sciendo
Proceedings on Privacy Enhancing Technologies ; 2019 (1):153–171

Chen Chen*, Anrin Chakraborti, and Radu Sion

PD-DM: An efficient locality-preserving block device mapper with plausible deniability

Abstract: Encryption protects sensitive data from unauthorized access, yet is not sufficient when users are forced to surrender keys under duress. In contrast, plausible deniability enables users to not only encrypt data but also deny its existence when challenged.
Most existing plausible deniability work (e.g. the successful and unfortunately now-defunct TrueCrypt) tackles "single snapshot" adversaries, and cannot handle the more realistic scenario of adversaries gaining access to a device at multiple time points. Such "multi-snapshot" adversaries can simply observe modifications between snapshots and detect the existence of hidden data.
Existing ideas handling "multi-snapshot" scenarios feature prohibitive overheads when deployed on practically-sized disks. This is mostly due to a lack of data locality inherent in certain standard access-randomization mechanisms, one of the building blocks used to ensure plausible deniability.
In this work, we show that such randomization is not necessary for strong plausible deniability. Instead, it can be replaced by a canonical form that permits most of writes to be done sequentially. This has two key advantages: *1) it reduces the impact of seek due to random accesses; 2) it reduces the overall number of physical blocks that need to be written for each logical write.* As a result, PD-DM increases I/O throughput by orders of magnitude (10–100× in typical setups) over existing work while maintaining strong plausible deniability against multi-snapshot adversaries.
Notably, PD-DM is the first plausible-deniable system getting within reach of the performance of standard encrypted volumes (dm-crypt) for random I/O.

Keywords: Plausible deniability, Storage security

DOI 10.2478/popets-2019-0009
Received 2018-05-31; revised 2018-09-15; accepted 2018-09-16.

***Corresponding Author: Chen Chen:** Stony Brook University, E-mail: chen18@cs.stonybrook.edu
Anrin Chakraborti: Stony Brook University, E-mail: anchakrabort@cs.stonybrook.edu
Radu Sion: Stony Brook University, E-mail: sion@cs.stonybrook.edu

1 Introduction

Although encryption is widely used to protect sensitive data, it alone is usually not enough to serve users' intentions. At the very least, the existence of encryption can draw the attention of adversaries that may have the power to coerce users to reveal keys.

Plausible deniability(PD) aims to tackle this problem. It defines a security property making it possible to claim that "some information is not in possession [of the user] or some transactions have not taken place" [23]. In the context of storage devices, as addressed in this paper, PD refers to the ability of a user to plausibly deny the existence of certain stored data even when an adversary has access to the storage medium.

In practice, PD storage is often essential for survival and sometimes a matter of life and death. In a notable example, the human rights group Network for Human Rights Documentation-Burma (ND-Burma) carried data proving hundreds of thousands of human rights violations out of the country on mobile devices, risking exposure at checkpoints and border crossings [27]. Videographers brought evidence of human rights violations out of Syria by hiding a micro-SD card in body wounds [24], again at high risk of life.

Yet, unfortunately, existing PD storage solutions are either slow or insecure against "multi-snapshot" adversaries that can gain access to the storage multiple times over a longer period. For example, steganographic filesystems [5, 23, 26] (also known as "deniable filesystems"), resist only adversaries who can access storage mediums once ("single-snapshot" adversaries). And although some attempts [26] have been made to handle multi-snapshot adversaries, they are insecure. A strong adversary can straightforwardly infer what data is hidden with a relatively high probability, better than random guessing. On the other hand, while block-level PD solutions [6, 8] handling multi-snapshot adversaries exist, their performance is often simply unacceptable and many orders of magnitude below the performance of non-PD storage.

Nevertheless, most realistic PD adversaries are ultimately multi-snapshot. Crossing a border twice, or having an oppressive government collude with a hotel maid and subsequently a border guard, provides easy and cheap multi-snapshot capabilities to any adversary.

It is obvious that the security of a PD system should not break down completely (under reasonable user behavior) and should be resilient to such realistic externalities (hotel maids, border guards, airline checked luggage etc). Thus, undeniably, *a practical and sound plausible-deniable storage solution crucially needs to handle multi-snapshot accesses.*

In virtually all PD designs, handling multi-snapshot adversaries requires hiding the "access patterns" to hidden data. Otherwise, adversaries could trivially observe implausible modifications between snapshots and infer the existence of hidden data.

Some of the first solutions [8, 31] hide access patterns by rendering them indistinguishable from random. Unfortunately, existing randomization-based mechanisms completely break data locality and naturally impose large numbers of physical writes that are needed to successfully complete one logical write. This results in unacceptable performance, especially for high-latency storage (e.g., rotational disks). For example, HIVE [8] performs 3 orders of magnitude slower than a non-PD disk baseline.

We observe that randomization is not the only way to hide access patterns. Canonical forms that depend only on non-hidden public data (e.g., sequential structuring) suffice and can yield dramatic throughput increases. This is mainly due to two reasons. First, it is well known that sequential structuring can vastly speed up I/O by reducing the impact of random seeks [14]. This has been previously well explored in log-structured filesystems [13, 28]. Second, the predictable nature of such sequential access leads to more efficient searches for free block locations (to write new data) with lower numbers of I/O operations, when compared with existing (mostly randomized access) solutions.

Finally, we note that, by definition, a well-defined multi-snapshot adversary cannot observe the device (including its RAM, etc) between snapshots. This enables additional optimizations for increased throughput.

The result is PD-DM, a new, efficient block-level storage solution that is strongly plausibly deniable against multi-snapshot adversaries, preserves locality and dramatically increases performance over existing work. In typical setups PD-DM is orders of magnitude (10–100×) faster than existing approaches.

2 Storage-centric PD Model

PD can be provided at different layers in a storage system. However, properly designing multi-snapshot PD at the filesystem level is not trivial. As filesystems involve many different types of data and associated metadata, resolving all data and metadata modifications and associated chain reactions in a plausibly deniable manner requires significant and careful consideration. We are not aware of any successful, fully secure design. Moreover, data loss (overwriting of data at a high deniability level by writes to data at a lower deniability level) is likely an unavoidable drawback of filesystem PD ideas.

Instead, the focus of this work is on storage-centric PD for block devices – allowing a user to store files of different confidentiality levels to different logical volumes stored on the same underlying physical device. These volumes are accessed independently by upper-layer applications such as filesystems and databases.

It is important to note that *no existing PD solution hides the fact that a PD system is in place.* So far (and in this paper), the role of a sound PD system has been to hide whether a user stores any hidden files (for filesystems) or uses any hidden volumes (in block level schemes). In other words, it is assumed that adversaries won't punish users for merely using a PD-enabled system, at least as long as the system allows also non-PD purposes, e.g., storing in a public volume.

2.1 PD Framework at Block Level

A PD system at the block level intermediates between a raw storage device and upper logical volumes (Figure 1). It structures accesses to the underlying device while providing upper-level abstractions for multiple independent logical volumes. Logical public volumes V_p^i can be revealed to adversaries, whereas the existence and usage of hidden volumes V_h^j should be hidden from adversaries. The physical device contains N_d blocks of B bytes each, while each logical volume has N_p^i/N_h^j logical data blocks, respectively. The volumes are encrypted with different encryption keys K_p^i and K_h^j (e.g., derived from passwords P_p^i and P_h^j). If coerced, users can provide the public passwords and deny the existence or use of hidden volumes even to multi-snapshot adversaries.

Adversary. We consider a strong computationally bounded "multi-snapshot" adversary able to get complete representations of the device's static state at multiple times of her choosing. Importantly, the adversary cannot see the running state (memory, caches etc) of

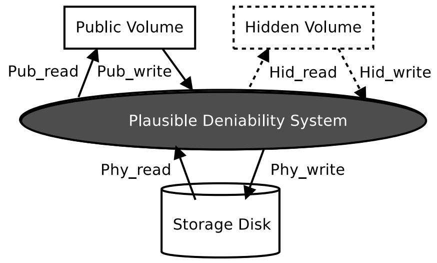


Fig. 1. Overview of a PD system with one public volume and one hidden volume as an example. PD-DM provides virtual volume abstractions. The choice of whether the hidden volume is used will be hidden.

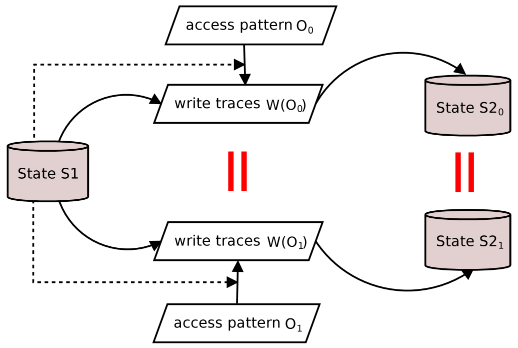


Fig. 2. Consider two access patterns $\mathcal{O}_0$ and $\mathcal{O}_1$ that transform the state of the disk from $S1$ to $S2_0$ and $S2_1$ respectively. Adversaries should not be able to distinguish $S2_0$ from $S2_1$ based on the snapshots. Effectively the adversary should not be able to distinguish $\mathcal{W}(\mathcal{O}_0)$ from $\mathcal{W}(\mathcal{O}_1)$

the PD layer occurring before she takes possession of the device. For example, a border guard in an oppressive government can check a user's portable hard drives every time he crosses the border, but the guard cannot install malwares or otherwise spy on the device.

Access Pattern. An *access* can be either a read or write to a logical volume. $Pub\_read(l_p^i)$, $Pub\_write(l_p^i, d_p^i)$ denote accesses to public volume V_p^i, while $Hid\_read(l_h^j)$, $Hid\_write(l_h^j, d_h^j)$ denote accesses to hidden volume V_h^j. l_p^i and l_h^j are logical block IDs while d_p^i and d_h^j are data items. An *access pattern* $\mathcal{O}$ refers to an ordered sequence of accesses being submitted to logical volumes, e.g., by upper-layer filesystems.

Write Trace. A *write trace* $\mathcal{W}(\mathcal{O})$ refers to a series of actual modifications to the physical block device resulting from a logical access pattern $\mathcal{O}$. Let $Phy\_write(\alpha, d)$ denote an atomic physical device block write operation of data d at location α (physical block ID). Then a *write trace* $\mathcal{W}(\mathcal{O})$ can be represented as an ordered sequence of physical block writes $Phy\_write(\alpha, d)$.

