the device leaves possession of the user, which could presumably be detected by a trusted boot solution. The adversary can destroy hidden data, say by formatting the device or setting it on fire, but should not be

Plausible Deniability System	*Multiple Snapshot Attack*	*Hidden Data Overwrite Resistance*	*Faking Public Accesses*	*PDS Capability Hidden from Adversary*
TrueCrypt [15]	-	-	-	-
StegFS [3]	-	-	-	-
Mobiflage [20]	-	-	-	-
HIVE [4]	+	-	-	-
DEFY [18]	+	-	-	-
This Work	+	±	+	-

Table 1: Important features of plausibly deniable systems. ECD overwrites hidden data only after a large, known, and configurable period of time.

able to prove that hidden data exists or extract the hidden data except by brute force. Finally, we assume the system being used cannot always maintain a secure network connection. Therefore, users must rely on a local, non-network based storage solution to plausibly deny the existence of hidden data on their device.

Multiple Snapshot Attacks. This attack involves an adversary repeatedly accessing and checkpointing the device state, including data, metadata, and any physical characteristics. The adversary can then compare several snapshots and detect unexplained modifications. An example of such an adversary might be the security officer in a corporate research lab who compares the state of a visitor's smartphone upon entry and exit to make sure she did not store sensitive data during her visit.

Unaware Users Destroying Hidden Data. Plausible deniability systems should be able to operate in "public-only" mode without destroying the hidden data. In many designs, a user's secret key is an input to an algorithm that determines where hidden data is stored. In a situation where the device may be under the control of an adversary, such as when the device is under inspection at a border checkpoint, the user will want to purge the secret key and any metadata pertaining to the hidden volume from memory. This can either be a manual process or a periodic, automatic process, such as to protect against disclosure if the device is misplaced.

When the device is under inspection by an adversary, it should still work properly. For example, if the hidden data is stored in the free space of the public file system, writing new public file data may require allocating seemingly free blocks that effectively contain hidden data. Once the user regains possession of the device, the hidden data should still be present.

Faking System Activity. Hidden data is typically stored alongside public data. To maintain plausible deniability, the system should make updates to hidden data on disk appear to an adversary as if there is a plausible reason for the change, other than updating hidden data. First, hidden data must appear semantically similar to public data. Modern PDSes often encrypt hidden data so that it semantically fits in a large (often pre-existing) volume of pseudorandom data in the system. Second, data layout following the hidden data I/O workload should also appear to be the result of plausible system activity, such as updating or deleting a file in the public volume. Some systems, such as HIVE, require the user to offset writes to hidden data with updates to public data, creating "cover traffic". In such cases, many users may find it difficult at times to easily concoct a public write workload to match the hidden one such that the resulting modified state of the system appears to be the outcome of plausible user behavior (e.g., not writing some random content file). This feature is related to the multiple snapshot attack, as well as to usability; ideally, users should not have to be extremely familiar with system internals in order to reliably fake system activity.

Prior work. Prior work does not address all of these challenges. We summarize the properties of these systems and ECD in Table 1.

Many existing works consider only a weaker threat model where the adversary is only capable of taking a single snapshot of the device being used for storing hidden data [3, 15, 20]. In these systems modifications to hidden data result in unexplained changes to the system's state, which undermines the system's resistance to a multiple snapshot attack. HIVE [4] is resistant to this attack since I/O accesses must be accompanied by additional writes to "k random, distinct hard disk block indices.". DEFY [18] provides deniability following a multiple snapshot attack by attributing "undecryptable blocks to old versions of data or metadata, the frequency of which would rely entirely on the user's usage patterns"

To protect against a multiple snapshot attack, systems often require users to fake system activity. HIVE's design requires users to couple every k random writes with a public data write to provide "cover traffic" for hidden data [4]. It is unclear how convincing the resemblance of the required cover traffic is to system activity, and many users may find it difficult to concoct such plausible activity. DEFY [18] claims to avoid this impediment by attributing modifications to the state of the system to writing and deleting new public data. We note however that typical user activity is not composed of writing and deleting the same data. Thus, at some point these undecryptable blocks and resulting data layout may raise the suspicion of an adversary if they are not accompanied by sufficiently innocuous public write activity.

