HelmsDeep: Isolated Time-Based Defense for Storage Systems

Abstract

Ransomware poses escalating risks, significantly impacting organizations like hospitals. Attackers exploit system and human vulnerabilities to encrypt data, and demand a ransom for its release. While most organizations rely on backups, these backups are also vulnerable. Prior attempts to reduce the trusted computing base (TCB) of backup systems failed to eliminate all exploits; thus, a new approach is needed.

We present HelmsDeep (HD), a secure backup system with a minimal TCB that uses time as a defense mechanism. It features a storage primitive called a timelock, which protects data from everything (including credentialed users) for a set duration. We propose a novel design that excludes versioning logic from our TCB and eliminates all exploits. This results in a small, formally verified HD controller isolated from all other parts of the system. The controller's narrow yet flexible interface supports optimizations such as incremental checkpointing, and we propose a secure method for caching HD metadata in the untrusted host system. Our experiments show HD incurs negligible space, performance, and storage I/O overheads compared to conventional versioning systems.

1 Introduction

Ransomware is expected to cost its victims around $265 billion annually by 2031 [48]. According to a recent international survey [58], 67% of hospital organizations were hit by ransomware in 2024 alone. When attackers target critical infrastructure such as hospitals [3], they commit not just financial crimes, but also life-threatening crimes.

In ransomware, an attacker gains control of a victim's computer system through methods such as exploiting computer system vulnerabilities [4], compromising credentials [38], malicious emails [24], phishing [5, 6, 23], etc. After intrusion, ransomware encrypts the victim's data or locks them out, rendering the system inaccessible or unusable. The attacker then demands a ransom in exchange for restoring access.

A widely used defense strategy is to periodically create backups of the storage system state with a versioning system (VS) [12, 13, 20, 40], and then revert to an uncorrupted state after an intrusion. Unfortunately, the VS is itself vulnerable to attacks. 95% of healthcare organizations hit by ransomware reported that attackers attempted to compromise their backups, and two-thirds of those attempts were successful [58].

Self-securing storage (S3) [53] is a promising solution where the storage device implements its own VS and retains backups for a fixed period, which we refer to as a "timelock". We define the timelock as a security primitive that ensures the **transient immutability** of data, preventing any modification to that address for a specified duration. During this time, no entity can modify these backups without physical access to the storage device. Compared to traditional backups, S3 has a smaller trusted computing base (TCB) that excludes applications, the OS, and system administrators. Stolen credentials represent two-thirds of system exploits [58], so eliminating trust outside of the storage device is essential, and timelocks are how S3 achieves this.

A timelock adds a significant hurdle to an attacker who compromises the host system. The value of a timelock is three-fold: (1) attackers who "wait-out" the timelock risk being detected before the threat takes effect, and increase the time to collect a ransom (2) a safety window provides administrators with more time to discover exploits and deploy patches (3) expensive deep scans that detect intrusions, invalid/corrupted data, etc. can run less frequently and analyze changes in system state over time. Timelocks allow administrators to trade extra storage space for greater security guarantees. With unlimited storage capacity, an infinite timelock makes ransomware impossible, but more realistically, having enough capacity to store six months of old data provides a massive challenge for ransomware, as seen in Figure 2.

Existing S3 solutions [25, 53, 57] use timelocks within the VS on the storage device. However, this leaves the TCB large enough for exploitation, such as the recent "trimming attack" that bypasses timelock logic in the VS to delete backups [44]. Another significant disadvantage of an integrated VS and timelocks is that the VS policies are embedded within the TCB. This means users cannot modify these policies without making changes to the TCB itself.

While timelocks are a powerful defense against ransomware, their tight integration with versioning leads to vulnerabilities [44] and inflexible policies. Our goal is to create a secure backup system by decoupling versioning from timelocks, such that the entire VS is excluded from the TCB.