A multi-snapshot adversary may get information about a write trace by observing differences in the "before" and "after" device states. A PD system is designed to prevent the adversary from inferring that the write trace corresponds to hidden volume accesses.

Not unlike existing work, PD-DM achieves this by providing plausible explanations for accesses to hidden volumes using accesses to public volumes. Figure 2 illustrates. Consider two access patterns $\mathcal{O}_0$ and $\mathcal{O}_1$. $\mathcal{O}_1$ contains hidden accesses while $\mathcal{O}_0$ does not. The two corresponding write traces bring the disk from input state $S1$ to output state $S2_0$ and $S2_1$ according to which of $\mathcal{O}_0$ and $\mathcal{O}_1$ has been executed. The idea then is to empower a user to plausibly deny the execution of hidden accesses by arguing that any observed device state changes between snapshots would have been indistinguishably induced by $\mathcal{O}_1$ or $\mathcal{O}_0$. Effectively, the corresponding write traces $\mathcal{W}(\mathcal{O}_0)$ and $\mathcal{W}(\mathcal{O}_1)$ should be indistinguishable.

One key observation is that *only write accesses are of concern here since read accesses can be designed to have no effect on write traces.* This also illustrates why it is important to guarantee PD *at block level* – the above may not necessarily hold for filesystems which may update "last-accessed" and similar fields even on reads.

2.2 Security Game

An intuitive explanation of PD at block level is illustrated in Figure 2. Towards more formal insights, we turn to existing work and observe that security requirements are sometimes quite divergent across different works. For example, Blass et al. [8] introduced a hidden volume encryption (HIVE) game. It outlines two main concerns: frequency of snapshots and "degree of hiding". "Restricted", "opportunistic" and "plausible" hiding are defined along with three kinds of adversaries, "arbitrary", "on-event" and "one-time". Chakraborti et al. [6] proposed the PD-CPA game (Appendix A.1) targeting a more powerful "arbitrary" adversary, further excluding reliance on any specific device property.

We discover that both models feature a shared structural backbone coupled with certain adversary-specific features. The games are defined between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ on a device $\mathcal{D}$ with a "meta game" structure.

Before presenting the meta game, we first introduce $IsExecutable(\mathcal{O})$ – a *system-specific predicate* that returns $True$ if and only if the access patterns $\mathcal{O}$ can be "fully executed". It is hard to have a universal definition of $IsExecutable(\mathcal{O})$ for all the PD solutions. For example, in HIVE, $IsExecutable(\mathcal{O})$ models whether there exists one corresponding read after each access to a hidden volume. In PD-CPA, $IsExecutable(\mathcal{O})$ models whether the ratio between the number of hidden writes and the number of public writes is upper bounded by a parameter ϕ. Then, the meta game is defined as follows:

1. Let n_p and n_h denote the number of public and hidden volumes respectively.
2. Challenger $\mathcal{C}$ chooses encryption keys $\{K_p^i\}_{i\in[1,n_p]}$ and $\{K_h^j\}_{j\in[1,n_h]}$ using security parameter s.
3. $\mathcal{C}$ creates logical volumes $\{V_p^i\}_{i\in[1,n_p]}$ and $\{V_h^j\}_{j\in[1,n_h]}$ in $\mathcal{D}$, encrypted with keys K_p^i and K_h^j respectively.
4. $\mathcal{C}$ selects a random bit b.
5. $\mathcal{C}$ returns $\{K_p^i\}_{i\in[1,n_p]}$ to $\mathcal{A}$.
6. The adversary $\mathcal{A}$ and the challenger $\mathcal{C}$ then engage in a polynomial number of rounds in which:
   (a) $\mathcal{A}$ selects a set of access patterns $\{\mathcal{O}_p^i\}_{i\in[1,n_p]}$ and $\{\mathcal{O}_h^j\}_{j\in[1,n_h]}$ with the following restrictions:
      i. $\mathcal{O}_p^i$ includes arbitrary writes to V_p^i.
      ii. $\mathcal{O}_h^j$ includes writes to V_h^j, subject to restriction 6(a)iv below.
      iii. $\mathcal{O}_0$ includes all the accesses in $\{\mathcal{O}_p^i\}_{i\in[1,n_p]}$
      iv. There exists $\mathcal{O}_1$, a sequence composed (e.g., by adversary $\mathcal{A}$) of all accesses in $\mathcal{O}_0$ and $\{\mathcal{O}_h^j\}_{j\in[1,n_h]}$, such that $IsExecutable(\mathcal{O}_1) = True$
   (b) $\mathcal{C}$ executes $\mathcal{O}_b$ on $\mathcal{D}$.
   (c) $\mathcal{C}$ sends a snapshot of the resulting device to $\mathcal{A}$.
7. $\mathcal{A}$ outputs b', $\mathcal{A}$'s guess for b.

Definition 1. *A storage mechanism is plausibly-deniable if it ensures $\mathcal{A}$ cannot win the security game with a non-negligible advantage over random guessing. Or, more formally, there exists a function $\epsilon(s)$, negligible in security parameter s, such that $Pr[b' = b] \leq 1/2 + \epsilon(s)$.*

The restrictions on access pattern choices (in step 6a) are mandatory in the game to prevent "trivial" impossibility. In the absence of access patterns restrictions, it may be impossible to find a solution. E.g., an adversary may choose different public accesses for $\mathcal{O}_0$ and $\mathcal{O}_1$, so that she can distinguish access patterns by public accesses only and trivially win the game. The restrictions 6(a)i - 6(a)ii ensure that the public writes chosen to cover the hidden writes are not a function of the hidden writes, lest information about the existence of the hidden volume is leaked by the public write choices (which are unprotected from adversaries).

The $IsExecutable(\mathcal{O}_1)$ in the restriction 6(a)iv is related to the systems' ability to combine public and hidden accesses to ensure PD. It prevents adversaries winning the game by trivially choosing access patterns with hidden accesses that cannot be hidden by a given system. In existing work, $IsExecutable(\mathcal{O}_1)$ depends only on the ratio of hidden to public operations (either public read or public write). This yields actual practical solutions, but also limits the solution space to ORAM-based mechanisms. Yet, this is not necessary if we consider the fact that adversaries in reality have much less freedom in choosing access patterns.

Specifically, requiring adversaries to choose no matter how many public accesses that are needed to cover those hidden accesses in the access pattern is quite reasonable. The number of public accesses chosen could be related to not only the number of hidden accesses but also the current state of hidden data on disk. In reality, this corresponds to a multi-snapshot adversary who gains access to storage devices at different points in time, but does not monitor the system running. As a result, the adversary has limited knowledge about what has happened in the meantime in the hidden volumes (e.g. hard disk snooping [18, 22]).

In this case, $IsExecutable()$ can still be easily fullfilled by access patterns coming from many realistic upper-layer workloads. For example, if PD-DM runs underneath log-structure filesystems on both volumes, $IsExecutable()$ becomes exactly the same as that in PD-CPA. Further, once the public and hidden volume workloads sufficiently correlate, the PD-DM $IsExecutable()$ is True with high probability (see Section 4.2).

Timing. Although a system log may record the public writes which cover the hidden accesses in PD-DM, it is important to note that *timing information is assumed to not provide adversaries any significant advantage.*

3 System Design

The key idea in PD-DM is to ensure a canonical/sequential physical device write trace, notwithstanding the logical layer workload. *All* writes to the physical device are located at sequentially increasing physical addresses, similar to append operations in log-structure filesystems.

This canonical nature eliminates the ability of an adversary to infer logical layer access patterns, and the possible existence of a hidden volume stored in apparently "free" space – e.g., by noticing that this "free" space is being written to in between observed snapshots.

More importantly, it brings two significant performance benefits. Firstly, it reduces the impact of disk seeks. This is very important as the seek time dominates the disk access time [20]. For example, the average seek time for modern drives today may range from 8 to 10 ms [21], while the average access time is around 14 ms [20]. Secondly, as we will see, a block that is chosen to be written in PD-DM is more likely to contain invalid data compared to a randomly-selected block. This reduces the average number of physical blocks that need to be accessed to serve one logical write request.

System layout. Figure 3 illustrates the disk layout of PD-DM and how logical volumes are mapped to the underlying physical device. For simplicity, we consider one public volume and one hidden volume in PD-DM.[1] As shown later, it is easy to add more volumes. In general, the disk is divided into two segments: a data segment and a mapping segment. Both public and hidden data is mapped to the data segment, more specifically, the *Data* area (in Figure 3). Several data structures (PPM, PBM, HM_ROOT in Figure 3) are stored in the mapping segment, to maintain the logical volume to physical device mapping obliviously and provide physical block status information. As will be detailed later, the PPM and PBM are associated with the public volume, and HM_ROOT with the hidden volume.

For better performance, recently used mapping-related blocks are cached in memory using assigned memory pools.[2]

Log-structure. The *Data* area is written sequentially as a log and the P_{head} records its head. An atomic write to the *Data* area writes not only public data but also either hidden data as well as its related mapping blocks or some dummy data to several successive blocks pointed by P_{head}.[3] The pointer P_{head} indicates the block *after* the last written physical block and *wraps around* to point again to the first physical block once the last block of the *Data* area is written.

Underlying Atomic Operation: PD_write**.** As mentioned, PD-DM writes to the *Data* area sequentially with an atomic operation. Thus, we define the atomic operation as PD_write (see Section 3.3 for details). The content written by a PD_write operation can be adjusted by users. We assume that one block of public

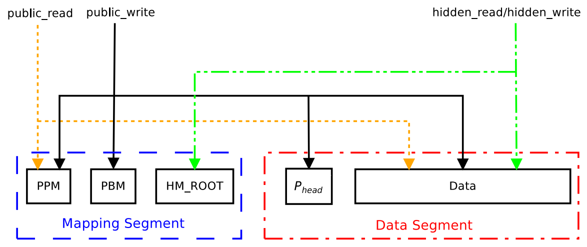


Fig. 3. PD-DM layout. The left-most dotted rectangle (blue) highlights the data structures used in mapping logical volume blocks to physical locations: PPM, PBM and HM_ROOT. The right-most dotted rectangle (red) highlights the data storage segment including "P_{head}" and a set of consecutive blocks on the disk. This segment stores all data from both public and hidden volumes, as well as parts of the hidden map.

data and one block of hidden data can be included in one PD_write operation without loss of generality.

