Toward Robust Hidden Volumes using Write-Only Oblivious RAM

Erik-Oliver Blass   Travis Mayberry   Guevara Noubir   Kaan Onarlioglu

College of Computer and Information Science
Northeastern University, Boston MA
Email: {blass|travism|noubir|onarliog}@ccs.neu.edu

ABSTRACT

With sensitive data being increasingly stored on mobile devices and laptops, hard disk encryption is more important than ever. In particular, being able to plausibly deny that a hard disk contains certain information is a very useful and interesting research goal. However, it has been known for some time that existing "hidden volume" solutions, like TrueCrypt, fail in the face of an adversary who is able to observe the contents of a disk on multiple, separate occasions. In this work, we explore more robust constructions for hidden volumes and present HIVE, which is resistant to more powerful adversaries with multiple-snapshot capabilities. In pursuit of this, we propose the first security definitions for hidden volumes, and prove HIVE secure under these definitions. At the core of HIVE, we design a new *write-only* Oblivious RAM. We show that, when only hiding writes, it is possible to achieve ORAM with optimal $O(1)$ communication complexity and only poly-logarithmic user memory. This is a significant improvement over existing work and an independently interesting result. We go on to show that our write-only ORAM is specially equipped to provide hidden volume functionality with low overhead and significantly increased security. Finally, we implement HIVE as a Linux kernel block device to show both its practicality and usefulness on existing platforms.

1. INTRODUCTION

Disk encryption is an important security technology that is increasingly being used by individuals and businesses alike. All major operating systems now support basic encrypted volumes natively, and both corporations and governments are increasingly mandating [6] the use of full disk encryption. Additionally, there are open source software products, most prominently TrueCrypt [20], that provide more advanced solutions.

One of the advanced features that TrueCrypt offers is a so-called "hidden volume". Instead of a single encrypted volume, a user may optionally have two encrypted volumes. These volumes are encrypted with different keys (derived from passwords), and the user has the ability to *plausibly deny* the existence of the second volume. An adversary, knowing only the password to the first volume, cannot tell for sure whether there exists a second volume, let alone what its contents may be. Given the widespread use of encrypted disks, this is a very useful feature. If an adversary takes possession of an encrypted disk, they know that there is at least *some* data on that disk. They can then coerce the user to reveal their password used to encrypt the disk. With a hidden volume, the user can reveal the password to the first volume while withholding the password for the second. The adversary will not know whether the second volume exists, and therefore cannot be sure if there even is a second password for the user to reveal.

TrueCrypt accomplishes this by storing the second, "hidden" volume inside the free space of the first, "main" volume. Since the semantics of TrueCrypt guarantee that all free space in the encrypted volume will be filled with random data, and the encryption used is presumed to be indistinguishable from random, an adversary cannot tell if the blocks marked "free" in the main volume are actually free or if they contain encrypted data that is part of a hidden volume.

However, as already noticed by Czeskis et al. [4], TrueCrypt's approach has a significant flaw: if the adversary has the ability to take multiple "snapshots" of the hard disk at different times, they can determine with high probability whether a hidden volume exists. Since disk encryption is specifically designed to protect against scenarios where the user loses control of their device, this is a major drawback. Requiring that an adversary has access to the machine and hard disk only one time is in many situations unrealistic. As a motivating example for why we consider multiple access to a hard disk a real threat, it is common for users to travel with their devices and lose direct possession of them on multiply occasions (checking bags, leaving them in a hotel room, etc.).

Consequently, we design a system whereby a user can plausibly deny the existence of a hidden volume even if the adversary has been able to take several snapshots of their disk and knows the password for the main volume. The reason that TrueCrypt does not maintain security against multiple snapshots is that it makes no attempt to hide the *pattern of accesses* that the user makes to the disk. That, combined with the fact that the hidden volume is stored separately from the main volume (in the free blocks of the main volume), gives the adversary a large advantage. An adversary can compare separate snapshots and see if a large number of "free" blocks have changed values. This would indicate that they are actually encrypted blocks that are part of a hidden volume, since they would otherwise not have a reason to change spontaneously.

These weaknesses lead us to our first observation: a system that is secure against multiple snapshots must make some attempt to hide the user's access pattern. There are several ways to do this, chief among them being Oblivious RAM [7]. We show how ORAM can be used to create exactly such a solution. Yet, a straightforward ORAM application comes with a significant read/write overhead. This motivates us to introduce a more efficient ORAM that is *write-only*, but provides sufficient security for our purposes and has a significantly lower overhead than related work. Based upon this write-only ORAM, we finally present HIVE, a new scheme for Hidden Volume Encryption.

The technical highlights of this paper are:

- The first formal treatment for hidden volume encryption and security, including multiple adversarial models. In addition, we show that several intuitive notions of security against a strong adversary are impossible to achieve.
- HIVE, a new solution which provides hidden volume encryption and provably achieves our notions of security using any write-only ORAM.
- A novel write-only ORAM construction which achieves optimal (constant) communication complexity. This result is of independent interest.
- An implementation of HIVE in the Linux kernel that is practical. Our implementation realizes hidden volumes as regular Linux block devices. We evaluate our implementation, and our benchmarks show that HIVE is both efficient and usable. The source code is available for download [2].

Plausibly Deniable Encryption: The hidden volume functionality offered by HIVE, TrueCrypt, and others [5, 9, 17] is also called plausibly deniable encryption. Yet, in this paper, we refrain from using this term, not to confuse it with the slightly different ideas of *deniable encryption* by Canetti et al. [3], see Section 7. Instead, we use *hidden volume encryption* in this paper.

2. ROBUST HIDDEN VOLUMES

We start by introducing the system and adversary model for hidden volume encryption. We envision a typical scenario with a user $\mathcal{U}$ having read/write access to a block storage device, e.g., a hard disk, USB stick or a network block device. User $\mathcal{U}$ is running a special software for hidden volume encryption. This software gives access to a sequence of independent *volumes* V_i that are mapped to the underlying storage device. To get access to these volumes, $\mathcal{U}$ knows a sequence of passwords $\mathcal{P}$. We stress that each $P_i \in \mathcal{P}$ gives *full* access to hidden V_i, i.e., encryption keys used in V_i are derived from passwords P_i, and $\mathcal{U}$ chooses the P_i carefully. Although passwords have notorious security issues, we simplify the exposition and assume that each P_i is chosen securely and can be used to derive a key with at least s bits of entropy, where s is a security parameter [11].

2.1 Model

One of our main contributions is the first formalization for hidden volume encryption. A hidden volume encryption scheme Σ provides an interface to access max volumes. We generalize to max volumes instead of two for increased flexibility. Each volume V_i has a password P_i associated with it and holds n_i blocks of data, each of size B. The size of the entire disk is N blocks. For simplicity, both volume blocks and hard disk blocks have the same size B. Usually, $B=512$ Byte, but B can be varied up to 4096 Byte as discussed later. We do not try to hide the value of max, but rather allow the user, $\mathcal{U}$, to choose some number of volumes $\ell \leq max$ which he will be actively using. It is this choice ℓ that will be hidden. Thus a hidden volume encryption scheme Σ works in such a way that an adversary, seeing changes to the blocks of a hard disk and knowing one or more passwords, has some uncertainty about ℓ. We also assume that the user has t blocks of RAM which are not visible to the adversary. To avoid the trivial solution of storing everything out of sight of the adversary, we constrict the size of the RAM to be much smaller than the size of the disk (i.e., logarithmic in N).

Typically, a hidden volume encryption scheme Σ is embedded in an operating system. There, it provides the functionality of a block device driver, yet it resides on top of an underlying (hardware) block device which we now call *the disk* for simplicity.

DEFINITION 1 (HIDDEN VOLUME ENCRYPTION Σ). *Let s denote the security parameter, t denotes the number of available RAM blocks, and $\mathcal{P} = < P_1,\ldots,P_\ell >$ denotes the sequence of user passwords. A hidden volume encryption (HVE) scheme Σ comprises the following algorithms.*

- $\mathsf{HVESetup}(s,t,\mathcal{P},B,< n_1,\ldots,n_\ell >)$*: Using parameters s, t, the sequence of passwords $\mathcal{P}$, block size B, and the size of each volume n_i, this algorithm generates volumes $< V_1,\ldots, V_\ell, V_{\ell+1},\ldots,V_{max} >$.*
- $\mathsf{HVEWrite}(b,d,i,\mathcal{P})$*: Using passwords $\mathcal{P}$, if $i \leq \ell$, then this algorithm stores data d at block index b in volume V_i, where V_i was output by* $\mathsf{HVESetup}(s,\mathcal{P})$.
- $\mathsf{HVERead}(b,i,\mathcal{P})$*: Using passwords $\mathcal{P}$, if $i \leq \ell$, then this algorithm returns data from the block indexed by b in volume V_i, where V_i was output by* $\mathsf{HVESetup}(s,\mathcal{P})$.

DEFINITION 2 (SOUNDNESS). *Hidden volume encryption scheme Σ is called sound,* iff *for any sequence of* $\mathsf{HVERead}$ *and* $\mathsf{HVEWrite}$ *operations, the last* $\mathsf{HVEWrite}(b,d,i,\mathcal{P})$ *to block b in volume V_i, $i \leq \ell$, implies $d =$* $\mathsf{HVERead}(b,i,\mathcal{P})$.

To allow a scheme Σ to read from and to write to the disk, we assume availability of regular read and write system calls. In the rest of this paper, we will use $\mathsf{DiskRead}(\beta)$ to denote a read from block index β of the disk and $\mathsf{DiskWrite}(\beta,d)$ to denote a write of data d to block β. Again for simplicity, we assume that a scheme Σ creates, for each volume, a new virtual block device within the operating system, which can be formatted with a file system and used just like a regular device. Informally speaking, a scheme Σ has to 1) translate OS reads and writes to one of the block devices (volumes) into calls of $\mathsf{HVERead}$ and $\mathsf{HVEWrite}$, and 2) apply its logic to finally use $\mathsf{DiskRead}(\beta)$ and $\mathsf{DiskWrite}(\beta)$. Note that b denotes the index of a virtual block in one of the volumes, while β denotes the index of a physical block of the disk.