Most PDSes do not reveal any metadata pertaining to hidden data when operating in "public only" mode. Instead, hidden data is treated as invalid public data [4, 15, 18, 20] to avoid divulging the presence and size of hidden data stored in the system. As a result, hidden data may be overwritten when allocating space for newly-written public data. Hidden data overwrites are partially mitigated in StegFS [3] by replicating hidden blocks.

One open and difficult challenge is that it is hard to hide evidence that a PDS is *installed*. In other words, if a border agent

sees a PDS installed on the system, she may be unmoved by the argument that a purported free or random-content set of blocks really doesn't contain more hidden volumes. Worse, the user can no longer prove that she has revealed all of the volumes under coercion (for simplicity, many systems implement a single hidden volume; in practice, a system may offer multiple hidden volumes, and the more plausible question is whether, after revealing n hidden volumes, there is one more). Failing to meet this challenge is common to all systems, including the ECD, and is very difficult to avoid.

4 THE SOLUTION: AN EVER-CHANGING DISK

In this section, we present the design of an ECD, a new solution for hiding data with plausible deniability. Our design allows users to configurably control and meet the following requirements: (1) resisting multiple snapshot attacks; (2) ensuring hidden data integrity; and (3) disguising hidden data writes as part of normal system operation. The ECD does so without compromising the security of hidden data and ease of use. We envision ECD being implemented primarily in device firmware so that the public volume appears as the normal logical block address (LBA) space, and the capacity of the device is simply reduced by some factor for hidden data. Figure 1 illustrates a simple example of the ECD design.

An ECD partitions the storage space into two parts. The first part contains the public data volume and the second contains the hidden data volume, hereby referred to as V_p and V_h respectively. For simplicity of explanation, we assume that each volume manages 50% of the system's storage space. The firmware manages physical placement, and, when the user enters the secret key, a second volume is visible to the system. When the hidden volume is unmounted, the key is flushed from any memory in ECD, and any related metadata is encrypted. Both volumes appear as contiguous LBA ranges.

V_p is mounted and managed as a normal volume. V_h is treated differently. On initialization, V_h is filled with pseudorandom data. For simplicity, V_h is maintained over a separate, physically-consecutive portion of the storage space. Writes to V_h are handled in a log-structured manner, and V_h is partitioned into N segments.

ECD maintains the invariant that all live data is in the active (S_i) or most-recently-active (S_{i-1}) segment. Data is always written to the current active segment (S_i).

What makes an ECD unique is that, in addition to servicing I/O requests, the device periodically copies data from segments containing valid hidden data to the currently active segment S_i. Specifically, data is copied from S_{i-1} to S_i at a predetermined rate, R. Later in this paper we explain the importance of R and its implications for various aspects of ECD operation.

Incoming hidden data write requests are buffered in a FIFO queue in the on-board RAM of the storage device. Writes are issued in 4KB increments at regular intervals. Before appending new data to S_i's head, the ECD waits for a write timeout expiration as dictated by R. For example, for R = 20 MB/s, the ECD will wait for 195us before a new 4KB write is performed in V_h. When that timeout expires, ECD must write data to the next available block in S_i after encrypting it using a special secret key K_{hid} known only to the hiding user.

There are three types of data that the ECD writes: (1) re-encrypted existing hidden data; (2) newly written hidden data; and (3) pseudorandom data. We note that re-encryption of existing hidden data is necessary to hide any correlations between multiple versions of the same content on disk. To avoid re-using the same encryption key ECD uses a composite key K_{hid}, c, where c is a global counter whose value is incremented on every active segment transition. The counter's value can easily be saved and restored using the first page of S_i.

To decide which type of data to write to the current head of the log in S_i, the ECD does the following:

(1) Check in-memory metadata (§5) to determine whether the block at the same offset in S_{i-1} contains valid data. If yes, read the block, re-encrypt, and write.
(2) If the queue contains buffered write requests, pop the queue's head, encrypt it, and write.
(3) Otherwise, write new pseudorandom data.

Once the write is completed, the ECD updates any relevant mapping data structures in RAM (See §5) to indicate the location and validity of newly written data, and invalidates the relevant block in S_{i-1}.