PD-DM receives upper-layer I/O requests and translates them into physical I/O to underlying devices. For the sake of simplicity, we first discuss write requests and later read requests. The remaining part of this section is structured as follows: Section 3.1 describes how public and hidden data is arranged in the *Data* area under the assumption that mappings are in memory. Section 3.2 details disk layout of mapping data and its associated plausibly deniable management. Finally, Section 3.3 outlines the entire end-to-end design of PD-DM.

3.1 Basic Design

In this section, we simplify the discussion by considering certain part of the system as black boxes. Specifically, to handle logical write requests ($Pub\_write(l_p, d_p)$ and $Hid\_write(l_h, d_h)$), PD-DM needs to arrange the received data d_p/d_h on the physical device and record the corresponding mappings from logical volumes to the physical device for future reference. For now, we will consider the maps as black boxes managing the logical-physical mappings, as well as the status (free, valid, invalid, etc.) of each physical block on the device. The map blackbox internals will be provided in Sections 3.2. The black boxes support the following operations, where α is the ID of the physical location storing the logical block (l_p or l_h) and S denotes the "status" of a physical block: free, valid, or invalid:

- $\alpha = Get\_Pub\_map(l_p)$, $Set\_Pub\_map(l_p, \alpha)$
- $\alpha = Get\_Hid\_map(l_h)$, $Set\_Hid\_map(l_h, \alpha)$
- $S = Get\_status(\alpha)$, $Set\_status(\alpha, S)$

With these auxiliary functions, incoming pairs of $Pub\_write(l_p, d_p)$ and $Hid\_write(l_h, d_h)$ requests are

1 V_p and V_h are used for the two volumes instead of V_p^i and V_h^j, and the index i or j is also omitted in all the other notations defined in Section 2 for simplicity.

2 Not to be confused with chip level caches (PLB) used in some ORAM-based designs [15].

3 This is a key point and reduces the impact of disk seeks by an order of magnitude.

handled straightforwardly as described in Algorithm 1. As we assumed that one block of public data and one block of hidden data are written together in this paper, two blocks in the *Data* area are written. Specifically, public data d_p and hidden data d_h are encrypted with K_p and K_h, respectively, and written to the two successive blocks at physical locations P_{head} and $P_{head}+1$. Initially, P_{head} is set to 0 and PD-DM is ready to receive logical write requests.

In a public-volume-only setting, random data is substituted for the encrypted hidden data. In Algorithm 1, $d_h = Null$ signifies the lack of hidden write requests from the upper layers.

Algorithm 1 Pub_write + Hid_write – Simple version

```
Input: $Pub\_write(l_p, d_p)$, $Hid\_write(l_h, d_h)$, $P_{head}$
 1: if $d_h$ is not $Null$ then
 2:     $Phy\_write(P_{head}, d_p)$
 3:     $Phy\_write(P_{head}+1, d_h)$
 4:     $Set\_Pub\_map(l_p, P_{head})$
 5:     $Set\_Hid\_map(l_h, P_{head}+1)$
 6: else            ▷ a public volume only, no hidden write request
 7:     Randomize data $dummy$
 8:     $Phy\_write(P_{head}, d_p)$
 9:     $Phy\_write(P_{head}+1, dummy)$
10:     $Set\_Pub\_map(l_p, P_{head})$
11: end if
12: $P_{head} \leftarrow P_{head}+2$
```

Wrap around. As described, PD-DM always aims to write to the “oldest” (least recently written) block in the *Data* area. To achieve this, it needs to mitigate the possibility that, after a wrap-around, valid data is in fact stored in that location and cannot be overwritten. For simplicity, consider the size of the *Data* area being a multiple of 2 and aligned by public-hidden data pairs.

We denote the existing public-hidden data pair pointed to by P_{head}, $d_p^{existing}$ and $d_h^{existing}$. Their status can be either valid or invalid. Invalid data may be either the result of a previous dummy block write or the case of data being an “old” version of a logical block.

Now, the key to realize here is that *preserving any of the valid existing data can be done by including it into the current input of the ongoing round of writing.* Table 1 illustrates how function $Get\_new\_data\_pair(d_p, d_h, P_{head})$ generates the data pair (d_p^{new}, d_h^{new}) written in the ongoing round in PD-DM. The data pair is generated based on the status of existing data and d_h. In brief, the priority for constructing the new pair of data is $d_p^{existing}/d_h^{existing} > d_p/d_h > dummy$. Then, logical requests *Pub_write* and *Hid_write* are handled with Algorithm 2.

Note that the while loop in this algorithm may end without setting d_h to $Null$, i.e., the hidden write request is not completely satisfied within one public write. In this case, more public writes are required to complete the same hidden write. After a sufficient number – a constant, as will be shown later – of rounds, the hidden write request can be guaranteed to go through.

Algorithm 2 Pub_write + Hid_write – Considering wrap-around

```
Input: $Pub\_write(l_p, d_p)$, $Hid\_write(l_h, d_h)$, $P_{head}$
 1: while $d_p$ is not $Null$ do
      // The new pair of data is constructed based on Table 1
 2:     $(d_p^{new}, d_h^{new}) = Get\_new\_data\_pair(d_p, d_h, P_{head})$
 3:     $Phy\_write(P_{head}, d_p^{new})$
 4:     $Phy\_write(P_{head}+1, d_h^{new})$
 5:     if $d_p^{new} == d_p$ then
 6:         $Set\_Pub\_map(l_p, P_{head})$, $d_p = Null$
 7:     end if
 8:     if $d_h^{new} == d_h$ then
 9:         $Set\_Hid\_map(l_h, P_{head}+1)$, $d_h = Null$
10:     end if
11:     $P_{head} \leftarrow P_{head}+2$
12: end while
```

Buffering and seek-optimized writes. Notwithstanding the logical data layout, PD-DM always writes physical data blocks sequentially starting from the current value of P_{head} until enough blocks with invalid data have been found to store the current payload (Algorithm 2). Depending on the underlying stored data, written contents are either re-encrypted valid existing data or newly written payload data (Table 1). For example, consider the following case where the pattern of existing valid (V) and invalid (I) data is $[VIIVVI\cdots]$. Instead of seeking to position 2 to write new data there, PD-DM writes both position 1 and 2 with re-encrypted/new data sequentially.

d_h	$d_p^{existing}$	$d_h^{existing}$	d_p^{new}	d_h^{new}
$NULL$	invalid	invalid	d_p	$dummy$
	invalid	valid	d_p	$d_h^{existing}$
	valid	invalid	$d_p^{existing}$	$dummy$
	valid	valid	$d_p^{existing}$	$d_h^{existing}$
not $NULL$	invalid	invalid	d_p	d_h
	invalid	valid	d_p	$d_h^{existing}$
	valid	invalid	$d_p^{existing}$	d_h
	valid	valid	$d_p^{existing}$	$d_h^{existing}$

Table 1. PD-DM constructs the data pair (d_p^{new}, d_h^{new}) that can be written by a PD_write operation according to the status of existing data $(d_p^{existing}, d_h^{existing})$ and the availability of hidden write requests (d_h).

As a result, by construction, all physical data writes to the underlying device happen sequentially. The pattern is in fact composed of reads and writes to the same block, followed by the next block, and so on. For rotational media this means two things: (i) disk heads do not need to switch cylinders to write, which results in significantly lower (mostly rotational-only) latencies, and (ii) any predictive read/write-ahead buffering (at the various layers within the drive and the OS) can work extremely well, especially when compared with existing work that randomizes physical access patterns.

Reduced number of writes. The sequential nature of the physical write trace results in a smaller average number of blocks written per logical write compared to existing work [11]. In a PD solution, a few blocks will be accessed following certain rules to find a physical block for the logical data, lest the write trace leaks the access pattern. It is more likely that the block selected in PD-DM contains no valid data as it is the least recently written physical block. To set the stage for a detailed analysis, we first introduce two related components: "spare factor" and "traffic model".

"Spare factor" ρ. In order to store all the logical blocks of each logical volume, the *Data* area in PD-DM should contain at least $2N$ physical blocks ($N = max(N_p, N_h)$). In addition to these blocks, similar to existing work [11], "spare" space is reserved so that it can be easy to find available blocks to write. We denote the ratio of this additional space to the total size of the *Data* area as the "spare factor" ρ. Then the size of the *Data* area can be represented as $\frac{2N}{1-\rho}$. Note that the number of required blocks will change from $2N$ to $(k+1)N$ once we store maps on the disk (detailed in Section 3.2.2); and the "spare factor" will be $\frac{(k+1)N}{1-\rho}$ then.

Traffic model. We model the logical write traffic for each volume with Rosenblum's hot/cold data model [28]. It captures the spatial locality in real storage workloads [12] and can be applied to PD-DM by "separating" the logical address space of each volume into "hot" and "cold" groups, respectively. Two parameters r and f are introduced in the model. Specifically, a fraction f of the address space is "hot", and hit by a fraction r of the overall write requests. The remaining $1 - f$ fraction is "cold" and receives a fraction of $1 - r$ of all write requests. The write traffic inside each group is uniformly distributed. When $r = f = 100\%$, this becomes a special case where the entire traffic is uniformly distributed.

The average number of blocks written per logical write is related to the probability that the selected block is available for writing. Given a spare factor ρ, this probability is actually ρ for a randomly selected block. The average number of blocks needed to be accessed for each logical write is then $\frac{1}{\rho}$. The sequential write nature in PD-DM ensures that no less than $2N \cdot \frac{\rho}{1-\rho}$ logical writes can be performed during the period when the entire *Data* area is written once from beginning to end. This also implies that average number of blocks needed to be accessed for each logical write is then at most $\frac{1}{\rho} = O(1)$.

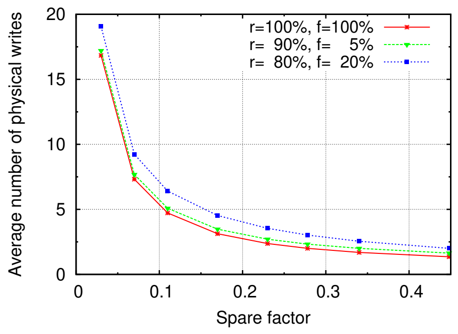


Fig. 4. Average number of physical writes required for one public write request under different types of logical write workload.