2.2 Security Definitions

We now formalize security for hidden volume encryption. To start, we define an *access* $o = (\mathsf{op}, b, V, d)$. If $\mathsf{op} = \mathsf{write}$, then this access is a write to block b with value d in volume V, and if $\mathsf{op} = \mathsf{read}$, then this access is a read of block b in volume V which returns data value d. We call the sequence of accesses $O = < o_1, \ldots, o_n >$ the access *pattern*. We also allow $o = \perp$, which is a "null" operation that is simply ignored.

Our formalization of different security levels uses standard game-based definitions. All games Γ will be played between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ running a scheme Σ. All games Γ will adhere to the following *generic* game. We present the specific differences between the games and corresponding levels of security after the generic game.

In our games, an adversary $\mathcal{A}$ is allowed to repeatedly retrieve *snapshots* of the disk. We define a snapshot as the entire contents of the disk (i.e., addresses and current values of every block), and a snapshot is meant to represent a dump or capture of the hard drive from a machine.

The generic game $\Gamma^{\text{generic}}_{\mathcal{A},\Sigma}(s)$, cf. Fig. 1, is defined as:

1. Adversary $\mathcal{A}$ chooses $\ell \leq max$ and sends it to $\mathcal{C}$.

$\mathcal{A}$		$\mathcal{C}$
chooses $\ell \le max$	$\xrightarrow{\ell}$	
		$b \xleftarrow{\$} \{0,1\}$ $\forall 1 \le i \le \ell$, $P_i \xleftarrow{\$} \{0,1\}^s$ $\Sigma_0 \leftarrow \mathsf{HVESetup}(s,<P_1,...,P_\ell>)$ $\Sigma_1 \leftarrow \mathsf{HVESetup}(s,<P_1,...,P_{\ell-1}>)$
	$\xleftarrow{<P_1,...,P_{\ell-1}>,D_0}$	$D_0 = \mathrm{InitialSnapShot}_{\Sigma_b}$
Chooses accesses $O_{0,1}$ and $O_{1,1}$	$\xrightarrow{O_{0,1},O_{1,1},d_1}$	
		$D_1 \leftarrow \mathsf{Execute}_{\Sigma_b}(O_{b,1})$
	$\xleftarrow{D_1}$	If $d_1 = 1$, sends D_1
	... Let $p = poly(s)$	
Chooses accesses $O_{0,p}$ and $O_{1,p}$	$\xrightarrow{O_{0,p},O_{1,p},d_p}$	
		$D_p \leftarrow \mathsf{Exec}_{\Sigma_b}(O_{b,p})$
	$\xleftarrow{D_s}$	If $d_p = 1$, sends D_p
$\mathcal{A}$ outputs b'		

The outcome of this game is 1, *iff* $b = b'$, and 0 otherwise.

Figure 1: Security Game $\Gamma^{\mathrm{generic}}_{\mathcal{A},\Sigma}(s)$

2. Using security parameter s, $\mathcal{C}$ chooses ℓ passwords and a random bit b. $\mathcal{C}$ initializes two different hidden volume encryption schemes: Σ_0 (with ℓ passwords) and Σ_1 (with $\ell - 1$ passwords). Finally, $\mathcal{C}$ sends passwords $<P_1,...,P_{\ell-1}>$ and an initial snapshot D_0 of the disk to $\mathcal{A}$.
3. $\mathcal{A}$ chooses two accesses and sends them to $\mathcal{C}$, along with a bit d that specifies whether $\mathcal{A}$ would like a snapshot of the disk after execution. Both the access patterns and bit d will adhere to specific restrictions that we detail below.
4. Following scheme Σ_b, $\mathcal{C}$ "executes" one of the two accesses, depending on bit b.
5. If $d = 1$, $\mathcal{C}$ sends a snapshot D_i of the disk to $\mathcal{A}$.
6. Repeat steps 3 to 5 for $p = poly(s)$ times (*rounds* i). $O_{i,j}$ denotes access $j \in \{0,1\}$ in round i, $1 \le i \le poly(s)$. E
7. $\mathcal{A}$ outputs bit b'. The outcome of $\Gamma^{\mathrm{generic}}_{\mathcal{A},\Sigma}(s)$ is 1, *iff* $b' = b$.

From this general game, we develop two axes along which we can define varying levels of security: 1) the frequency with which $\mathcal{A}$ can access snapshots of the disk (regulating when and how often d_i can be 1 in our game above), and 2) the restriction applied to the two access patterns O_0,O_1 that $\mathcal{A}$ chooses and submits to $\mathcal{C}$. We denote by $\Gamma^{x,y}_{\mathcal{A},\Sigma}(s)$ a game that follows generic game $\Gamma^{\mathrm{generic}}_{\mathcal{A},\Sigma}(s)$, but implies restrictions "x" (defined below) on the frequency of snapshots and restrictions "y" (defined below) on the access patterns.

DEFINITION 3 (HIDDEN VOLUME ENCRYPTION). *A hidden volume encryption scheme Σ is secure regarding restrictions (x,y),* iff *for any probabilistic polynomial time adversary $\mathcal{A}$ there exists a function $\epsilon(s)$ that is negligible in security parameter s, such that*

$$Pr[\Gamma^{x,y}_{\mathcal{A},\Sigma}(s) = 1] \le \frac{1}{2} + \epsilon(s).$$

In conclusion, the rationale behind this game-based definition is to prevent adversary $\mathcal{A}$ from successfully guessing whether there exist $\ell - 1$ or ℓ volumes in use, even when $\mathcal{A}$ is able to choose ℓ. We allow $\mathcal{A}$ to specify two patterns O_0,O_1, similar to classic indistinguishability proofs, which allows us to capture a stronger chosen plaintext adversary. The restrictions we place on these access patterns will lead to an intuitive understanding of the security our scheme provides.

Table 1: Different Notions for Hidden Volume Encryption

	Restricted Hiding	Opportunistic Hiding	Plausible Hiding
Arbitrary	Impossible	HIVE-B	HIVE
On-Event	Reencryption	HIVE-B	HIVE
One-Time	TrueCrypt, HIVE	TrueCrypt, HIVE	TrueCrypt, HIVE

2.2.1 Snapshot Frequency

We consider three different adversarial capabilities in terms of snapshotting:

1. Arbitrary $\Gamma^{\mathsf{Arbitrary},(\cdot)}_{\mathcal{A},\Sigma}$: $\mathcal{A}$ can obtain snapshots after every operation to the disk (any number of d_i can be 1).
2. On-Event $\Gamma^{\mathsf{On-Event},(\cdot)}_{\mathcal{A},\Sigma}$: $\mathcal{A}$ can obtain snapshots after $\mathcal{U}$ has run an "unmount" operation. We represent this by having an additional function $\mathsf{Unmount}$ which is called after $\mathsf{Execute}$, whenever $d_i = 1$, to generate the snapshot.
3. One-Time $\Gamma^{\mathsf{One-Time},(\cdot)}_{\mathcal{A},\Sigma}$: $\mathcal{A}$ can obtain only a single snapshot ($d_i = 1$ for only a single value of i).

Our justification for also considering an "on-event" adversary is that, in reality, some adversaries will not have the capability to take arbitrary snapshots of the disk while it is in use. More likely, the machine will be confiscated or compromised while $\mathcal{U}$ does not have the hidden volume mounted. Therefore, a model where the adversary can only take snapshots after an "unmount" is interesting.

2.2.2 Access Pattern Choice

An important part of our formal security is allowing the adversary to *choose* some part of the access pattern. This represents the fact that, in reality, an adversary may have partial knowledge of $\mathcal{U}$'s access pattern. Informally, allowing adversary $\mathcal{A}$ to choose the access pattern will guarantee that no matter what *a priori* knowledge $\mathcal{A}$ might have, they still cannot learn the secret we are trying to protect (i.e., whether there is a volume still unrevealed). To formalize this, we allow $\mathcal{A}$ to choose a value ℓ, between 1 and max which represents the number of volumes in use by $\mathcal{U}$. To maintain some uncertainty about exactly how many volumes are actually being used, we then allow $\mathcal{A}$ to also choose two access patterns, one which includes accesses to volume ℓ and one which does not.

We stress that restrictions on access pattern choices are mandatory to prevent "trivial" impossibility. For instance, in our model $\mathcal{A}$ will get the passwords for all volumes up to $V_{\ell-1}$. If the two patterns contain different writes to these volumes, then the resulting snapshots will allow $\mathcal{A}$ to distinguish between them easily (i.e., he simply decrypts these volumes and checks which values were written). This is to be expected, since the point of hidden volume encryption is to protect accesses to the *secret* volumes which may or may not exist, from the adversary's perspective. Once a password is given up, we cannot hide the contents any longer.

Therefore, the first restriction which must exist for all of our definitions is that any access where both patterns write to a volume V_1 through $V_{\ell-1}$ must contain *identical writes* in both patterns chosen by adversary $\mathcal{A}$.

We now present three settings, along with a justification for why they are useful analogues for the real world. The restrictions in each settings are presented as $\mathcal{A}$ being able to choose an access pattern

O_0 and then a second pattern O_1 that must constrain to some restrictions based on O_0. Intuitively, the restrictions on O_0 govern which types of access patterns are protected, and the restrictions on O_1 express the level of security that is achieved.

Restricted Hiding: $\Gamma^{(\cdot),\mathsf{Restricted}}_{\mathcal{A},\Sigma}$.