4.1 ECD Features

The ECD avoids multiple snapshot attacks by periodically modifying the state of the device. The timing of hidden data overwrites is not a function of if data is hidden or accessed; rather, a simple, public parameter R controls the bandwidth to the hidden volume area. Therefore, users do not have to concoct innocuous public workloads as cause for hidden writes. Similarly to HIVE [4], the ECD assumes that hidden reads do not modify the state of the system and need not be disguised.

ECD users may occasionally operate the device without entering the secret key, i.e., with only the public volume accessible. Since ECD continues to move data regardless of the device operating mode, operating in public mode may eventually result in hidden data overwrites. R determines T_{pub}, the maximum possible time period that a device can operate in public mode without ECD overwriting hidden data with pseudorandom data. T_{pub} can be summarized in the following equation:

$$T_{pub} = \frac{SIZE(V_h)}{R} \cdot \frac{N-1}{N},$$

which is the time it takes for an ECD to copy/overwrite all N segments, except the last segment of the hidden volume (V_h) that was active before entering public mode. This segment will contain the latest version of each hidden sector. This implies that the user should keep the device in hidden mode for a time period within which the active segment can be scanned, and potentially copied, at least once for every ECD copy cycle. For example, assuming a 2TB SSD configured with R = 3 MB/s and N = 20 segments, T_{pub} = 3.84 days of non-stop operation, or a week of operating half the day. Hidden data may still be eventually, and inadvertently, overwritten in ECD, like in other systems. However, it is straightforward to calculate when data will be overwritten, and enter the key within that interval.

T_{pub} can also be extended by using a smaller R and reducing the throughput of V_h. For example, a user who is about to enter a country controlled by an oppressive regime can simply halve

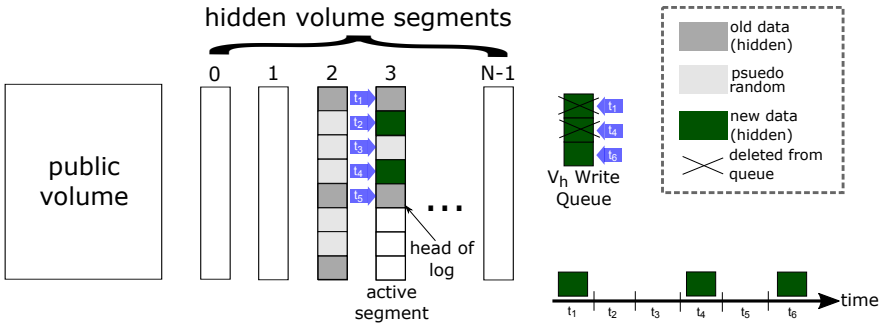

Figure 1: Illustration of an ECD. For simplicity, data is copied in time t=1,2,3, etc. Valid hidden data is copied from segment 2 to segment 3, the current active segment. Blocks that contain pseudorandom data are not copied. Instead, valid hidden data from V_h's write queue is inserted. If the write queue is empty (e.g., at time t=5), new pseudorandom data is written to the head of the log. Future writes (t=6) will be committed whenever possible.

R (thus doubling T_{pub}) and restore it to its original settings after leaving the country.

5 IMPLEMENTATION

We propose to implement ECD as part of an SSD firmware for several reasons. First, flash random access latencies are similar to those of sequential accesses. Second, due to the inherent asymmetry between flash write and erase units [11], SSD firmware usually adopts log-structured designs (not unlike the ECD itself). Third, HDDs have a single mechanical head; concurrently accessing partitioned physical space in HDDs can significantly disrupt performance for the public volume by seeking between the volumes, whereas SSDs can more easily apportion a fraction of the bandwidth to the hidden volume. We do note that unlike flash, lifespan of HDDs does not necessarily depend on the amount of writes issued to the device. Therefore, HDDs may better tolerate the additional writes of the ECD. Emerging SMR technology [12] is inherently log-structured, and may also serve as a useful substrate for the ECD in future work.