Moreover, as PD-DM always writes to the "least recently written" physical block, it is more likely to have been "invalidated" before being overwritten. The probability that the written block contains invalid data depends on both ρ and the incoming workload/traffic pattern. For example, if the traffic is uniformly distributed, the probability that the current written block contains invalid data can be represented as the closed-form Equation 1 [12]. Here $W(x)$ denote the Lambert's W function [10], which is the inverse of xe^x (i.e. $W(y) = x|xe^x = y$).

$$1 + (1-\rho) \cdot W(\frac{-1}{1-\rho} \cdot e^{\frac{-1}{1-\rho}}) \quad (1)$$

In general cases, the probability that the written block in PD-DM contains invalid data cannot be represented with a closed-form equation. But it can be calculated numerically [12].

Figure 4 shows how the average number of physical writes for one logical write changes when the logical traffic and spare factor ρ vary. The average number of physical writes decreases as ρ increases. Moreover, the more focused the write traffic (on a smaller logical address space), the smaller the average number of required physical write operations. To sum up, given a relatively small spare factor, the average number of physical write per logical write in PD-DM is much less compared to other randomization-based solutions.

3.2 Mapping & Block Occupancy Status

So far we have considered the mapping structures as black boxes. We now discuss their design, implementation and associated complexities.

3.2.1 Mapping-related Data Structures

Two different data structures are used to store logical-physical mappings for the two volumes, respectively: the public position map (PPM) for the public volume, and the hidden map B+ tree (HM) with its root node (HM_ROOT) for the hidden volume.

The PPM maps public logical block IDs to corresponding physical blocks on disk. It's an array stored at a fixed location starting at physical block ID α_{PPM}; subsequent offset logical block IDs are used as indices in the array. Its content requires no protection since it is related only to public data and leaks no information about the existence of hidden data.

Unlike in the case of the PPM for the public volume, we cannot store an individual position map for the hidden volume at a fixed location since accesses to this map may expose the existence of and access pattern to hidden data. Instead, we deploy an HM tree which is stored distributed *across the entire disk*, leaving only the root node (denoted by HM_ROOT) in a fixed location (physical block ID = α_{HM_Root}) of the mapping segment. To facilitate easy access and reconstruction of the HM, each node contains the location of all of its children nodes.

More specifically, HM indexes physical locations by logical block IDs. The leaf nodes are similar to the PPM which contains the mapping entries, and they are ordered from left to right (from leaf_1 to leaf_n in Figure 5). An example of a HM tree of height 3 is shown in Figure 5. A logical hidden block is mapped to its corresponding physical block by tracing the tree branch from the root to the corresponding leaf node containing the relevant mapping entry. Suppose b is the size of physical block IDs in bytes, then the height of the HM tree is $O(log_{\lfloor B/b \rfloor}(N_h))$. The last entry of each HM tree leaf node is used for a special purpose discussed in Section 3.2.3. Thus, only $\lfloor B/b \rfloor - 1$ mapping entries can be stored in one leaf node. Further, each internal node can have $\lfloor B/b \rfloor$ child nodes. Finally, the HM mapping entry of the i-th logical block is stored in the $(\frac{i}{\lfloor B/b \rfloor - 1})$-th leaf node at offset $i\%(\lfloor B/b \rfloor - 1)$. For a block size of 4KB and block IDs of 32 bits, the height of the map tree remains $k = 3$ for any volume size between 4GB and 4TB.

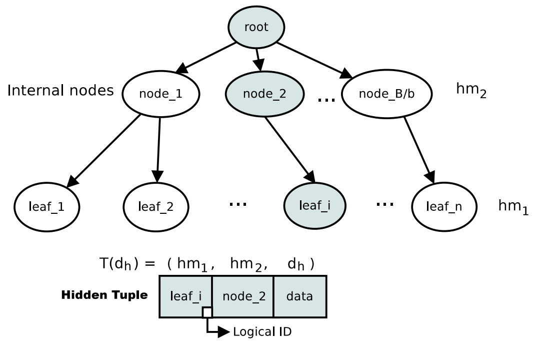


Fig. 5. Hidden logical blocks are mapped to the physical disk with a B+ tree map (e.g., of height 3 in this figure). A data block and the related HM blocks construct a "hidden tuple", denoted as $T(d_h)$. The HM blocks are the tree nodes in the path from the root to the relevant leaf containing the corresponding mapping entry for the data block. In this figure, a "hidden tuple" and the corresponding path of HM tree are colored gray.

The HM_ROOT block is encrypted with the hidden key K_h. Adversaries cannot read the map in the absence of K_h. Additionally, the HM_ROOT block will be accessed following a specific rule independent of the existence of a hidden volume. This ensures that accesses to HM_ROOT do not leak hidden information. More details in Section 3.3.

3.2.2 Get/Set_map

Managing the PPM is relatively simple. Logical block ID l_p is used to retrieve the physical block where the corresponding PPM entry resides (Algorithms 6 and 7 in Appendix A.2).

Managing the HM tree, however, requires much more care (Algorithms 8 and 9 in Appendix A.2). The access pattern to the HM tree should not be accessible to a curious multi-snapshot adversary. Fortunately, as the map entries are stored in the leaf nodes, accessing one map entry always requires traversing one of the tree paths from the root node to the corresponding leaf. Thus, each path (from the root to a leaf) is linked with and embedded within a given corresponding data block as a "hidden tuple" in PD-DM. And a "hidden tuple" is written to successive blocks in the *Data* area.

Figure 5 illustrates the "hidden tuple" concept for an example HM tree of height 3. The hidden tuples here consist of the data block itself and the nodes in the path from the root of HM to the relevant leaf node containing the mapping entry for the data block.[4] Let hm_i denote the HM tree node at level i in the path from the root to the leaf. Then the hidden tuple corresponding to hidden data d_h is denoted as $T(d_h) =< hm_1, hm_2, \cdots, hm_{k-1}, d_h >$ where k is the height of the HM tree. In order to accomplish a write request $Hid\_write(l_h, d_h)$, we need to not only write the data d_h but also update the HM entry to reflect the correct new mapping. This is achieved by also writing a hidden tuple and updating HM_ROOT.

Note that accessing one HM entry does not affect any other branches of the tree unrelated to it. Thus, importantly, *no cascading effects ensue.* To understand this better, consider that as leaves are ordered on logical addresses, changing the mapping entry in one leaf node only requires updating the corresponding index in its ancestor nodes which indicate the physical location of its corresponding child in the path. Other indexes remain the same and the rest of the tree is unchanged.

3.2.3 Get/Set_Status

PD-DM requires one more key structure providing the occupancy status of individual blocks. So far this was assumed to be a black box (Section 3.1) . We now detail.

Keeping track of the status of individual blocks is done using a public bit map (PBM). It contains one bit and one IV for each physical block in the *Data* area. A bit is set once public data is being written to the corresponding physical block. The bitmap ensures that we are able to check the state of a physical block before writing to it to avoid public data loss e.g., by overwriting. The PBM is public and not of concern with respect to adversaries. It is worth noting that the blocks occupied by hidden data are shown as unused in the PBM.

To protect the hidden data from overwriting, we deploy a number of additional mechanisms. We cannot use a public bitmap since it would simply leak without additional precautions, which necessarily introduce inefficiencies. Examples of such inefficient bitmaps include the in-memory bitmap used by Peters [27] and the fuzzy bitmap used by Pang [26] which introduced "dummy blocks" to explain blocks which were marked as in-use but unreadable by the filesystem.

4 The root node is not included in the hidden tuple since it is always stored as HM_ROOT as explained before. Updating the root node can be done in memory only so that the sequential disk write behavior is not affected.

Instead, PD-DM deploys a "reverse map" mechanism (similar to HIVE [8]) to map physical to logical blocks. The corresponding logical ID of each block containing hidden data is stored directly in the corresponding hidden tuple, or more specifically, in the last entry of the leaf node in the hidden tuple (see Figure 5).

This makes it easy to identify whether a hidden tuple (and its associated block) is still valid, i.e., by checking whether the logical block is still mapped to where the hidden tuple currently resides. For example, suppose the hidden tuple is stored in the physical blocks $\beta + 1$ to $\beta + k$, then the hidden data is in block $\beta + k$ and the logical ID of this hidden tuple is in block $\beta + 1$. Thus, the state of the hidden tuple will be obtained by first reading block $\beta+1$ for the logical ID l and then reading the current map entry for index l from the HM tree. If $HM[l] = \beta+k$ is true, then the hidden data there is still up to date. Otherwise, the hidden tuple is invalid (and can be overwritten).

As an optimization, in this process, one doesn't always need to read all k nodes in the path of the HM tree for the map entry. Any node read indicating that a node on the path is located between $\beta + 1$ to $\beta + k$ establishes $HM[l] = \beta + k$ as true.

Importantly, the above mechanisms do not leak anything to multi-snapshot adversaries since all enabling information is encrypted (e.g., the logical ID is part of the hidden tuple) and no data expansion occurs (would have complicated block mapping significantly). Status management ($Set\_status(\alpha, S)$ and $Get\_status(\alpha)$) can be done through reading/writing the hidden tuples. Thus, no additional Phy_write operations are needed.

Algorithm 3 $Set\_Hid\_map\_full(l_h, \alpha)$ – update a hidden map entry or fake map accesses

```
Input: $l_h$, $\alpha$, $\alpha_{HM\_Root}$, $k$
 1: if $l\_h = 0$ then                    ▷ Randomized data dummy is written
 2:     $map_{block} := Phy\_read(\alpha_{HM\_Root})$
 3:     $Phy\_write(\alpha_{HM\_Root}, map_{block})$
 4:     for $i \in \{2, \dots, k\}$ do
 5:         Fake map block $map_{dummy}$
 6:         $Phy\_write(\alpha - k + i - 1, map_{dummy})$
 7:     end for
 8: else                                   ▷ real hidden data is written
 9:     $Set\_Hid\_map(l_h, \alpha)$
10: end if
```

3.3 Full PD-DM With Maps On Disk

Once all mapping structures are on disk – in addition to replacing the *Get*/*Set_map* black boxes with Algorithms 6 – 9, a few more things need to be taken care of to ensure end-to-end PD.