The first setting we consider is the most straightforward: in each round, we allow $\mathcal{A}$ to choose *any* access pattern $O_0 =< o_{0,1},\ldots, o_{1,n} >$. As stated before, if access $o_{0,i}$ is a write in volume V_j, $j \leq \ell - 1$, then access $o_{1,i}$ in pattern O_1 must be *equal* to $o_{0,i}$. If $o_{0,i}$ is in V_ℓ, then $o_{1,i}$ must be $\perp$ (the null operation, it is simply ignored by $\mathcal{C}$). The access pattern executed when $b = 1$ is then the same as when $b = 0$, with all accesses to V_ℓ ignored by $\mathcal{C}$.

Since the difference between O_0 and O_1 is only the removal of all accesses to V_ℓ, a Restricted Hiding scheme effectively prevents an adversary from distinguishing between the case where a user uses ℓ volumes and the case where a user only uses $\ell - 1$ volumes, with no restriction on the user's access pattern. This means that a scheme with Restricted Hiding would be the ideal system, since it would protect any access pattern and an adversary would not be able to learn whether there are ℓ or $\ell - 1$ volumes in use. Unfortunately, this level of security is difficult to achieve.

LEMMA 1. *Let n^* be the number of blocks of volume V_ℓ. There is no scheme Σ that is $\Gamma^{\mathsf{Arbitrary},\mathsf{Restricted}}_{\mathcal{A},\Sigma}$-secure and requires less than n^* blocks of RAM.*

PROOF (COUNTEREXAMPLE). $\mathcal{A}$ submits two access patterns of length n^*, O_0 and O_1, where $O_{0,i} = (\mathsf{write}, i, \ell, r_i)$, $O_{1,i} = \perp$ and $r_i \xleftarrow{\$} \{0,1\}^B$. In case $\mathcal{A}$ observes changes to the disk, $\mathcal{A}$ outputs 1. Otherwise, $\mathcal{A}$ outputs 0. If $\mathcal{C}$ executes O_1, nothing on the disk will change because $\perp$ operations are ignored. If $\mathcal{C}$ executes O_0, then either the disk must change or $\mathcal{C}$ must have at least n^* blocks or RAM to hold all the writes from O_0. □

This essentially means that against an adversary with Arbitrary snapshotting capability you can do no better than storing all the hidden volumes in RAM, which is quite unrealistic.

Although none of this paper's solutions are directly targeted at On-Event security, we include it in our definitions because it is a distinctly separate adversarial model from Arbitrary and One-Time. It has at least one interesting property: even though it is impossible to achieve $\Gamma^{\mathsf{Arbitrary},\mathsf{Restricted}}_{\mathcal{A},\Sigma}$ security, it is quite simple (albeit inefficient) to obtain $\Gamma^{\mathsf{On-Event},\mathsf{Restricted}}_{\mathcal{A},\Sigma}$ security. If a TrueCrypt-like approach is taken except, upon unmount, every data block is *reencrypted* and every empty block is filled with a uniformly random string, we achieve this security. This is highly inefficient, but we point it out as an interesting observation on our various adversarial models and possible motivation for more efficient solutions in the future.

Opportunistic Hiding: $\Gamma^{(\cdot),\mathsf{Opportunistic}}_{\mathcal{A},\Sigma}$.

The second setting we consider, opportunistic hiding, is similar to restricted hiding, but with a slightly more specific access pattern: again, if $o_{0,i}$ is in volume $V_j, j \leq \ell - 1$, then $o_{1,i}$ must be equal to $o_{0,i}$. Also, if $o_{0,i}$ is in V_ℓ, then $o_{1,i}$ will be $\perp$. The additional restriction is that *between snapshots*, every write to a volume V_2 through V_ℓ in O_0 must have a corresponding read to volume V_1 which occurs after it. More formally, the additional restriction says that for $O_0 = (o_1, \ldots, o_n)$, there must exist a one-to-one mapping $f : [1 \ldots n] \rightarrow [1 \ldots n]$ such that if $O_{0,i}$ is a write to a volume higher than V_1, then $f(i) > i$, $O_{0,f(i)}$ is a read from volume V_1.

What this effectively means is that the system will hide writes to volumes higher than V_1 by executing them *simultaneously* with reads to V_1. This reflects the idea that, if we make execution of reads and writes similar, there will be extra capacity during a read to simultaneously do a write. We believe this that is not a very onerous constraint, because it is very reasonable that the "secret" volumes will be accessed much less frequently than the lowest volume, which is known to exist to the adversary.

This security definition also provides "complete" security to the user, in that an adversary should not be able to distinguish whether there are $\ell - 1$ or ℓ volumes. In that way, it is similar to our first definition, however, in order for the user to receive this security, it requires that the access pattern O_0 must have fewer accesses to V_2 through V_ℓ than V_1. In practice this means that the user must access volume V_1 more often than the other, more secret volumes.

Plausible Hiding: $\Gamma^{(\cdot),\mathsf{Plausible}}_{\mathcal{A},\Sigma}$.

Finally, we consider an even more restricted setting where writes to V_ℓ can be plausibly denied as operations to the other volumes V_1, $\ldots, V_{\ell-1}$. $\mathcal{A}$ may choose any access $O_{0,i}$. Again, if $O_{0,i}$ is a write in volume $V_j, j \leq \ell - 1$, then $o_{1,i}$ must be equal to $O_{0,i}$. Otherwise, we only require that neither pattern contains $\perp$ operations (so their "true" lengths are equal).

The intuition of this security definition is that, if an access pattern contains writes to V_ℓ, there is always a plausible access pattern containing only accesses to V_1 through $V_{\ell-1}$ (with writes to V_ℓ replaced by reads to other volumes) which would have produced the same sequence of disk operations. Therefore an adversary can never be sure of the existence of V_ℓ. Additionally, all writes to V_ℓ will be indistinguishable from each other. In contrast with the previous two definitions, we have restrictions on O_1 beyond that the accesses to lower volumes must be equal.

Finally, Table 1 summarizes our two orthogonal axes of hidden volume encryption. Our new scheme, HIVE, provides plausible hiding with arbitrary snapshots. A variant of HIVE, HIVE-B, provides opportunistic hiding with arbitrary snapshots (as well as weaker notions). TrueCrypt and other related work [1, 5, 9, 13, 17, 20] only provide security against one-time snapshots [4]. Additionally, as noted above, one can achieve on-event, restricted security with an expensive reencryption technique. Note that Arbitrary ⇒ On-Event ⇒ One-Time, and Restricted ⇒ Opportunistic.

We would like to stress that any hidden volumes scheme needs additional requirements to maintain security. For example, the OS and applications should not keep any trace of previously mounted hidden volumes, e.g. in a "recently opened documents" list. For a detailed discussion, we refer to Czeskis et al. [4] and [19].

3. TOWARD A GENERIC HIDDEN VOLUME ENCRYPTION

Current hidden volume encryption solution only protect against One − Time adversaries. We now present a first "generic" protocol that offers stronger, more robust security against One-Time and Arbitrary adversaries. This generic protocol is based on ORAM.

3.1 ORAM Preliminaries

An ORAM provides three operations: $\mathsf{ORAMSetup}(n, B, k, s)$, $\mathsf{ORAMRead}(i)$, and $\mathsf{ORAMWrite}(i, d)$. It is a block-based data structure storing n blocks of size B bits, each indexed by a $\log_2 n$ bit integer, using a key κ for encryption, and s is the security parameter. This data structure is backed by a simple key-value store. As shorthand, we write $\mathsf{Execute}(O)$ to be the reads and writes induced on the underlying data store by the ORAM executing an access pattern O. We refer to the large body of related work for more details, e.g., seminal work by Goldreich and Ostrovsky [7] or recently Shi et al. [16] or Stefanov et al. [18].

```
Input: Security parameter $s$, RAM size
       $t$, passwords $\mathcal{P}$, block size $B$, volumes sizes $<n_1,...,n_\ell>$
Output: Volumes $<V_1,...,V_\ell,V_{\ell+1},...,V_{max}>$
largestVolume $:=$ maximum$(n_1,...,n_\ell)$;
for $i := 1$ to $max$ do
    $V_i :=$ ORAMSetup(largestVolume,$B$,$P_i$,$s$);
    return $<V_1,...,V_{max}>$;
end
```

Algorithm 1: Generic HVESetup$(s,t,\mathcal{P},B,<n_1,...,n_\ell>)$

```
Input: Block $b$, data $d$, volume index $i$, passwords $\mathcal{P}$
ORAM_i.ORAMWrite$(b,d)$;
foreach $j \neq i$ do
    $r \xleftarrow{\$} \{1,...,$sizeof(ORAM$_j$)$\}$;
    // Using password $P_j$ as key
    dummy $:=$ ORAM$_j$.ORAMRead$(r)$;
    ORAM$_j$.ORAMWrite$(r$,dummy$)$;
end
```

Algorithm 2: Generic HVEWrite$(b,d,i,\mathcal{P})$

```
Input: Block index $b$, volume $i$, passwords $\mathcal{P}$
Output: Data $d$
$d :=$ ORAM$_i$.ORAMRead$(b)$;
forall the $i$ do
    $r \xleftarrow{\$} \{1,...,$sizeof(ORAM$_i$)$\}$;
    // Using password $P_i$ as key
    dummy $:=$ ORAM$_i$.ORAMRead$(r)$;
    ORAM$_i$.ORAMWrite$(r$,dummy$)$;
end
return $d$;
```

Algorithm 3: Generic HVERead$(b,i,\mathcal{P})$

DEFINITION 4 (ORAM SECURITY). *An ORAM is secure* iff*, for any probabilistic polynomial time adversary $\mathcal{A}$ and any two access patterns O_0 and O_1 of the same length, there exists a function ϵ negligible in security parameter s, such that*

$$|Pr[\mathcal{A}(\mathsf{Execute}(O_0)) = 1] - Pr[\mathcal{A}(\mathsf{Execute}(O_1)) = 1]| \leq \epsilon(s).$$

The goal of our first, "generic" hidden volume encryption is twofold: to hide the pattern of writes issued by the user and to give a plausible reason to access the disk when writing into hidden volumes. Broadly, we accomplish the first by using ORAM and the second by making a read to a volume cause a "dummy write". This dummy will cover for a write to a hidden volume, and is in line with the idea that all operations should "look the same" regardless of which volume it is in, or even if it is a read or write.