Another advantage of using an SSD to implement the ECD is that flash pages inherently contain an out-of-bound spare area, in addition to the page data area. It is standard practice for SSDs to use this spare area to store metadata that helps recover and reconstruct volatile firmware metadata structures. In an ECD, the additional hidden volume metadata mostly includes standard SSD data structures such as a logical-to-physical mapping table [2] and the active segment's page validity bitmap to improve the performance of segment copying [5, 9]. In the ECD this metadata will also be encrypted and stored on flash using the hidden volume key.

6 IMPLICATIONS AND COST

ECD design and underlying storage media have several important implications. In this section we analyze some of these implications and various tradeoffs resulting from the ECD's unorthodox design.

6.1 I/O Performance

Write requests to ECD's hidden volume are buffered in a queue, and are serviced at the next scheduled internal data copying timeout. Therefore, ECD's maximum hidden write I/O throughput is known and determined by R. Read requests for hidden data that is still queued on the disk may be stalled, depending on the use of disk I/O scheduling optimizations such as NCQ. Even so, read throughput is at worst constrained by R, and at best by the device's baseline (public) read throughput.

Public data I/O throughput remains high but is also partially affected by ECD operations. The higher R is, so are the resulting overhead and amortized delay for public I/O operations. The tradeoff is that R is configurable; users control its effect and can modify it according to their needs. For example, consider a user with a modern consumer-grade SSD capable of performing random write I/O at a maximum throughput of 200 MB/s. For R = 20 MB/s, the added overhead slows public I/O throughput by ~10%. To improve public data throughput the user can slow R down by half to R = 10 MB/s, which will increase public I/O throughput by 5%. Users should take into account that such adjustments may raise the suspicion of attackers capable of snapshotting the device before and after the user adjusts R.

The ECD uses 4KB write increments. It is possible to optimize the ECD to access the underlying device in larger sequential batches, at the cost of increasing latency and buffering uncommitted writes in a large non-volatile memory. In future work, we will explore trade-offs between hidden volume efficiency and write latency.

6.2 Capacity

Public data capacity in an ECD is directly derived from the user-configured partitioning of the storage space into public and hidden portions. Our design makes a trade-off between the space capacity of the hidden volume, and the time at which hidden data will be destroyed (T_{pub}). In the current ECD design, users can only write one segment's worth of hidden data; other segments can be overwritten in "public" mode without losing data. Thus, V_h/N is the maximum hidden data capacity. For a 2TB SSD where V_h occupies 50% of the disk and N = 20, the hidden data capacity is 50GB. In principle, one could increase capacity of the hidden volume by either increasing segment size or keeping live data on more than one segment, but lowering the time until hidden data would be lost.

6.3 Endurance and Persistence

As previously mentioned, for performance reasons, the ECD is optimally implemented using an SSD with limited lifetime. Even after employing unified wear-leveling policies [2, 11] over the blocks of V_h and V_p, the SSD's lifetime may be depleted earlier than expected due to the ECD's continuous writes. Yet, ECD users can safely use an SSD to hide data for sufficiently long periods of time. For example, consider an ECD implemented over a 2TB SSD that is constantly powered. We further assume that the SSD can endure a conservative total of 1000 complete device writes [13, 19]. In this setup, even with a fast $R = 25$ MB/s, we can expect that the entire hidden volume will be dynamically relocated and copied *once a day over the course of three years* until it reaches the vendor's guaranteed lifetime limit. This rate allows ECD users to avoid wearing out the device, while having little effect on the performance of the public volume V_p. We note that if the device is powered on only for part of the day, its lifetime will increase proportionally.

6.4 Energy

SSDs operate a large array of multiple flash packages in parallel. Therefore, the power consumption of SSDs mostly depends on that of the underlying flash chips. SSDs save power by entering idle state when the entire flash memory part is inactive [7]. An ECD would employ only a few chips concurrently at a relatively slow rate, preventing the SSD from entering idle mode. As a result, the energy consumption of an ECD may be up to 5x higher than in an SSD.

To conserve power, users can decrease R, allowing for longer idle periods. ECD should attempt to schedule public volume maintenance operations [11, 21] together with hidden volume operations.