Most importantly, accesses to the maps should not expose the existence of hidden data and behave identically regardless of whether users store data in hidden volumes or not. It is easy to see that no *Set_map* functions are called whenever the written data is either "existing" or "dummy" in Algorithm 2, which skews the access distribution and may arouse suspicion. To tackle this, additional fake accesses to the mapping-related data structures are performed as necessary. The *Set_Hid_map_full* function either sets a real hidden map entry or fakes those accesses (Algorithm 3), where $l_h = 0$ means faking accesses to related data structures.

As a result, the same number of physical write operations are performed regardless of whether the pair of public-hidden data written contains new data, existing data or dummy data in Algorithm 2. An atomic operation *PD_write* can be defined as Algorithm 4 to describe all those physical writes. $PD\_write(P_{head}, l_p, d_p, l_h, d_h)$ arranges a public-hidden data pair (d_p, d_h) together in the *Data* area and update HM (real or fake) using a number of sequential *Phy_write* operations.

Then the resulting complete set of steps for handling a logical write request can be illustrated with Algorithm 5. The l_h^{new} is set to be 0 when d_h is randomized data *dummy*, and l_p^{new} is 0 when d_p^{new} is existing public data. Logical read requests are straightforwardly handled by accessing the data and existing mapping structures as illustrated in Algorithms 10 and 11 (Appendix A.2).

Algorithm 4 $PD\_write(P_{head}, l_p, d_p, l_h, d_h)$

```
Input: $P_{head}, l_p, d_p, l_h, d_h$
 1: $Phy\_write(P_{head}, d_p)$                    ▷ Write the public data
 2: $Set\_Hid\_map\_full(l_h, P_{head} + k)$       ▷ Write the hidden map
 3: $Phy\_write(P_{head} + k, d_h)$                ▷ Write the hidden data
```

Algorithm 5 Public write + Hidden write – Full PD-DM with maps on the disk

```
Input: $Pub\_write(l_p, d_p)$, $Hid\_write(l_h, d_h)$, $P_{head}$
 1: while $d_p$ is not $Null$ do
 2:     $(d_p^{new}, d_h^{new}) = Get\_new\_data\_pair(d_p, d_h, P_{head})$
 3:     $PD\_write(P_{head}, l_p^{new}, d_p^{new}, l_h^{new}, d_h^{new})$
 4:     if $d_p^{new} == d_p$ then
 5:         $Set\_Pub\_map(l_p, P_{head})$, $d_p = Null$
 6:     end if
 7:     if $d_h^{new} == d_h$ then
 8:         $d_h = Null$
 9:     end if
10:     $P_{head} \leftarrow P_{head} + (k+1)$
11: end while
```

With maps on disk, *PD_write* operations writes more than just data blocks to the physical device. But those physical writes still happen sequentially. However, accessing the map-related data structures (PPM, PBM, ...) will bring in a few seeks. In-memory caches are maintained in PD-DM for those data structures, which can significantly reduce the number of seeks. Consider caching PPM blocks as an example: as $\lfloor B/b \rfloor$ mapping entries are stored in one mapping block, caching it allows the reading of map entries for $\lfloor B/b \rfloor$ logical addresses (in a sequential access) without seeking. Thus, comparing with the previous works which seeks before writing every randomly chosen block , PD-DM has lower seek-related expense.

Finally, it is easy to generalize PD-DM to have multiple volumes. A *PD_write* operation can write hidden data from any hidden volume.

3.4 Encryption

All blocks in the *Data* area are encrypted using AES-CTR with one-time-use, random per-block IVs. Those IVs are stored as part of the PBM. In fact, other similar schemes of equivalent security level can be envisioned and the deployed cipher mode can be any acceptable mode providing IND-CPA security.

4 Analysis

4.1 Security

We now prove that adversaries cannot distinguish the two write traces $\mathcal{W}(\mathcal{O}_0)$ and $\mathcal{W}(\mathcal{O}_1)$ and thus can win the game of Definition 1 with at most negligible advantage.

Theorem 1. *PD-DM is plausibly deniable.*

Proof (sketch): Each access pattern results in a write trace transcript available to the adversary through different device snapshots. The transcript contains these exact entries:

– Write $k+1$ blocks to the *Data* area and update the HM_ROOT with *PD_write* operations.
– Update the PPM accordingly.
– Update the PBM accordingly.
– Update the P_{head}.

In this transcript, Lemmas 4, 5, 6 and 7 (in Appendix A.3) show that the *locations* of all writes depend only on public information (public writes). Further, the *contents* of every write is either fully randomized or depends on public data only (public writes). This holds also for the write traces of $\mathcal{O}_0$ and $\mathcal{O}_1$ in Definition 1. As a result, adversaries cannot win the game of Definition 1 better than random guessing. □

4.2 I/O Complexity

As discussed above, to hide the existence of the hidden data, more work is being done in addition to just reading and writing the data itself. Here we discuss the associated I/O complexities and constants.

Public Read. A *Pub_read* request requires reading a PPM block for the map entry plus reading the corresponding data block. Thus, compared with the raw disk, *Pub_read* roughly reads twice as often and halves the throughput. It may be reduced using a smart memory caching strategy for the PPM blocks since consecutive PPM entries are stored in one PPM block.

Hidden Read. *Hid_read*, on the other hand, requires reading one HM tree path for the map entry first. Thus, $k+1$ blocks (k HM tree path blocks + data block) are read in total, which results in $B \cdot (k+1) = O(logN)$ complexity, where $k = logN$ is the height of the HM tree.

Underlying Atomic Write Operation. *PD_write* checks if the next $k+1$ blocks are free, fills those blocks with the newly formed pair of public data and hidden tuple, and updates the HM_ROOT block. Checking the state of the $k+1$ candidate blocks requires reading one PBM block and at most k HM tree nodes (Section 3.2.3). This first step gives $B \cdot (k+2)$, and results in $O(logN)$ since $k = logN$. Further, the step of writing the $k+1$ blocks in order gives $B \cdot (k+1)$ resulting in $O(logN)$ as well. Thus, the I/O complexity of the overall *PD_write* operation is $O(logN)$.

Public/Hidden Write. As described in Algorithm 5, only a few additional I/Os (constant number) are needed besides a *PD_write* for each iteration of the loop. As the I/O complexity of the *PD_write* operation is $O(logN)$, the complexity of each iteration is also $O(logN)$. The analysis in Section 3.1 shows that either a *Pub_write* or a *Hid_write* can be completed within $O(1)$ iterations ($O(1)$ *PD_writes*). Thus, the overall I/O complexity is $O(logN)$ for public/hidden write.

Throughput. Another fact worth noting is that, after an initial period in which sufficient writes have been seen for each volume, the device usually ends up with N valid blocks for each volume, regardless of the workload. At that point, throughput stabilizes and is not a function of uptime anymore.

4.3 Throughput Comparison

PD-DM accesses at most $O(logN)$ blocks for each logical access with relatively small constants (e.g., 2-3 for judiciously chosen spare factors). Moreover, physical accesses are mostly sequential, thus reducing latency components and improving performance.

By comparison, existing work such as HIVE features an I/O complexity of $O(log^2N)$ [8] with constants of either 64 (without a stash) or 3 (with a stash). Furthermore, access patterns are random, making seeks impactful and caching/buffering difficult. As a result, on rotational media, PD-DM outperforms existing work by up to two orders of magnitude.

5 Evaluation

PD-DM is implemented as a Linux block device mapper. Encryption uses AES-CTR with 256 bit keys and random IVs. Blocks are 4KB each. Two virtual logical devices (one public and one hidden) are created on one physical disk partition.

Experiments were run on a machine with dual-core i5-3210M CPUs, 2GB DRAM, and Linux kernel 3.13.6. The underlying block devices were a WD10SPCX-00KHST0 5400 rpm HDD and a Samsung-850 EVO SSD. The average seek time of the HDD is 8.9ms [4].

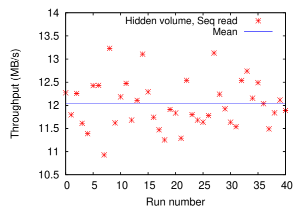
Two volumes of 40GB each were created on a 180GB disk partition ($\rho \approx 0.1$). Ext4 FSes were installed on each volume in ordered mode. Additionally, an ext4 FS mounted on top of a dm-crypt (encryption only) volume was benchmarked on the same disk. HIVE [2] was compiled and run on the same platform.

5.1 HDD Benchmarks

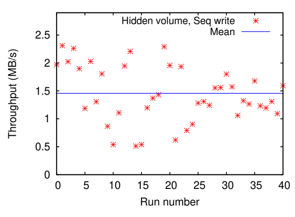
Two benchmarks (Bonnie++[1], IoZone[25]) were used to evaluate performance under sequential and random

Operation	dm-crypt	PD-DM	HIVE [8]
Public Read	**97.3**	**17.452 ± 0.367**	**0.095**
Public Write	**88.0**	**1.347 ± 0.093**	**0.013**
Hidden Read	**n/a**	**12.030 ± 0.088**	**0.113**
Hidden Write	**n/a**	**1.456 ± 0.086**	**0.016**

Table 2. Sequential workload throughput (MB/s) comparison with the Bonnie++ benchmark. PD-DM is 100-150 times faster than HIVE for sequential read and write. The PD-DM throughputs reported here are the average value over 40 runs with the standard error.



(a) Seq read



(b) Seq write

Fig. 6. PD-DM hidden I/O throughput results of multiple runs.

workloads, respectively. Results are in Tables 2 and 3. To further evaluate the impact of caching parts of mapping-related data structures, the IoZone experiment was run multiple times for different cache sizes. Results are illustrated in Figure 8.