To accomplish this, we must address two primary concerns. (1) The timelock mechanism in our TCB must be bug-free. (2) All prior S3 solutions [25, 53, 57] frequently modify version metadata using the VS, and trust all metadata created by the VS. Not trusting the VS means we cannot distinguish between (a) begnign overwrites versus malicious overwrites of version metadata, and (b) genuine version metadata versus spoofed metadata designed to confuse our recovery software (note that spoofing metadata does not require overwrites).

We propose three new ideas to address these issues: (1) a formally verified timelock implementation that guarantees transient immutability of data (Section 3.1). (2) Unlike existing solutions [25, 53, 57] that frequently modify version metadata, we design a VS that protects its version metadata with timelocks to defend against future intrusions. To apply timelocks on version metadata, we designed a VS that, to the best of our knowledge, is the first never to overwrite its own metadata (Section 4.3). (3) A method for disambiguating legitimate and spoofed version metadata (Section 4.4).

We introduce HelmsDeep[1] (HD), a secure backup system that provides a time-based last line of defense against ransomware through an isolated microcontroller that enforces timelocks. Unlike previous approaches, our timelock is independent of the VS. In HD, the VS is not part of the TCB and runs on the untrusted host. The HD controller interface to the host is a modified set of storage interface commands, allowing for reads, writes, and synchronization to a storage device along with timelock constraints. Due to its simplicity, the HD controller implementation is only ∼200 lines of code, enabling formal verification of our security properties.

We build a secure backup system on top of the isolated HD controller. The VS is untrusted – even if an attacker compromises the VS, our system is guaranteed to be able to recover to a pre-intrusion state. The VS continuously tracks version state and stores its metadata with a timelock. If an attacker gains complete control of the host system, including the VS, HD's transient immutability guarantee ensures that checkpoints created before the intrusion remain unaltered until their timelock expires.

We also enable conventional versioning optimizations while still enforcing timelock constraints. Incremental checkpointing [22, 41] is a common versioning optimization that saves storage space by aggregating multiple writes within a short time interval. This conflicts with the timelock constraint that prevents overwrites for the timelock duration. We solve this problem by initially writing data with a timelock of duration zero, allowing for instant modification. Once the time interval passes and the incremental checkpoint is made, the VS can issue a timelock *increment* to each version written in the interval. This allows for space-saving optimizations before the timelock protections are applied.

We implement our untrusted VS in a block device driver and mount it with an ext4 filesystem for applications to interact with. Our HD controller implementation is formally verified using Dafny [29]. We prototyped the HD controller on a Raspberry Pi to evaluate performance and space overheads. The controller sits physically between the untrusted host and the storage device, and acts as an interposer.

HD incurs no overhead for reads. For writes, the controller reads HD metadata, checks if the write is allowed, and then updates the HD metadata. We eliminate auxiliary storage reads by **securely caching HD metadata on the untrusted host**, an optimization we discuss in Section 5.2.3. A delayed write-back design amortizes one HD metadata write across 512 data writes. Our experiments on storage traces from FIU [28] and Microsoft [36], as well as several database applications [42], show that HD has negligible execution and throughput overheads of 0.3% and 0.5% compared to a conventional VS.

[1]Helm's Deep is a fortress from J.R.R. Tolkien's *The Lord of the Rings*, where the army of Rohan served as the last line of defense to protect the people from the forces of Saruman until Gandalf arrived to help.

Our paper makes the following contributions:

- We design and implement a secure backup system based on timelocks, without trusting the VS. It is the first to timelock version metadata and never require overwrites.
- We formally verify the HD controller, which guarantees transient immutability through timelocks.
- We design a time-based ransomware defense with a smaller TCB than prior systems (∼200 LoC).
- We propose a new technique using timestamps to disambiguate potentially spoofed version metadata.
- We demonstrate that HD incurs negligible performance overhead by securely caching HD metadata on the untrusted host and HD metadata write coalescing.

2 Background: Ransomware Threat Model and Secure Backups

In this section, we describe a threat model for ransomware attacks and the requirements for secure backups.