7 CONCLUSIONS

Users' privacy concerns and commercial needs require systems capable of storing sensitive data in a hidden volume so that the user can plausibly deny the existence of this volume and that its related blocks contain hidden data. Existing solutions do not fully meet many important requirements. Therefore, better solutions must be devised, both in terms of security and usability. We consider this work a step in this direction and plan to implement it as part of an SSD firmware to verify our design and its performance characteristics.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments on earlier drafts of the work. This research was supported by Grant 2014621 from the United States-Israel Binational Science Foundation (BSF), by Grant CNS-1526707 from the United States National Science Foundation (NSF), VMware, and the Zeff fellowship.

REFERENCES

[1] 2010. Youth jailed for not handing over encryption password. The Register http://www.theregister.co.uk/2010/10/06/jail_password_ripa/. (2010).
[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference (ATC)*.
[3] Ross Anderson, Roger Needham, and Adi Shamir. 1998. The Steganographic File System. In *Information Hiding*. Lecture Notes in Computer Science, Vol. 1525. Springer Berlin Heidelberg, 73–82.
[4] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. 2014. Toward Robust Hidden Volumes Using Write-Only Oblivious RAM. In *Proceedings of the SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. https://doi.org/10.1145/2660267.2660313
[5] Evgeny Budilovsky, Sivan Toledo, and Aviad Zuck. 2011. Prototyping a High-performance Low-cost Solid-state Disk. In *Proceedings of the Annual International Conference on Systems and Storage (SYSTOR)*.
[6] Anrin Chakraborti, Chen Chen, and Radu Sion. 2017. DataLair: Efficient Block Storage with Plausible Deniability against Multi-Snapshot Adversaries. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*.
[7] Seokhei Cho, Changhyun Park, Youjip Won, Sooyong Kang, Jaehyuk Cha, Sungroh Yoon, and Jongmoo Choi. 2015. Design Tradeoffs of SSDs: From Energy Consumption's Perspective. *Transactions on Storage* 11, 2, Article 8 (2015), 24 pages. https://doi.org/10.1145/2644818
[8] Microsoft Corporation. 2009. Windows BitLocker drive encryption frequently asked auestions. http://technet.microsoft.com/en-us/library/cc766200%28WS.10%29.aspx. (2009).
[9] Niv Dayan and Philippe Bonnet. 2015. Garbage Collection Techniques for Flash-Resident Page-Mapping FTLs. *CoRR* abs/1504.01666 (2015). http://arxiv.org/abs/1504.01666
[10] Disk 2017. Disk encryption in Arch Linux. (2017). http://wiki.archlinux.org/index.php/disk_encryption.
[11] Eran Gal and Sivan Toledo. 2005. Algorithms and Data Structures for Flash Memories. *Computing Surveys* 37, 2 (June 2005), 138–163. https://doi.org/10.1145/1089733.1089735
[12] Garth Gibson and Milo Polte. 2009. Directions for shingled-write and twodimensional magnetic recording system architectures: Synergies with solid-state disks. *CMU-PDL-09-014* (2009).
[13] Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The Bleak Future of NAND Flash Memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*.
[14] Key 2017. Key disclosure law. Wikipedia http://en.wikipedia.org/wiki/Key_disclosure_law. (2017).
[15] Open 2017. Open Crypto Audit Project. http://opencryptoaudit.org/. (2017).
[16] OS 2015. OS X Mavericks: Encrypt the information on your disk with FileVault. http://support.apple.com/kb/PH13729. (2015).
[17] Password 2012. Password case reframes Fifth Amendment rights in context of digital world. Denver Post http://www.denverpost.com/news/ci_19669803. (2012).
[18] Timothy M Peters. 2014. *DEFY: A Deniable File System for Flash Memory*. Master's thesis. California Polytechnic State University.
[19] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*.
[20] Adam Skillen and Mohammad Mannan. 2013. On Implementing Deniable Storage Encryption for Mobile Devices.. In *The Network and Distributed System Security Symposium (NDSS)*.
[21] Chengen Yang, Hsing-Min Chen, TrevorN. Mudge, and Chaitali Chakrabarti. 2014. Improving the Reliability of MLC NAND Flash Memories Through Adaptive Data Refresh and Error Control Coding. *Journal of Signal Processing Systems* 76, 3 (2014), 225–234. https://doi.org/10.1007/s11265-014-0880-5