5.1.1 Throughput

Sequential I/O. For consistency in comparing with HIVE, Bonnie++ was first used to benchmark the throughput of sequential read and write for both volumes. The mapping-related structures cache was limited to 1MB. Results *averaged* over 40 runs are shown in Table 2. Overall, PD-DM performs about two orders of magnitude faster for both public and hidden volumes. For example, the basic Bonnie++ benchmark required about one full week to complete on existing work [8] in the same setup. The same benchmark completed in under 4 hours on PD-DM

Figure 6 shows the result of every run as a sample point as well as the mean value for the hidden volume I/O throughput. Although the throughput varies around the mean value for both read and write, it does not decrease as more runs are completed. Note that Bonnie++ writes more than 8GB of data to the disk in each run – as a result the disk has been written over and over during the test. Moreover, write throughput nears the mean with increasing run number, i.e., the disk is converging to a stable state. The results for the public volume are similar and not presented here.

As expected, PD-DM and all existing PD systems are slower when compared to simple encryption (dm-crypt) without PD. However, notably, *PD-DM is the first PD system protecting against a multi-snapshot adversary with performance in the MB/s range* on hard disks, and 15MB/s+ for public reads.

Random I/O. IoZone was run to evaluate system behavior under random I/O workloads. IoZone reads files of different sizes using random intra-file accesses. Direct I/O is set for all file operations so that I/O caching by the OS is bypassed. Further, the RAM allocated for PD-DM mapping-related data structures (PPM, PBM, etc) is limited to 1MB. Results of random I/O throughputs for file sizes of 1GB are shown in Table 3.

It can be seen that the PD-DM sequential write mechanism really makes an impact and performs 3–30× faster than HIVE. Notably, even standard dm-crypt is only 2–3× faster.

Overall, PD-DM performs orders of magnitude faster than existing solutions for both sequential and random workloads. Moreover, a great potential advantage of PD-DM for random workloads can be seen by analyzing the results in Figure 7, depicting the performance for sequential and random I/O of the dm-crypt and the two volumes in PD-DM respectively.

Note that both read and write throughputs drop significantly faster for dm-crypt when the I/O pattern changes from sequential to random. In contrast, throughputs of the two PD-DM volumes are significantly less affected, especially for writes. This is because the underlying write pattern is always sequential in PD-DM, regardless of logical I/O request locality.

It is important to note however that accessing mapping structures may disturb the sequentiality of resulting write patterns. Fortunately, the design allows for even a small amount of cache to have a great impact.

5.1.2 Cache Size

To evaluate the influence of cache sizes on performance under sequential or random access with files of different sizes, IoZone is run for different cache sizes. Results are in Figure 8.

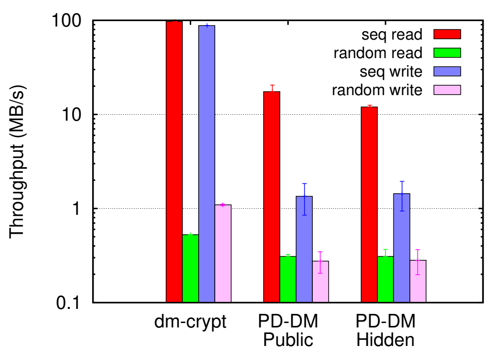
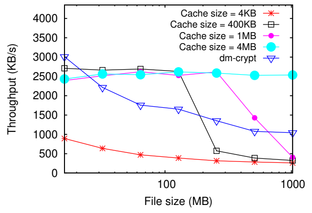
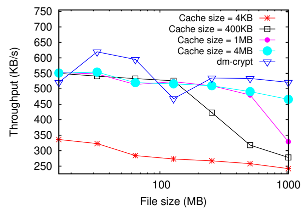


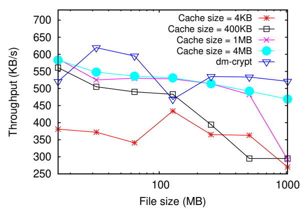
Fig. 7. PD-DM throughput comparison with dm-crypt. Random I/O results in a throughput drop. PD-DM is less affected because the underlying write pattern is always sequential regardless of logical I/O request locality.



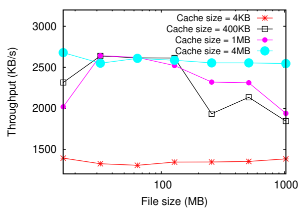
(a) Hidden volume of PD-DM vs. dm-crypt, Random write



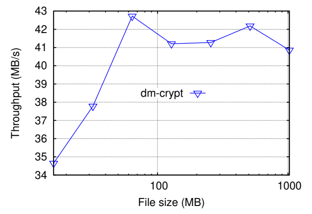
(b) Hidden volume of PD-DM vs. dm-crypt, Random read



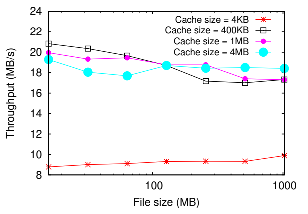
(c) Public volume of PD-DM vs. dm-crypt, Random read



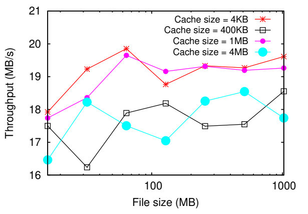
(d) PD-DM, Hidden volume, Sequential write



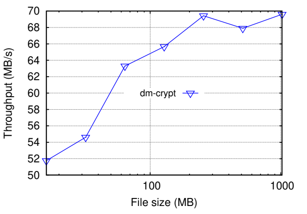
(e) dm-crypt, Sequential write



(f) PD-DM, Hidden volume, Sequential read



(g) PD-DM, Public volume, Sequential read



(h) dm-crypt, Sequential read

Fig. 8. IoZone throughput comparison for different workloads, cache sizes, and file sizes between PD-DM and dm-crypt. Figure 8(a) shows how PD-DM random-write throughput is affected by cache size. Write throughput exceeds that of dm-crypt as the cache size increases, since caching the mapping data structures reduces extraneous seeks and increases locality/sequentiality in the underlying writes. Similarly, for random read (Figure 8(b) and 8(c)), increased caching improves performance for both public and hidden volumes, especially when the file size is relatively large. But the throughput cannot be better than dm-crypt. Cache size determines the turning point. Sequential I/O is illustrated in Figures 8(d)–8(h). Throughputs of dm-crypt are shown in Figures 8(e) and 8(h). PD-DM write throughput (Figure 8(d)) varies in a small range except from one special case (cache size=4KB), since one hidden map entry is related to 2 map blocks (8KB). Read throughput (Figures 8(f) and 8(d)) is also less affected by caching as long as the number of cache blocks is larger than the number of mapping blocks read for one mapping entry (4KB for public volume and 8KB for hidden volume). Contrastingly, the sequential-write throughput of dm-crypt increases with the increasing file size at first, then remains in a stable range.

Operation	dm-crypt	PD-DM	HIVE [8]
Public Read	0.526	0.309 ± 0.005	0.012
Public Write	1.094	0.276 ± 0.017	0.010
Hidden Read	n/a	0.309 ± 0.011	0.097
Hidden Write	n/a	0.282 ± 0.024	0.014
Seeks Per Second	1227	336/89	10.3/5.7

Table 3. Performance comparison for random workloads. The PD-DM throughputs reported here are the average value over 40 runs with the standard error. The first four rows show throughput (MB/s) of random read and write for different volumes respectively as measured by IoZone. The last row represents the number of seeks per second (the number for public and hidden volume are separated by a slash) reported by Bonnie++.

Write throughput. Figure 8(a) illustrates random-write throughput of the hidden volume with different cache sizes. Random-write throughput for dm-crypt decreases gradually (from 3MB/s to 1MB/s) with increasing file sizes, mainly because of higher number of seeks for larger files. However, PD-DM random-write throughput behaves differently. For caches above 400KB, it remains stable for relatively small files, and then drops for file sizes beyond a threshold. As expected, larger caches result in larger thresholds. For caches over 4MB, write throughput stabilizes for files smaller than 1GB. This means that map blocks are mostly read from/write to the caches without disk seeks. As a result, PD-DM random-write throughput exceeds that of dm-crypt.

It is worth noting that the same cache mechanism does not work for dm-crypt as its logical-to-physical block mapping is fixed and the larger number of seeks derives from the logical-layer access randomness. In contrast, in PD-DM, additional seeks occur only when accessing mapping blocks where multiple entries are stored together. This effectively maximizes the overall cache-hit rate for the mapping data structures.

Sequential-write throughput for the hidden volume (Figure 8(d)) is compared with the dm-crypt curve (Figure 8(e)). As seen, the cache size has a decreasing effect above caches larger than 4KB (1 block). Moreover, the throughput is not greatly affected by file size (unlike in the case of dm-crypt), although a larger file still results into a lower throughput.

Write throughput for the public volume is similar to the hidden volume mostly because hidden writes are executed together with public writes.

Read throughput. Random-read throughput for both volumes (Figures 8(b) and 8(c)) follow a similar trend as the random-write throughput (Figure 8(a)) – throughput decreases significantly for file sizes beyond a certain threshold. Further, larger caches result in larger thresholds. Finally, the threshold for both volumes is the same for the same cache size.

Sequential-read throughputs for both volumes are less affected by either file or cache sizes (Figures 8(f)–8(h)). A special interesting case occurs for caches of only 4KB (1 block). In that case, the sequential-read throughput of the hidden volume drops by half (Figures 8(f)), similar to what happens in Figures 8(d). The reason for this is that reading a hidden mapping entry requires reading more than one block when the height of the hidden map tree is larger than 2. This results in continuous cache misses. The same situation doesn't happen for the public volume, mainly because a single physical block can contain multiple adjacent public mapping entries, thus even a one-block cache can make an impact.

The variation in sequential-read throughput is mildly affected by the existing layout of file blocks on disk. After all, a continuous file may be fragmented by existing data on disk.

5.2 SSD Benchmarks

PD-DM is obviously expected to perform most of its magic on HDDs where reduced numbers of disk seeks can result in significant performance benefits. Fortunately however, PD-DM performs also significantly better than existing work when run on SSDs.

A throughput comparison including PD-DM, HIVE and dm-crypt is shown in Table 4. Only a minimal 40KB cache is being used – since seeks are cheap in the case of SSDs, we expected (and found) that cache size has little effect on performance.

Overall, although the performance advantage of reducing the number of seeks decreases for SSDs, PD-DM still outperforms existing work due to two important facts: (i) reads are unlinked with writes, and (ii) PD-DM write complexity is only $O(logN)$, compared to $O(log^2N)$ in HIVE.