3.2 Generic Construction

To start, we use max separate ORAMs ($\mathrm{ORAM}_{1 \leq i \leq max}$), each holding the data for a single volume and encrypted with its own password, e.g., we simply run ORAMSetup for each of these ORAMs. The only requirement we have on our ORAM scheme is that it be efficiently simulatable, i.e., a simulator S, without the password or any knowledge of the access pattern beyond its length can output a series of disk accesses which are indistinguishable from those output by an actual ORAM. Additionally, this simulator needs to be stateless, in that its output for each operation cannot depend on anything other than the output from the previous operations. Fortunately, typical ORAM constructions meet this definition. So, for $j > \ell$, ORAM_j is replaced by a simulator S. Therewith, an ORAM can execute a "dummy" operation containing no information, which, to all adversaries, looks identical to a real operation. This is necessary so that the ORAMs not actually in use by the user will not even *have* keys that could be revealed to the adversary. This gives the user deniability, since they can reasonably claim that no key exists for a certain volume.

When the system executes a write to volume i (Algorithm 2), it executes that write on ORAM_i. For all $j \neq i$, it picks a random block in V_j and writes its same data value back to ORAM_j, i.e., a "dummy" operation which changes no data. When the system executes a read (Algorithm 3), it reads from the respective ORAM and then does a dummy write for all ORAMs (or executes the simulator, for volumes greater than ℓ). The idea is that, if an adversary does not have a key to volume i, they cannot tell whether we are reading from some volume (maybe i, but maybe not) or writing to volume i. For the internal encryption and decryption within ORAM_i, we use password P_i as the key.

THEOREM 1. *If (*ORAMSetup, ORAMRead, ORAMWrite*) is a simulatable ORAM, then our* generic *hidden volume encryption is a $\Gamma^{\mathsf{Arbitrary},\mathtt{Plausible}}_{\mathcal{A},\Sigma}$-secure hidden volume encryption.*

PROOF. Under Plausible Hiding security, the access patterns given by $\mathcal{A}$ will differ only when neither $O_{0,i}$ nor $O_{1,i}$ is a write to a volume less than ℓ. Since $\mathcal{A}$ cannot distinguish on operations of the access patterns that are identical, we only need to show that, for operations that differ, $\mathcal{A}$ cannot distinguish. So, if $O_{0,i}$ and $O_{1,i}$ differ, then they can each be either a read to any volume or a write to a volume $j \geq \ell$. By the security definition of ORAM, a "dummy" write in a volume $j \geq \ell$ is not distinguishable from an actual write with probability greater than $1/2 + \epsilon(s)$. This implies that a read to one of these volumes cannot be distinguished from a write. We also have that for all i, j, a read to V_i is indistinguishable from a read to V_j. Therefore, since $\mathcal{A}$ cannot distinguish between the outputs with probability greater than $1/2 + \epsilon(s)$, they cannot win the game with any non-negligible advantage. □

3.3 Opportunistic Hiding Security

Opportunistic security gives more freedom to the user, since it does not require them to "pretend" that they did some reads that were actually writes in other volumes. Thus, it is relatively simple to achieve. Instead of writing a block immediately when the user wants to, if it is part of a volume $V_i, i > 1$, we add it to a queue Q_i. Every time the user does an operation on V_1 (read or write), for all $i > 1$ if Q_i is not empty, we write one block from Q_i to V_i, instead of doing the dummy write. Reads to volumes other than V_1 are trivial, i.e., we read the requested block, but do not change anything on the disk. Writing during such a reads is no longer necessary, because writes to all volumes higher than V_1 are simultaneously hidden by accesses to V_1.

THEOREM 2. *If (*ORAMSetup, ORAMRead, ORAMWrite*) is a simulatable ORAM, then this* modified generic *hidden volume encryption is $\Gamma^{\mathsf{Arbitrary},\mathtt{Opportunistic}}_{\mathcal{A},\Sigma}$-secure.*

PROOF. O_0 and O_1 are only different when $O_{0,i}$ is an operation in V_ℓ. If $O_{0,i}$ is a read, it will be identical to a $\perp$ operation in our *modified generic* scheme. Such a read does not trigger any write. If $O_{0,i}$ is a write, it equally triggers no writes, as it just adds a block to the Q_ℓ, and again is indistinguishable. Therefore, we only need to show that a read to V_1 which also writes from Q_ℓ is indistinguishable from one which just causes a dummy operation in V_ℓ. This follows directly from the security of our ORAM, and so $\mathcal{A}$ cannot win the game with any non-negligible advantage. □

4. WRITE-ONLY ORAM

We have proven a generic hidden volume encryption secure using ORAM as a building block. Note, however, that the snapshots $\mathcal{A}$ gets in our security game do not include any information about block *reads* that occur to the disk. This reflects the idea that an adversary can see the impact of block writes, in the form of modified data, but they cannot see where or how often user U reads. Typically, block reads do not leave any discernible trace on the disk.

This means that the ORAMs we are using are actually more powerful than we need them to be. In fact, an ORAM which only hides writes to the disk is sufficient. We can define the security of such a write-only ORAM as follows:

DEFINITION 5 (WRITE-ONLY ORAM SECURITY). *Let sequence* $\mathsf{ExecW}(O)$ *be the sequence of* writes *caused to the disk when an ORAM executes access pattern O. A write-only ORAM (with algorithms* $\mathsf{ORAMSetup}, \mathsf{ORAMRead}, \mathsf{ORAMWrite}$*) is secure* iff*, for any probabilistic polynomial time adversary* $\mathcal{A}$ *and any two access patterns* O_0 *and* O_1 *that contain the same number of writes there exists a function* ϵ *negligible in security parameter s, such that*

$$|Pr[\mathcal{A}(\mathsf{ExecW}(O_0))=1]-Pr[\mathcal{A}(\mathsf{ExecW}(O_1))=1]| \leq \epsilon(s).$$

There has been limited work on write-only ORAMs until now. There are several schemes by Li and Datta [12], but they all have significant drawbacks. They are able to obtain an amortized write communication complexity of $O(B \cdot \log n)$, but only at the cost of reads being in $O(B \cdot n)$. As ORAM communication complexities directly relate to the hidden volume encryption overhead, this does not suit our needs. Applications might perform as many reads as writes, and reads would quickly become too expensive for increasing n. For efficient reads, Li and Datta [12] require either memory or communication complexity to be polynomial in n. We stress that their schemes only provide *amortized* complexity guarantees, with worst-case complexity being polynomial in n.

Since we target good read and write performance for our hidden volume encryption to be useful, we now present a new write-only ORAM which achieves worst-case constant communication complexity and only poly-logarithmic memory requirements. We will start with a very simple, inefficient construction and show how its shortcomings can be addressed one at a time until we have our final, efficient construction HIVE.

4.1 Basic Write-Only ORAM Construction

We now present our new ORAM that only supports write operations. Our ORAM makes use of a mapping data structure Map that maps blocks from the ORAM to physical blocks (sectors) on the disk. For now, we will consider it as an associative array such that $\mathsf{Map}[b]$ contains the physical address β on the disk where ORAM block b is currently located. Later, we will show how this map is structured and actually implemented, but for now assume that it is simply stored in RAM and can be efficiently accessed. Also assume that the hard disk has at least twice the size of the ORAM, i.e., $N \geq 2 \cdot n$, so we have at least twice as much storage available as needed. All mapping entries $\mathsf{Map}[b]$ are set to $\perp$. For encryption and decryption, we employ any pair Enc, Dec of algorithms realizing $\mathsf{IND\$\text{-}CPA}$ encryption [15]. An $\mathsf{IND\$\text{-}CPA}$ encryption is an encryption that produces ciphertexts indistinguishable from random strings, e.g., AES in CBC or counter mode. The encryption makes use of secret key κ, only known to the ORAM user $\mathcal{U}$.

ORAMWrite: For an $\mathsf{ORAMWrite}(b,d)$ operation (see Algorithm 4), $\mathcal{U}$ picks a sequence $\mathcal{S}$ of k random, distinct hard disk block indices β_i, where k is a security parameter. $\mathcal{U}$ then randomly picks one index $\beta \in \mathcal{S}$ that is "free". Here, free means that there is no block in the ORAM that is mapped to disk block β. User $\mathcal{U}$ writes $\mathsf{Enc}_\kappa(d)$ at position β to disk. Of the $k-1$ remaining block indices, if their corresponding blocks contain encrypted data, $\mathcal{U}$ reencrypts their contents, and if the blocks are free $\mathcal{U}$ writes a random strings into them ("$\mathsf{ReencryptOrRandomize}$"). Finally, $\mathcal{U}$ updates the mapping for b. Since $\mathcal{U}$ picks the k block indices in $\mathcal{S}$ randomly and independently from b, and the encryption produces ciphertexts indistinguishable from random, an adversary seeing these k blocks change cannot learn anything about b or d.

```
Input: Block index $b$, data $d$
$\mathcal{S} := <\beta_i>$,
such that $\beta_i \xleftarrow{\$} \{1,\ldots,N\} \wedge 1 \leq i \leq k \wedge \forall u,v : \beta_u \neq \beta_v$ holds;
$\beta \xleftarrow{\$} \mathcal{S}$, such that $\beta$ is free;
$\mathsf{DiskWrite}(\beta, \mathsf{Enc}_\kappa(d))$;
$\mathsf{ReencryptOrRandomize}(\mathcal{S} \setminus \beta)$;
// Find out which block
   in the Map ORAM has the address we want
$M := \lfloor \frac{B}{\log N} \rfloor$ ;
$\mathrm{map_{block}} := \mathsf{Map.ORAMRead}(\lfloor \frac{b}{M} \rfloor)$ ;
// Retrieve the address from the map block
$\mathrm{map_{block}}[b \mod M] := \beta$ ;
$\mathsf{Map.ORAMWrite}(\lfloor \frac{b}{M} \rfloor, \mathrm{map_{block}})$;
```

Algorithm 4: $\mathsf{ORAMWrite}(b,d)$

```
Input: Block index $b$
$M := \lfloor \frac{B}{\log N} \rfloor$ ;
$\mathrm{map_{block}} := \mathsf{Map.ORAMRead}(\lfloor \frac{b}{M} \rfloor)$ ;
$\beta := \mathrm{map_{block}}[b \mod M]$ ;
return $\mathsf{Dec}_\kappa(\mathsf{DiskRead}(\beta))$;
```

Algorithm 5: $\mathsf{ORAMRead}(b)$

ORAMRead: Our ORAM does not aim at protecting read operations. Thus, reads are trivial as shown in Algorithm 5.

ORAMSetup: For initialization, we require only that the map be initialized to an "empty" state, e.g. looking up the address of any block should return a $\perp$ value indicating that the block has not been written in the system yet.

Choice of k: To guarantee that $\mathcal{U}$ always finds at least one free block in $\mathcal{S}$ to put data d into, $\mathcal{U}$ has to choose k sufficiently large. Since at least half the disk is empty, the probability that any randomly chosen block is free is at least $1/2$. Let X be the random variable that, when selecting k blocks uniformly from all N, describes the number of blocks among those k that are free. As N is typically large compared to k, we approximate the hypergeometrically distributed X with a binomial distributed X. $Pr[X \geq 1] = 1 - Pr[X=0] \approx 1 - \binom{k}{0} \cdot (\frac{n}{N})^k = 1 - 2^{-k}$ for $N = 2 \cdot n$. Therefore, if we set k equal to our security parameter s, the probability of *not* finding at least one free block for any single write will be negligible small in s with 2^{-s}. Setting N to twice n will "only" double storage requirements, but leads to high efficiency and practicality as we will show in Section 4.2.

Free Blocks: A remaining detail is how to determine which physical blocks β on the disk are actually "free", so no block b of the ORAM maps to them. The challenge is to do this in an efficient way in order to keep a low complexity. A simple solution is a reverse mapping that tags each block β on the disk with the index for the ORAM block that maps to it. For an operation $\mathsf{ORAMWrite}(b,d)$, what $\mathcal{U}$ actually writes to disk block β is $\mathsf{Enc}_\kappa(d) || \mathsf{Enc}_\kappa(b)$. Then, when $\mathcal{U}$ needs to determine if a disk block β is free, $\mathcal{U}$ decrypts β's content to restore ORAM address b and checks if $\mathsf{Map}[b] = \beta$. If that condition is *not* true, then it means that β is an "old" version of ORAM block b, so β is free, and it can be safely overwritten. Consequently, $\mathcal{U}$ can therewith check if a block is free in constant time.

In practice, we cannot write the two ciphertexts $\mathsf{Enc}_\kappa(d) || \mathsf{Enc}_\kappa(b)$ into a single block of size B, as $|d| = B$ already. Also, for each $\mathsf{IND\$\text{-}CPA}$ encryption, we need to store random coins such as an IV or counter. Consequently, we write the ciphertext of $\mathsf{Enc}_\kappa(d)$ into block β, and we write $\mathsf{Enc}_\kappa(b)$ and the encryptions' random coins into another *metadata* block β'. One can imagine that, in addition to the N hard disk blocks for the write-only ORAM, we need additional N blocks for metadata. Each time an ORAM block β is written to disk, the corresponding metadata block β' is also updated. As the mapping between an ORAM block and its metadata block is fixed, this does not have any consequences for security. A straightforward optimization of this idea is to store multiple IVs and multiple $\mathsf{Enc}_\kappa(b)$ for multiple ORAM blocks in a single metadata block β', as typically $|IV| + |\mathsf{Enc}_\kappa(b)| < B$.

In conclusion, $\mathcal{U}$ writes a block b into one of the k blocks from $\mathcal{S}$ chosen randomly and thus independently of b, updating Map accordingly. Since k is a security parameter independent from n, we only need to actually write a constant $O(1)$ number of blocks to disk for each ORAM write operation. Still, our write-only ORAM has two drawbacks: first, it requires us to access a large number of blocks per operation (k is, for example, 64), and second, it requires $O(B \cdot n \cdot \log N)$ memory complexity to store the Map.

We now move on to optimizing our initial write-only ORAM to allow for smaller k and for efficient storage of the map.

4.2 Stash-Optimized Write-Only ORAM

We have set k to be equal to a security parameter s. This is to ensure that with probability $1 - 2^s$ $\mathcal{U}$ will choose a free block to write new data into on every write operation. It turns out that this is quite wasteful, because in expectation, $\mathcal{U}$ will find $s/2$ free blocks per operation – many more than are necessary. We only have to set k so high to avoid a potential worst-case situation where $\mathcal{U}$ does not find any free blocks.

We now exploit that fact that typically there will be many more blocks free than just one. We set k such that in expectation we will have one (or slightly more than one) free blocks, e.g., by setting k to only 4. The intuition is that in that case, there will be many times where $\mathcal{U}$ does not find a free block, but there will even more times when $\mathcal{U}$ finds more than one free block. We can take advantage of this situation by having a small *stash* of pending blocks in memory. If $\mathcal{U}$ chooses k blocks, and none of them are free, then $\mathcal{U}$ puts the block to be written at that time in the stash, to be written later. If $\mathcal{U}$ ends up with more than one free block during a write, then $\mathcal{U}$ additionally writes out some extra blocks from the stash. Note that this idea and its analysis is different from the one by Stefanov et al. [18], where a stash is used in case one of the bucket ORAMs is full.

The first challenge is to bound the size of the stash, such that we do not overburden user $\mathcal{U}$'s memory, but still have high probability of the stash not overflowing. To determine the stash's size, we use a standard queueing argument. Our stash can be modeled as a $D/M/1$ queue with deterministic arrival rate $\gamma = 1$ and service times exponentially distributed with rate parameter $\mu = k/2$. As shown by Jansson [10], we can then express the steady state probability P of having i items in the stash at any time as $P = (1-\delta) \cdot \delta^i$, where δ is the root of the equation $\delta = 2^{-\mu\gamma(1-\delta)}$ with the smallest absolute value. If μ is larger than 1, then $\delta < 1$, and the steady state probability of having i blocks in the stash will be $O(2^{-i})$. That means that the size of the stash will be $O(B \cdot s)$ with probability all but negligible in s.

This optimization allows us a huge savings in performance. We can go from $k = s$ to k being a small constant, independent of the security parameter. For example, if $k = 3$, we can solve to find $\delta = 0.41718$, and we can bound the probability of overflowing the stash at 2^{-64} using only a stash of size 50 blocks. With a block/sector size of 4096 Byte, the stash would consume only 200 KByte RAM.

Security: The actual writing that $\mathcal{U}$ performs to the k blocks of $\mathcal{S}$ is hidden by the security of the $\mathsf{IND\$}$-$\mathsf{CPA}$ encryption. Therefore, whether $\mathcal{U}$ writes one, two, or no blocks from the stash on any given operation is not observable by the adversary. Consequently, the stash does not impact security at all.

4.3 Recursive map

As we have described our write-only ORAM thus far, we have good communication complexity, but at the cost of a large map that must be kept in memory. The total size of the map will be $O(B \cdot n \cdot \log N)$, prohibitively large if we hope to store it in memory. However, we can use a standard technique [12, 16] which is to recursively store the map in smaller and smaller ORAMs. If our block size B is at least $\chi \cdot \log N$ for some constant $\chi > 2$, then we are guaranteed that if we (recursively) store our map in another ORAM, that ORAM in turn will have a map that is no greater than half the size of the original map. Therefore, after $O(\log n)$ recursive ORAMs, we will have a map that is constant size and can be stored in memory.

This reduces the map to a much more comfortable size, but at the expensive of increasing the communication complexity: now, to access the map, we have to, in turn, access $O(\log n)$ recursive ORAMs. This will also slightly impact our stash analysis, since the ORAMs that hold the map do not have deterministic arrival rates, since their arrivals are in fact the result of service on the above queue. Fortunately, we can model them as $M/M/1$ queues with *expected* arrival rate equal to 1 which still results in exponentially distributed stationary probabilities [8], and we end up with the same $O(\log n)$ bound on the size.

As seen in algorithms 5 and 4, we can then simply treat the map for each level as another ORAM and issue $\mathsf{ORAMRead}$ and $\mathsf{ORAMWrite}$ calls as necessary. Each level is unaware of how the next level structures itself, only that it provides an interface to read and write, and that it can do so securely. However, we cannot treat the map as a simple associative array now because the recursive ORAM will need to store more than one address per block in order to guarantee that we have only $O(\log n)$ levels of recursion. Therefore, to read an entry from the map, we have to calculate $N = \lfloor \frac{B}{\log N} \rfloor$, the number of entries we can fit per block, then figure out which block the entry we want will be in. For instance, if we want the address for block i, we would get block $\lfloor \frac{i}{n} \rfloor$ from the map, and read the $i \mod N$th entry from it.