Operation	dm-crypt	PD-DM	HIVE [8]
Public Read	225.56	99.866 ± 0.015	0.771
Public Write	210.10	2.705 ± 0.023	0.546
Hidden Read	n/a	97.373 ± 0.009	4.534
Hidden Write	n/a	3.176 ± 0.017	0.609

Table 4. Throughput (MB/s) comparison on SSD for a sequential workload using the Bonnie++. PD-DM is 30-180x faster than HIVE for sequential read, and 5x faster for sequential write.

6 Discussion

Advantages. PD-DM has a number of other advantages over existing schemes aside from the performance. First, *PD-DM does not require secure stash queues* as long as there are enough public writes ongoing. Notwithstanding, it is worth noting that caches and filesystem related buffers should be protected from adversaries since they contain information about hidden volumes. Second, *PD-DM preserves data locality from logical accesses.* The logical data written together will be arrange together on the device with high probability. Third, *PD-DM breaks the bond between reads and writes and ensures security.* Adversaries may have a prior knowledge of the relationship between physical reads and writes since reads and writes are usually generated together by a filesystem. HIVE masks hidden writes using public reads, which may break this FS-implied relationship and arouse suspicions. For example, a hidden write masqueraded as a read to a public file data block may raise suspicion if usually real public reads are coupled with writing metadata or journaling, writes which may be absent in this case.

Physical Space Utilization. Considering only the data ORAM in HIVE with two logical volumes, HIVE can use at most 50% of the disk for both public data and hidden data since half the disk is required to be free. Thus, assuming that the two logical volumes are of the same size, each would be of 25% of the total device.

In PD-DM, ρ and k decide volume sizes. Both the public and the hidden volume is a fraction of $(1-\rho)\cdot\frac{1}{k+1}$ of the entire device (considering only the *Data* area). Assuming that $k = 3$ (works for volume size from 4GB to 4TB), an empirically chosen sweet spot balancing space and performance can be found at $\rho = 20\%$ which results in an overall volume size of around 20% of the device.

Optimization Knobs. Performance is impacted by a number of tuning knobs that we discuss here.

Increasing ρ results in increased throughput at the cost of decreased space utilization. The intuition here is that the probability that the $k + 1$ target blocks for *PD_writes* contain overwritable (invalid) data increases with increasing ρ. ρ enables fine-tuning of the trade-off between throughput and space utilization.

Further, changing the ratio between the number of public data blocks and that of hidden data blocks written together can help balancing the public write throughput and the waiting time of hidden writes. In this paper, we assume that one public data block is always written together with one hidden data block. For workloads where the user accesses public data more often than hidden data, we can actually change the ratio so that more than one public data blocks are written with one hidden data block (e.g., 2 public data paired with 1 hidden data). This results in a higher throughput for public writes at the cost of delayed writes for hidden data and a requirement for more memory to queue up outstanding hidden data writes.

The physical space utilization will also be affected when the above ratio changes. The size of public volume increases while the size of hidden volume decreases. For example, if 2 public data are paired with 1 hidden data, then $k + 2$ instead of $k + 1$ blocks will be written in the *Data* area for that. For an unchanged ρ, the size of public volume becomes $(1-\rho)\cdot\frac{1}{k/2+1} > (1-\rho)\cdot\frac{1}{k+1}$ of the size of the *Data* area while the size of hidden volume becomes $(1-\rho)\cdot\frac{1}{2(k/2+1)} = (1-\rho)\cdot\frac{1}{k+2} < (1-\rho)\cdot\frac{1}{k+1}$ of the size of the *Data* area.

7 Related Work

Plausibly-deniable encryption (PDE) is related to PD and was first introduced by Canetti et al. [9]. PDE allows a given ciphertext to be decrypted to multiple plaintexts, by using different keys. The user reveals the decoy key to the adversary when coerced and plausibly hides the actual content of the message. Some examples of PDE enabled systems are [7, 19]. Unfortunately, most PDE schemes only work for very short messages and are not suitable for larger data storage devices.

Steganographic Filesystems. Anderson et al. [5] explored "steganographic filesystems" and proposed two ideas for hiding data. A first idea is to use a set of cover files and their linear combinations to reconstruct hidden files. This solution is not practical since the associated performance penalties are significant. A second idea is to store files at locations determined by a crypto hash of the filename. This solution required storing multiple copies of the same file at different locations to prevent data loss. McDonald and Kahn [23] implemented a steganographic filesystem for Linux on the basis of this idea. Pang et al. [26] improved on the previous constructions by avoiding hash collisions and providing more efficient storage.

These steganographic filesystem ideas defend only against a single-snapshot adversary. Han et al. [17] designed a "Dummy-Relocatable Steganographic" (DRSteg) filesystem that allows multiple users to share the same hidden file. The idea is to runtime-relocate data to confuse a multi-snapshot adversary and provide a degree a deniability. However, such ideas do not scale well to practical scenarios as they require assumptions of multiple users with joint-ownership and plausible activities on the same devices. Gasti et al. [16] proposed PD solution (DenFS) specifically for cloud storage. Its security depends on processing data temporarily on a client machine, and it is not straightforward to deploy DenFS for local storage.

DEFY [27] is a plausibly-deniable file system for flash devices targeting on multi-snapshot adversaries. It is based on WhisperYAFFS [30], a log structured filesystem which provides full disk encryption for flash devices. It is important to note that, due to its inherent log-structure nature derived from WhisperYAFFS, similar to PD-DM, DEFY also writes data sequentially underneath on flash devices with the intention of maintaining a reasonable flash wear-leveling behavior and thus not significantly reduce device lifespan.

However, unlike PD-DM, PD in DEFY is unrelated to the sequential access pattern and instead is achieved through encryption-based secure deletion. Additionally, DEFY takes certain performance-enhancing shortcuts which immediately weaken PD assurances and associated security properties as follows.

Firstly, DEFY doesn't guarantee that a block DEFY writes is independent from the existence of hidden levels. Specifically, all writes in DEFY will bypass the blocks occupied by hidden data unless the filesystem is always mounted at the public level. This is a significant potential vulnerability that undermines PD and requires careful analysis.

Secondly, DEFY writes checkpoint blocks for hidden levels after all data has been written to the device. Thus, these checkpoint blocks cannot be explained as deleted public data any more, which leaks the existence of hidden levels.

Thirdly, DEFY requires all filesystem-related metadata to be stored in memory. This does not scale for memory constrained systems with large external storage. The use of an in-memory free-page bitmap aggravates this problem. In contrast, PD-DM stores mapping-related data structures securely on disk and allows the option of caching portions thereof.

Block Devices. At block device level, disk encryption tools such as Truecrypt [3] and Rubberhose [19] provide PD against single-snapshot adversaries. Mobiflage [29] provides PD for mobile devices but cannot protect against multi-snapshot adversaries as well.

Blass et al. [8] (HIVE) are the first to deal with PD against a multi-snapshot adversary at device level. They use a write-only ORAM for mapping data from logical volumes to an underlying storage device and hiding access patterns for hidden data within reads to non-hidden (public) data. Chakraborti et al. [6] use writes to public data to hide accesses to hidden data based on another write-only ORAM that does not need recursive maps.

8 Conclusion

PD-DM is a new efficient block device with plausible deniability, resistant against multi-snapshot adversaries. Its sequential physical write trace design reduces important seek components of random access latencies on rotational media. Moreover, it also reduces the number of physical accesses required to serve a logical write.

Overall, this results in orders of magnitude (10–100×) speedup over existing work. Notably, PD-DM is the first plausible-deniable system with a throughput finally getting within reach of the performance of standard encrypted volumes (dm-crypt) for random I/O. Further, under identical, fine-tuned caching, PD-DM outperforms dm-crypt for random I/O.

Acknowledgments

This work was supported by the National Science Foundation through awards 1161541, 1318572, 1526102, 1526707.

References

[1] *Bonnie++*. "http://www.coker.com.au/bonnie++".

[2] *Hive*. "http://www.onarlioglu.com/hive".

[3] *TrueCrypt*. "http://truecrypt.sourceforge.net/".

[4] Western digital blue wd10spcx. "https://www.goharddrive.com/Western-Digital-Blue-WD10SPCX-1TB-5400RPM-2-5-HDD-p/g02-0319.htm".

[5] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *Information Hiding*, pages 73–82. Springer, 1998.

[6] C. C. Anrin Chakraborti and R. Sion. DataLair: Efficient block storage with plausible deniability against multi-snapshot adversaries. *Proceedings on Privacy Enhancing Technologies*, 2017(3), 2017.

[7] D. Beaver. Plug and play encryption. In *Advances in Cryptology – CRYPTO'97*, pages 75–89. springer.

[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*, pages 203–214. ACM, 2014.

[9] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable encryption. In *Advances in Cryptology – CRYPTO'97*, pages 90–104. Springer, 1997.

[10] R. M. Corless, G. H. Gonnet, D. E. Hare, D. J. Jeffrey, and D. E. Knuth. On the lambertw function. *Advances in Computational mathematics*, 5(1):329–359, 1996.

[11] P. Desnoyers. Analytic models of ssd write performance. *Trans. Storage*, 10(2):8:1–8:25, Mar. 2014.

[12] P. Desnoyers. Analytic models of SSD write performance. *ACM Transactions on Storage (TOS)*, 10(2):8, 2014.

[13] F. Douglis and J. Ousterhout. Log-structured file systems. In *COMPCON Spring'89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers.*, pages 124–129. IEEE, 1989.

[14] J. Dursi. On random vs. streaming I/O performance; or seek(), and you shall find – eventually, 2015. "http://simpsonlab.github.io/2015/05/19/io-performance/".

[15] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas. Freecursive ORAM:[nearly] free recursion and integrity verification for position-based oblivious ram. In *ACM SIGPLAN Notices*, volume 50, pages 103–116. ACM, 2015.

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

[17] J. Han, M. Pan, D. Gao, and H. Pang. A multi-user steganographic file system on untrusted shared storage. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 317–326. ACM, 2010.

[18] J. Hinks. Nsa copycat spyware could be snooping on your hard drive, 2015. "https://goo.gl/Ktesvl".

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

[20] C. M. Kozierok. Access time, 2001. "http://pcguide.com/ref/hdd/perf/perf/spec/pos_Access.htm".