4.4 Write Complexity

For each $\mathsf{ORAMWrite}$ operation, our write-only ORAM reads k random blocks, checks if they are empty, fills from zero to $k-1$ of them with blocks from our stash, and updates the map accordingly.

The initial reading of the k blocks from the map (and hence checking if a block is empty) requires reading one block from each of $\log n$ recursive levels. Therefore, we can accomplish that first step with $k \cdot O(B \cdot \log n) = O(B \cdot \log n)$ communication complexity. However, when we then *write* to the map, each recursive ORAM has to, itself, read from all the recursive ORAMs "below" it in order to read its own map. Thus, these writes costs $O(B \cdot \log^2 n)$, bringing the total complexity for an $\mathsf{ORAMWrite}$ to $O(B \cdot \log^2 n)$.

This overhead would be disappointing, as there are several full-functionality (read and write) ORAMs which provide the same overhead. However inspired by Stefanov et al. [18], we can use the following optimization to reduce the cost of the expensive map accesses. If we set the size of only the data blocks on our disk to $B = \Omega(\log^2 n)$, the total size of $\log n$ blocks in the recursive map is no greater than the size of one B-sized data block at the "top level". What we end up with is non-uniform block sizes: top level blocks (actual data blocks) have size $\Omega(\log^2 n)$, and blocks that are part of the map have only size $\chi \log n$ for some constant $\chi \geq 2$. We are still guaranteed that the map will have $O(\log n)$ levels, but we reduce the communication complexity of a map operation by a factor of $O(\log n)$. Consequently, in terms of communication complexity, reading from the map is constant in $O(B)$, and updating/writing is in $O(B \cdot \log n)$. We can apply the same optimization again, as long as $B = \Omega(\log^3 n)$. In terms of communication for an $\mathsf{ORAMWrite}$, we reduce total complexity to $O(B)$.

In conclusion, our write-only ORAM features $O(B \cdot s)$ memory and constant $O(B)$ communication complexity.

4.5 Security Analysis

Security for our scheme follows directly from the fact that we only write to uniformly randomly chosen blocks at each level of the ORAM. Since the blocks we choose are independent of the addresses in the user's access pattern, they cannot reveal any information about it to the adversary. Additionally, all the data we write is freshly encrypted with a semantically secure encryption, so the data itself cannot reveal any information.

Simulator: We note in the previous section that, to be useful in our scheme, an ORAM needs to be (efficiently) simulatable. Fortunately, such a simulator S is simple to construct for our scheme. S proceeds in every operation to change k uniformly random blocks at each level of the recursion to fresh random strings. Since, in normal operation of our ORAM, we will access k random blocks at each level, and those blocks will be filled with either random strings or $\mathsf{IND\$\text{-}CPA}$ encryptions indistinguishable from random, S will be indistinguishable from an actual execution of our ORAM.

5. PRACTICAL HIDDEN VOLUME ENCRYPTION WITH HIVE

Thus far we have presented a generic hidden volume encryption scheme which uses a write-only ORAM as a building block and has constant communication complexity per access. We now present HIVE that builds upon this idea and makes it practical. We start by addressing an important consideration that we must take into account when designing a practical, real-world system.

5.1 Uniform vs. Non-uniform Blocks

Current storage devices, e.g., today's hard disks, have fixed size blocks (sectors). This means that we cannot use our non-uniform block optimization from the previous section, unless we wanted to use the base device blocks as "small" blocks and combine many of these blocks together to make "large" blocks. However, most systems use either 512 or 4096 byte blocks, so there is not much room for optimizing these parameters.

Fortunately, although we cannot easily obtain optimal $O(B)$ complexity with uniform blocks, our write-only scheme scheme is still substantially more efficient than the currently most efficient full-functionality ORAMs, e.g., Path ORAM [18]. In Figure 2, we show the comparative costs for our scheme and Path ORAM for a concrete selection of parameters (see Appendix A for the recurrence relations used to calculate these numbers). Our write-only ORAM is more than an order of magnitude more efficient than Path ORAM.

To see why we are more efficient, it is useful to consider the complexity of our write-only ORAM and Path ORAM in terms of the level of recursions required to store the map. We denote this required level of recursion with L. In the uniform block setting, Path ORAM has $O(B \cdot L \cdot \log n)$ communication complexity. Since $L = O(\log n)$, this leads to the overall complexity of $O(B \cdot \log^2 n)$. Our write-only ORAM, by comparison, has $O(B \cdot L^2)$ complexity, with no independent $\log n$ factor. Again, since $L = O(\log n)$ our scheme has overall $O(B \cdot \log^2 n)$ complexity. However, for L to actually approach the worst-case $\log n$, the block size needs to be close to $2 \cdot \log n$. In other words, with 512 byte = 4096 bit blocks, we would need a disk holding 2^{2056} bytes to approach that many levels of recursion ($4096 = 2 \cdot \log n$, so $n = 2^{2048}$ blocks, each of size $B = 512$ byte. Note that our write-only ORAM wastes 50% of all blocks, though). In contrast, up to 16 TB requires only $L \leq 3$ levels of recursion. However, for that same disk, we have $\log n = 35$. Therefore, for practical parameter choices, L^2 will be significantly smaller than $L \cdot \log n$, resulting in large savings of at least an order of magnitude.

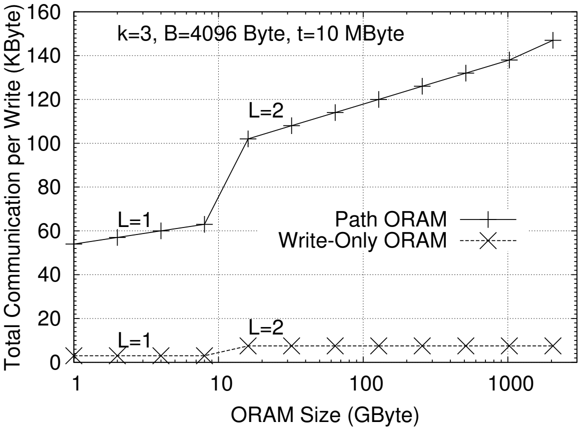

Figure 2: Communication cost

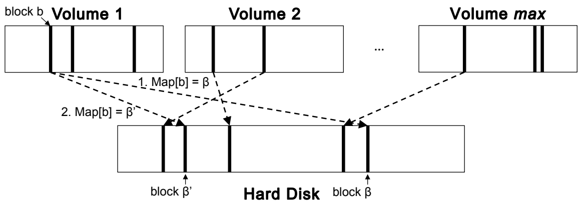

Figure 3: Random mapping of blocks from volumes to disk blocks

The "jump" at 16 GByte ORAM size in Figure 2 is due to an increased level of recursion L required for ORAMs of this size.

5.2 HIVE: Combining Volumes

A significant drawback of our generic hidden volume construction is that it requires a separate ORAM for each volume. This requires us to do a full operation on each of those ORAMs, resulting in a complexity dependent on max. If we make a reasonable choice, say $max = 10$, this could be a significant overhead. Fortunately, our ORAM construction is particularly suited to solving this problem. As we are only actually changing one data block each operation, independent of the number of volumes (the rest are reencryptions), we can thus combine all of our volumes into one. This is the main idea of our new scheme HIVE.

Overview: HIVE's approach will be to store all the volumes randomly interleaved. Then, as shown in Figure 3, after writing to a block b, we will randomly change b's mapping $\mathsf{Map}[b]$ from β to β' using our write-only ORAM trick. This hides the write pattern for all volumes. We now present the full procedure for HIVE's writing ($\mathsf{HVEWrite}$) in Algorithm 6 and for HIVE's reading ($\mathsf{HVERead}$) in Algorithm 7.

HVEWrite: Compared with our original ORAM scheme, there is only one major change: instead of only modifying one block out of k given by the indices in $\mathcal{S}$, we now have to potentially modify all k blocks. To see why, it is useful to consider the following scenario. Imagine user $\mathcal{U}$ wants to write a block into V_1. In order to do that, they randomly choose k blocks to form $\mathcal{S}$. If, for instance, three of these blocks contain data from V_2, and one is empty, we cannot simply write our block from V_1 into this empty space because, after continuously writing "around" blocks in V_2, $\mathcal{A}$ could infer its presence by the pattern we make trying to avoid it.

Therefore, in order to be secure, we have to make sure that no *previous* writes to a $V_j, j > i$ can influence a write to V_i. Intuitively, if operations in higher volumes cannot influence operations in lower volumes, then the existence of a higher volume cannot be inferred from the observed behavior in a lower volume.

This can be accomplished by structuring the stash as a series of multiple *queues* (called Stash_i in Algorithm 6), one for each volume V_i. For an HVEWrite, all the blocks in $\mathcal{S}$ are read and their content copied into the respective queues for their volume. A block on the disk may belong to some volume i, or it may be a random string not containing any information. To put it into the correct queue, we have to attempt to decrypt the block with every key until we find the right volume or we run out of keys. This means that our ciphertexts must contain some redundant information, for decryption verification, but this can easily be accomplished by padding our plaintexts with a number of zero bits proportional to security parameter s. Verification then consists of decrypting and checking if the plaintext begins with a sufficient number of zeroes. This meshes with our notion of security, because $\mathcal{A}$, knowing some number of keys, cannot tell whether a block is part of a volume which he does not have the key for or if it is simply a random string.

Now, having freed k blocks on the disk, we will write k blocks back to the positions given by $\mathcal{S}$. For this, we read out of our stash queues, giving priority to the queues for lower volumes. That is, we empty the queue Stash_1 for V_1 first, followed by Stash_2 for V_2, etc. If there are less than k blocks in all stashes, then the leftover blocks from $\mathcal{S}$ are filled with random strings.

HVERead: For a HVERead, we read the block just like in our regular ORAM by querying the recursive map for the address and then reading that block from the disk. After we retrieve the target block, we also perform a "dummy" write to make reads and writes look identical to adversary $\mathcal{A}$. This write does not change any values in the system, but it gives us a chance to write items from the stash, if necessary. Note that a block b we want to read in volume V_i might reside in the stash, so we first lookup b in Stash_i.

HVESetup: Initialization is identical to our base ORAM, we simply initialize the map to an "empty" state where every block starts unmapped.

Complexity: Figure 4, moved together with its recurrence relations to Appendix A, shows how HIVE is more efficient than the generic construction using max separate ORAMs. Although HIVE performs better for a large range of parameters, note that HIVE scales exponentially in L. Since we have to change up to k blocks at each level, which in turn requires changing k blocks in all the levels below them, the overall complexity is $O(k^L)$. With $max = 10$, for up to 1 Exabyte, HIVE is still cheaper than our generic construction since it is independent of max. However, with larger volumes and more levels of recursion, HIVE becomes less practical than our first construction. Therefore, depending on the choice of parameters, it can be more efficient to use one or the other.

For completeness sake, we note that Path ORAM can also be modified to produce a "combined volume" version in a similar manner (due to its use of a stash), but it would be significantly less efficient than our construction, just as Path ORAM is less efficient than our write-only ORAM.

Security: Since a block from $V_j, j > i$ is written only if the queue for V_i is already empty, blocks from V_j cannot influence $\mathcal{A}$'s view of V_i. Additionally, since an encryption under P_j is indistinguishable from a random string to $\mathcal{A}$ which does not know P_j. What this means is that $\mathcal{A}$'s view of the disk cannot be impacted by volumes which he does not have the key to, therefore we achieve $\Gamma_{\mathcal{A},\Sigma}^{\mathsf{Arbitrary},\mathsf{Plausible}}$ security as before.

HIVE-B: To make HIVE secure against $\mathsf{Opportunistic}$ adversaries, we can follow the same idea as in Section 4.2. Instead of immediately writing in volumes $V_i, i > 1$, we just add the block to Stash_i. When we do operations in V_1, we proceed as normal, writing as much as we can from $\mathsf{Stash}_2, \mathsf{Stash}_3$, etc. We omit full algorithms and proofs for HIVE-B, because of space constraints, but they follow immediately from existing descriptions and proofs.

```
Input: Block index $b$, data $d$, volume $i$, passwords $\mathcal{P}$
if $b \neq \perp$ then
    $\mathsf{Enqueue}(\mathsf{Stash}_i, (b,d))$ ;
end
$\mathcal{S} := <\beta_j>$,
such that $\beta_j \xleftarrow{\$} \{1,...,N\} \wedge 1 \leq j \leq k \wedge \forall u,v : \beta_u \neq \beta_v\}$ holds;
// Fetch blocks from $\mathcal{S}$ and put into stashes
for $u := 1$ to $k$ do
    $d := \mathsf{DiskRead}(\mathcal{S}[u])$;
    if $d$ is block $b$ from volume $V_v$ then
        // Derive $\kappa_v$ from $P_v$
        $d := \mathsf{Dec}_{\kappa_v}(\mathcal{S}[u])$;
        $\mathsf{Enqueue}(\mathsf{Stash}_v, (b,d))$ ;
    end
end
$v := 1$ ;
for $u := 1$ to $max$ do
    if $v \leq k \wedge \mathsf{Stash}_u \neq \varnothing$ then
        $(b,d) := \mathsf{DeQueue}(\mathsf{Stash}_v)$;
        $\mathsf{DiskWrite}(\mathcal{S}[v], \mathsf{Enc}_{\kappa_u}(d))$;
        $v := v+1$;
    end
end
while $v \leq k$ do
    // Fill remaining
       blocks in $\mathcal{S}$ with random strings
    $r \xleftarrow{\$} \{0,1\}^B$ ;
    $\mathsf{DiskWrite}(\mathcal{S}[v], r)$;
end
$M := \lfloor \frac{B}{\log N} \rfloor$ ;
for $j := 1$ to $k$ do
    // Let $b'$, $i'$ be
       the block index and volume number of the
       block that was written to disk block $\mathcal{S}[j]$
    $\mathrm{map_{block}} := \mathsf{Map.Read}(i', \lfloor \frac{b'}{M} \rfloor)$ ;
    $\mathrm{map_{block}}[b' \mod M] := \mathcal{S}_j$ ;
    $\mathsf{Map.HVEWrite}(i', \lfloor \frac{b'}{M} \rfloor, \mathrm{map_{block}})$;
end
```

Algorithm 6: HIVE $\mathsf{HVEWrite}(b,d,i,\mathcal{P})$

```
Input: Volume $i$, block index $b$
if block $b$ is in $\mathsf{Stash}_i$ then
    Read and return most recent version of block $i$ from $\mathsf{Stash}_i$ ;
end
$M := \lfloor \frac{B}{\log N} \rfloor$ ;
$\mathrm{map_{block}} := \mathsf{Map.Read}(i, \lfloor \frac{b}{M} \rfloor)$ ;
$location := \mathrm{map_{block}}[b \mod M]$ ;
$d := \mathsf{DiskRead}(location)$;
// Do a "dummy" write
$\mathsf{HVEWrite}(\perp,\perp,\perp,\perp)$;
return $v$
```

Algorithm 7: HIVE $\mathsf{HVERead}(b,i)$

6. IMPLEMENTATION

To show its real-world practicality, we have implemented HIVE for Linux. Our implementation comprises a kernel module offering a virtual block device for each volume and a userland tool to manage these volumes. The source code is available for download [2].

The kernel module is built using *device-mapper*, a standard Linux kernel framework for mapping block devices onto virtual devices, also used to implement technologies such as LVM, dm-crypt and software RAID.

Table 2: HIVE Benchmarks, $L=2$, $k=3$

	Seq. Write (MB/s)	Seq. Read (MB/s)	Create (Kfiles/s)	Stat (Kfiles/s)	Delete (Kfiles/s)
Raw disk	216.04	221.74	82.29	201.18	105.10
HIVE	0.97	0.99	1.57	3.23	1.79

Device-mapper allows placing HIVE between the Linux block IO layer and the underlying device drivers. There, HIVE intercepts all block IO requests in flight, splits them into single-block-sized chunks, remaps them to their new physical blocks on the disk, and performs cryptography operations, as previously described. Note that our implementation works on any block device (e.g., hard disks, USB sticks, network block devices, etc.) since it stacks on top of the actual device driver which communicates with hardware.

We use AES-CBC with 128 Bit keys for encryption and PBKDF2 for key derivation. For performance reasons, we generate randomness using RC4, using the kernel's entropy pool only to generate an initial key for RC4. Our implementation supports up to a 4 KB logical block size (this limit is imposed by the x86 architecture and kernel internals), regardless of the underlying hardware's physical structure. In our evaluation presented below, we set the block size B to 4 KB, even though our test device has 512-byte sectors. This minimizes the number of random disk accesses performed during IO and results in a significant performance improvement. As another performance optimization, our system disables IO reordering and scheduling in the kernel for the virtual devices, because HIVE always performs random device access and cannot benefit from kernel's access pattern anticipation features.

The userland tool allows users to create, mount and unmount HIVE devices `/dev/mapper/HIVEi` for volume V_i on top of any other block device (e.g., `/dev/sda`). Upon receiving the create command, our tool formats the specified device by creating the necessary metadata structures, such as the max different Maps, max stashes of fixed size, IVs and the reverse mappings for data blocks. Note that our implementation allows for recursion, so it recursively stores and accesses the Maps of fixed size as described in Section 4.3. Finally, to mount or unmount these volumes, our tool issues the appropriate ioctl commands to the kernel's device-mapper module.

Benchmarks: We have tested our implementation on a standard desktop computer with an Intel i7-930 CPU, 9 GB RAM (although RAM was not an issue during our evaluation), running Arch Linux x86-64 with kernel 3.13.6. As the underlying block device, we have used an off-the-shelf Samsung 840 EVO SSD.

For the evaluation, we used bonnie++, a standard disk and filesystem benchmarking tool. Note that in the face of IO caching by the OS, files created in the bonnie++ benchmarks must be set to twice the size of system memory installed (9 GB in our case) to reliably measure device performance. To speed up the total benchmark time, we modified bonnie++, flushing IO buffers to the device after running a benchmark, and signaling the kernel to drop $page$, $dentry$, and $inode$ caches before the next run. This ensured that our performance measurements remained unaffected from caching.

We have first tested an ext4 filesystem with 4 KB blocks on the "raw" disk to get a baseline. We then created 2 hidden volumes on our disk and set $L=2$ and $k=3$. With this configuration, HIVE supports volumes of a total size of up to 16 TByte. We repeated the experiments by running bonnie++ on an ext4 filesystem created on top of the HIVE block device. Table 2 presents the results, averaged over 5 runs with a relative standard deviation of $<6\%$. These results show that IO operations (sequential writes and reads in MB per second) were slower by a factor of $\approx$ 200, while filesystem operations (create, stat, and delete in thousands of files per second) were slower by a factor of 50 to 60. Random seek performance was not measurable on the raw SSD (i.e., bonniee++ reported that the tests completed too quickly to measure reliable timings), whereas HIVE achieved 1.2 Kseeks/s. The HIVE induced CPU utilization was low with $<1\%$ during measurements, indicating that random access IO constitutes the main bottleneck.

We conclude that, while the slowdown is certainly significant, a throughput of 1 MB/s on an off-the-shelf disk is acceptable in many scenarios, rendering HIVE practical for the real-world. Future work is dedicated for additional performance tuning.