[21] C. M. Kozierok. Seek time, 2001. "http://pcguide.com/ref/hdd/perf/perf/spec/posSeek-c.html".

[22] L. Mathews. Adobe software may be snooping through your hard drive right now, 2014. "https://goo.gl/lXzPS3".

[23] A. D. McDonald and M. G. Kuhn. StegFS: A steganographic file system for Linux. In *Information Hiding*, pages 463–477. Springer, 1999.

[24] J. Mull. How a Syrian refugee risked his life to bear witness to atrocities. Toronto Star Online, posted 14-March-2012, 2012. "https://goo.gl/QsivgB".

[25] W. Norcott and D. Capps. IOZone filesystem benchmark, 2016. "http://www.iozone.org/".

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

[27] T. Peters, M. Gondree, and Z. N. J. Peterson. DEFY: A deniable, encrypted file system for log-structured storage. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*, 2015.

[28] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[29] A. Skillen and M. Mannan. On implementing deniable storage encryption for mobile devices. 2013.

[30] WhisperSystems. Github: Whispersystems/whisperyaffs: Wiki, 2012. "https://github.com/WhisperSystems/WhisperYAFFS/wiki".

[31] X. Zhou, H. Pang, and K.-L. Tan. Hiding data accesses in steganographic file system. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 572–583. IEEE.

A Appendix

A.1 PD-CPA Game

1. Adversary $\mathcal{A}$ provides a storage device $\mathcal{D}$ (*the adversary can decide its state fully*) to challenger $\mathcal{C}$.
2. $\mathcal{C}$ chooses two encryption keys K_{pub} and K_{hid} using security parameter λ and creates two logical volumes, V_{pub} and V_{hid}, both stored in $\mathcal{D}$. Writes to V_{pub} and V_{hid} are encrypted with keys K_{pub} and K_{hid} respectively. $\mathcal{C}$ also fairly selects a random bit b.
3. $\mathcal{C}$ returns K_p to $\mathcal{A}$.
4. The adversary $\mathcal{A}$ and the challenger $\mathcal{C}$ then engage in a polynomial number of rounds in which:
   (a) $\mathcal{A}$ selects access patterns $\mathcal{O}_0$ and $\mathcal{O}_1$ with the following restriction:
      i. $\mathcal{O}_1$ and $\mathcal{O}_0$ include the same writes to V_{pub}.
      ii. Both $\mathcal{O}_1$ and $\mathcal{O}_0$ may include writes to V_{hid}
      iii. $\mathcal{O}_0$ and $\mathcal{O}_1$ should not include more writes to V_{hid} than ϕ times the number of operations to V_{pub} in that sequence.
   (b) $\mathcal{C}$ executes $\mathcal{O}_b$ on $\mathcal{D}$ and sends a snapshot of the device to $\mathcal{A}$.
   (c) $\mathcal{A}$ outputs b'.
   (d) $\mathcal{A}$ is said to have "won" the round iff. $b' = b$.

A.2 Algorithms

Algorithm 6 *Set_Pub_map*(l_p, α)

```
Input: $l_p$, $\alpha, \alpha_{PPM}$
    // Find the physical block containing the map entry
 1: $\alpha_M := \alpha_{PPM} + \lfloor l_p / \lfloor B/b \rfloor \rfloor$
 2: $map_{block} := Phy\_read(\alpha_M)$
    // Set the map entry and write back
 3: $map_{block}[l_p mod \lfloor B/b \rfloor] \leftarrow \alpha$
 4: $Phy\_write(\alpha_M, map_{block})$
```

Algorithm 7 $\alpha = $ *Get_Pub_map*(l_p)

```
Input: $l_p$, $\alpha_{PPM}$  ▷ $\alpha_{PPM}$ is the starting block ID of PPM in
    the physical device
Ouput: $\alpha$
    // Find the physical block containing the map entry
 1: $\alpha_M := \alpha_{PPM} + \lfloor l_p / \lfloor B/b \rfloor \rfloor$
 2: $map_{block} := Phy\_read(\alpha_M)$
    // Get the map entry
 3: $\alpha := map_{block}[l_p mod \lfloor B/b \rfloor]$
```

Algorithm 6 to 9 define how the logical to physical maps for public and hidden volumes are maintained in PD-DM.

Algorithm 8 *Set_Hid_map*(l_h, α)

```
Input: $l_h$, $\alpha$, $\alpha_{HM\_Root}$, $k$
    // Update the HM_ROOT
 1: $\alpha_M \leftarrow \alpha_{HM\_Root}$, $X \leftarrow (\lfloor B/b \rfloor - 1) \cdot \lfloor B/b \rfloor^{k-2}$
 2: $map_{block} := Phy\_read(\alpha_M)$
 3: $\alpha_M := map_{block}[l_h/X]$
 4: $map_{block}[l_h/X] := \alpha - k + 1$
 5: $Phy\_write(\alpha_{HM\_Root}, map_{block})$
 6: $l_h \leftarrow l_h mod X$
 7: for $i \in \{2, \ldots, k\}$ do
    // Update each layer of the HM tree by writing sequentially
 8:     $X \leftarrow (\lfloor B/b \rfloor - 1) \cdot \lfloor B/b \rfloor^{k-i-1}$
 9:     $map_{block} := Phy\_read(\alpha_M)$
10:     $\alpha_M := map_{block}[l_h/X]$
11:     $map_{block}[l_h/X] := \alpha - k + i$
12:     $Phy\_write(\alpha - k + i - 1, map_{block})$
13:     $l_h \leftarrow l_h mod X$
14: end for
```

Algorithm 9 $\alpha = $ *Get_Hid_map*(l_h)

```
Input: $l_h$, $\alpha_{HM\_Root}$, $k$   ▷ $\alpha_{HM\_Root}$ is the block ID
    of HM_ROOT in the physical device, $k$ is the height of the
    HM tree
Ouput: $\alpha$
 1: $\alpha_M \leftarrow \alpha_{HM\_Root}$
 2: for $i \in \{1, \ldots, k-1\}$ do
    // Traverse one path in the HM tree from root to corre-
    sponding leaf node
 3:     $X \leftarrow (\lfloor B/b \rfloor - 1) \cdot \lfloor B/b \rfloor^{k-i-1}$
 4:     $map_{block} := Phy\_read(\alpha_M)$
 5:     $\alpha_M := map_{block}[l_h/X]$
 6:     $l_h \leftarrow l_h mod X$
 7: end for
 8: $map_{block} := Phy\_read(\alpha_M)$
    // Get the map entry
 9: $\alpha := map_{block}[l_h]$
```

Algorithm 10 and 11 illustrate how logical read requests are performed in PD-DM for public and hidden volume respectively.

A.3 Security Proofs

Lemma 1. *Write traces of a PD_write without a hidden write are indistinguishable from write traces of a PD_write with a hidden write request.*

Algorithm 10 $d_p = Pub\_read(l_p)$

```
Input: $l_p$
Ouput: $d_p$
1: $\alpha := Get\_Pub\_map(l_p)$
2: $d_p := Phy\_read(\alpha)$
```

Algorithm 11 $d_h = Hid\_read(l_h)$

```
Input: $l_h$
Ouput: $d_h$
1: $\alpha := Get\_Hid\_map(l_h)$
2: $d_h := Phy\_read(\alpha)$
```

Proof (sketch): As shown in Algorithm 4, any PD_write writes a pair of public data d_p and hidden tuple $T(d_h)$ or $T(dummy)$ to the next $k+1$ blocks of the *Data* area and updates the HM_ROOT – notwithstanding of whether hidden write requests exist or not. Further, the written public data block and hidden tuple are encrypted with a semantically secure cipher which prevents adversaries from distinguishing between tuples with new hidden data, existing hidden data or just dummy data. Overall, both the location and the content of write traces are independent, fixed and unrelated to whether or not a $Hid\_write(l_h, d_h)$ request was associated with this underlying PD_write. □

Lemma 2. *The number of PD_write operations executed for a Pub_write request is not related to whether Hid_write requests are performed together with it.*

Proof (sketch): As illustrated in Algorithm 2, the while loop will not stop until the public data d_p is written by one PD_write and then set as Null. This will happen only when the obtained public status s_p is invalid, which only relates to the state of the *public* data on the disk – and is thus independent on whether any Hid_writes are performed. □

Lemma 3. *The number of underlying PD_write operations triggered by any access pattern is not related to whether it contains any hidden writes.*

Proof (sketch): This follows straightforwardly now since PD_write operations can only be triggered by public write requests. Note that hidden writes can certainly be completed with those PD_writes under the restrain 6(a)iv. Accordingly (Lemma 2), the number of PD_write operations needed to execute any access pattern depends on the public writes and has nothing to do with the hidden writes. □

Lemma 4. *Write traces to the* Data *area are indistinguishable for any access pattern that contains the same public writes, notwithstanding their contained hidden writes.*

Proof (sketch): For access patterns that contain the same public writes, Lemma 1 and Lemma 3 ensure that the corresponding number of PD_write operations are the same and their actual write traces are indistinguishable. As PD-DM always write to the *Data* area with PD_write operations, the write traces to the *Data* area are indistinguishable . □

Lemma 5. *Write traces to the PPM are identical for any access pattern that contains the same public writes, notwithstanding their contained hidden writes.*

Proof (sketch): PPM is written to only in the case of a Pub_write. For access patterns that contain the same Pub_write operations, the corresponding PPM write traces are the same. □

Lemma 6. *Write traces to the PBM are indistinguishable for any access pattern that contains the same public writes, notwithstanding their contained hidden writes.*

Proof (sketch): Each $Pub\_write(l_p, d_p)$ marks the corresponding bits in the PBM bitmap for the written physical blocks as well as the “recycled” physical block where the “old” version of l_p resided (Section 3.2). Since no other operations modify the bitmap, any modifications thereto relate only to Pub_write operations in an access pattern. □

Lemma 7. *Write trace to the P_{head} is indistinguishable for any access pattern that contains the same public writes, notwithstanding their contained hidden writes.*

Proof (sketch): P_{head} is incremented by $k+1$ for each PD_write. As a result, the associated write trace is only related to the number of PD_write operations – this number is decided by the public writes in an access pattern only (Lemma 3). Thus, the resulting write trace is indistinguishable for any access pattern that contains the same public writes. □