7. RELATED WORK

Anderson et al. [1] present *StegFS*, two different techniques to hide data on a hard disk. The first technique creates a set of "cover files" such that any user file can be reconstructed as a password-based linear combination of the cover files. As the security of this scheme is based on the number of linear combinations possible, note that knowledge of some user files jeopardizes secrecy of other files. Modifications to files with unknown passwords are not deniable in the case of multiple snapshots. Consequently, this technique offers weaker security than the ones discussed in this paper. The second technique by Anderson et al. [1] (and its extension [13]) is based on a simple hash scheme, where an encrypted file is stored in the hard disk block indexed by $\beta := h(\text{file name})$. To avoid resulting hash collisions, Pang et al. [14], iterate over the physical blocks following β, until a free block is found. While this mitigates the original problem of known files, none allow deniability against multiple snapshots, as write access patterns are not protected.

The same holds for prominent disk encryption tools such as TrueCrypt [20], Mobiflage [17], FreeOTFE [5], and Rubberhose [9]: these tools use a fixed mapping between blocks of a volume ("virtual disks", "aspects") to hard disk blocks. With access to one volume, other (hidden) volumes look like free space to adversary $\mathcal{A}$. Thus, repeated access to the same blocks in the hidden volumes will be impossible to deny.

The idea of deniable disk encryption originates to work by Canetti et al. [3]. There, $\mathcal{A}$ intercepts ciphertext $c=\mathsf{Enc}_{pk}(m,r)$ sent from a sender to a receiver, where pk is the receiver's public key, m a (delicate) plaintext, and r a random coin. The goal is that, under coercion, the sender can present an innocent plaintext, random coin pair (m',r'), where $m'\neq m$, such that $c=\mathsf{Enc}_{pk}(m',r')$.

8. CONCLUSION

In this paper, we have proposed new, parameterized definitions of formal security for hidden volume encryption. Existing work lacks security against strong adversaries with multiple snapshot and chosen plaintext capabilities. We have proposed new constructions which meet strong security definitions using ORAM as a building block. Observing that strong security can be provided with less powerful *write-only* ORAM, we have then introduced a novel construction of a write-only ORAM which achieves optimal $O(1)$ communication complexity. This is a surprising result, indicating that writes are easier to hide than reads. We leave further exploration of this idea to future work. Additionally, we have shown how our ORAM can be specially adapted to the problem of hidden volume encryption to produce an even more efficient solution, which we dub HIVE. Finally, we have implemented our scheme as a kernel-level block device and benchmarked its performance on commodity hardware, achieving a throughput of ≈ 1 MByte/s.

References

[1] R.J. Anderson, R.M. Needham, and A. Shamir. The Steganographic File System. In *Proceedings of Information Hiding*, pages 73–82, Portland, USA, 1998. ISBN 3-540-65386-4.

[2] Anonymized. HIVE Source Code, 2014. `https://www.dropbox.com/s/miis1x681wrrrak/hive.zip`.

[3] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable Encryption. In *Proceedings of CRYPTO*, pages 90–104, Santa Barbara, USA, 1997. ISBN 3-540-63384-7.

[4] A. Czeskis, D.J. St. Hilaire, K. Koscher, S.D. Gribble, T. Kohno, and B. Schneier. Defeating Encrypted and Deniable File Systems: TrueCrypt v5.1a and the Case of the Tattling OS and Applications. In *Proceedings of HotSec*, San Jose, USA, 2008.

[5] S. Dean. FreeOTFE, 2010. Archive available at `https://web.archive.org/web/20130531062457/http://freeotfe.org/`.

[6] Executive Office of the President. Protection of Sensitive Agency Information, 2006. `http://www.whitehouse.gov/sites/default/files/omb/memoranda/fy2006/m06-16.pdf`.

[7] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3): 431–473, 1996. ISSN 0004-5411.

[8] P.G. Harrison and N.M. Patel. *Performance Modelling of Communication Networks and Computer Architectures*. Addison-Wesley Longman Publishing Co., Inc., 1992. ISBN 0201544199.

[9] R.P. Weinmann J. Assange and S. Dreyfus. Rubberhose Filesystem, 2001. Archive available at `http://web.archive.org/web/20120716034441/http://marutukku.org/`.

[10] B. Jansson. Choosing a good appointment system – A study of queues of the type $(D,M,1)$. *Operations Research*, 14(2): 292–312, 1966. ISSN 0030-364X.

[11] B. Kaliski. RFC 2898, PKCS 5: Password-Based Cryptography Specification Version 2.0, 2000. `https://tools.ietf.org/html/rfc2898`.

[12] L. Li and A. Datta. Write-Only Oblivious RAM based Privacy-Preserved Access of Outsourced Data, 2013. Cryptology ePrint Archive, `https://eprint.iacr.org/2013/694`.

[13] A.D. McDonald and M.G. Kuhn. StegFS: A Steganographic File System for Linux. In *Proceedings of Information Hiding*, pages 462–477, Dresden, Germany, 1999. ISBN 3-540-67182-X.

[14] H.H. Pang, K.-L. Tan, and X. Zhou. StegFS: A steganographic file system. In *International Conference on Data Engineering*, pages 657–667, Bangalore, India, 2003. ISBN 0-7803-7665-X.

[15] P. Rogaway. Nonce-Based Symmetric Encryption. In *Proceedings of Fast Software Encryption*, pages 348–358, Delhi, India, 2004. ISBN 978-3-540-22171-5.

[16] E. Shi, T.-H.H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost. In *Proceedings of Advances in Cryptology – ASIACRYPT*, pages 197–214, Seoul, South Korea, 2011. ISBN 978-3-642-25384-3.

[17] A. Skillen and M. Mannan. On Implementing Deniable Storage Encryption for Mobile Devices. In *Proceedings of NDSS*, San Diego, USA, 2013.

[18] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of Conference on Computer & Communications Security*, pages 299–310, Berlin, Germany, 2013. ISBN 978-1-4503-2477-9.

[19] TrueCrypt. Security Requirements and Precautions, 2014. `http://www.truecrypt.org/docs/security-requirements-and-precautions`.

[20] TrueCrypt. Free open-source on-the-fly encryption, 2014. `http://www.truecrypt.org/`.

APPENDIX

A. RECURRENCE RELATIONS

For Path ORAM, we define relation $A(n)$ expressing the number of block operations (reads and writes) which need to be performed for one ORAM access.

$$A(n) = \begin{cases} 0, & \text{if } n \le \frac{t}{B} \\ 2 \cdot k \lceil \log_2 n \rceil + A(\lceil \frac{n}{\lfloor \frac{B}{\lceil \log_2 n \rceil} \rfloor} \rceil), & \text{if } n > \frac{t}{B} \end{cases}$$

Access requires that the user retrieve k blocks from each bucket on a path of a heigh $\log_2 n$ tree, plus retrieval of one block from the map which is itself stored in an ORAM.

For a generic hidden volume construction, the number of block operations using Path ORAM is then $max \cdot A(\frac{n}{max})$.

Similarly, we can define relations $W(n)$ and $R(n)$ for our write-only ORAM which express the number of block accesses required per write and read operation respectively.

$$W(n) = \begin{cases} 0, & \text{if } n \le \frac{t}{B} \\ k \cdot R(\lceil \frac{n}{\lfloor \frac{B}{\lceil \log_2 2n \rceil} \rfloor} \rceil) + & \\ W(\lceil \frac{n}{\lfloor \frac{B}{\lceil \log_2 2n \rceil} \rfloor} \rceil) + 2k, & \text{if } n > \frac{t}{B} \end{cases}$$

Writing requires reading and writing k blocks, plus reading k entries from the recursive map and writing one entry.

$$R(n) = \begin{cases} 0, & \text{if } n \le \frac{t}{B} \\ 1 + R(\lceil \frac{n}{\lfloor \frac{B}{\lceil \log_2 2n \rceil} \rfloor} \rceil), & \text{if } n > \frac{t}{B} \end{cases}$$

Reading requires just one block per level of the recursive map.

Again, for a generic hidden volume construction, the number of block operations using our ORAM is then $max \cdot W(\frac{n}{max})$.

Finally, we define a similar set of recurrences for HIVE as follows.

$$W(n) = \begin{cases} 0, & \text{if } n \le \frac{t}{B} \\ k \cdot R(\lceil \frac{n}{\lfloor \frac{B}{\lceil \log_2 2n \rceil} \rfloor} \rceil) + & \\ k \cdot W(\lceil \frac{n}{\lfloor \frac{B}{\lceil \log_2 2n \rceil} \rfloor} \rceil)\rceil + 2 \cdot k, & \text{if } n > \frac{t}{B} \end{cases}$$

Writing is the same as above, except now we have to change k blocks in the recursive map instead of just one.

$$R(n) = \begin{cases} 0, & \text{if } n \leq \frac{t}{B} \\ 1 + R(\lceil \frac{n}{\lfloor \frac{B}{\lceil \log_2 2n \rceil} \rfloor} \rceil), & \text{if } n > \frac{t}{B} \end{cases}$$

However, for this construction the total number of access is just $W(n)$, since we do not divide the system into separate volumes.

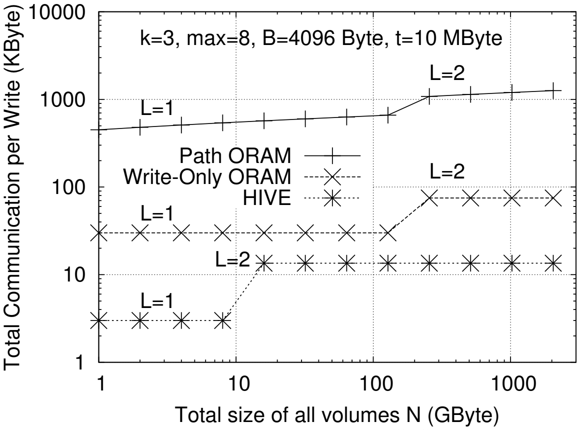

Figure 4: Communication cost, $N = 2 \cdot \sum_{i=1}^{max} n_i$, Log-Log plot