For an encryption ransomware attack, following intrusion, the attacker encrypts the persistent state using their private key [38], thereby hijacking the user's data. Encryption ransomware is the most common ransomware attack [38]. The fundamental requirement for ransomware attacks is that the attacker gains the ability to *modify* users' storage data.

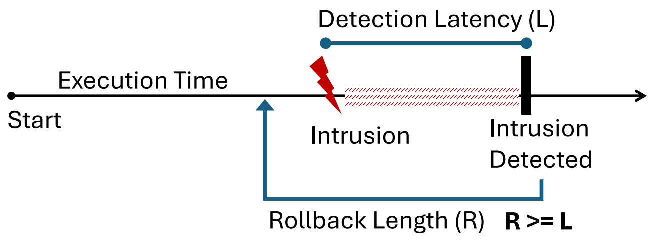

Figure 1: Recover by restoring state before the intrusion.

Figure 1 illustrates a simplified timeline of a ransomware attack and recovery using a continuous backup system. Before intrusion, a VS takes periodic checkpoints of data on a storage device. After intrusion, we assume that an attacker has complete control over the system and storage.

At some point, the intrusion is detected (in the worst case, an intrusion is detected when the attacker makes the ransom request). We define the latency between intrusion and detection as intrusion detection latency (L), as shown in Figure 1.

The fundamental property of secure backups is enabling rollback that restores the storage state to *before* the time of intrusion. We refer to this as rollback length (R). To defend against intrusion, the rollback length has to be greater than the expected intrusion detection latency (L).

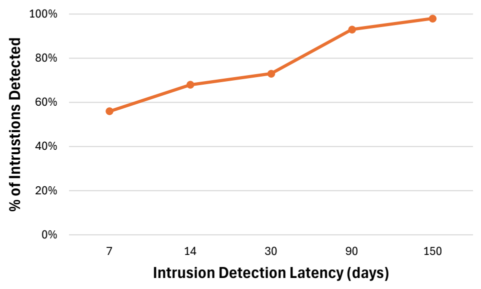

Figure 2: Global ransomware dwell time CDF in 2023 [1].

HD Command	Inputs	Outputs/Impact	ATA Command
read	addr	Data @ addr	(0x25) READ DMA EXT
read-md	addr	HD metadata	(0x28) Unused opcode
write	Data, Timelock	Write data @ addr; Freeze Timelock	(0x35) WRITE DMA EXT
unfreeze	addr	Update timelock state to Countdown Timelock	(0x48) Unused opcode
inc	addr, increment	Add increment to timelock	(0x49) Unused opcode
sync	N/A	Make disk caches persistent	(0xEA) FLUSH CACHE EXT
identify	N/A	Obtain disk model, serial number, geometry, etc.	(0xEC) IDENTIFY DEVICE

Table 1: HD Controller's Modified ATA Interface

Need for time-based defenses: We aim to create a secure backup solution that defends against the most dangerous ransomware threat: an attacker stealing the data owner's credentials [58]. Ideally, we would protect data backups indefinitely, but this goal is infeasible due to storage constraints. According to a recent report [1] from Google, the median intrusion detection latency (dwell time) in 2024 is only five days, and $\geq 99\%$ of intrusions are detected within 5 months (Figure 2).

Requiring attackers to wait for a period of time before they can modify data is a significant hurdle that complements existing ransomware prevention and recovery techniques. Timelocks raise the chances of detection [7, 35, 39], provide time for patches to be discovered and applied [16, 30], and reduce the frequency that expensive deep scans need to be run.

Isolation for Time-Based Defenses: Prior S3 systems [25, 53, 57] integrate time-based security checks into their VS. Recent work has shown that vulnerabilities in version management can bypass timelock checks and delete old versions, violating the time-based defenses [44]. Thus, we propose isolating the timelock logic from the VS.

Isolation is a well-known security concept that reduces attack surfaces and minimizes the impact of software bugs. For example, in OpenSSH, isolating authentication logic from pseudo-terminal management and running them at different privilege levels prevents privilege escalation attacks [43]. Another example is Page Table Isolation, which uses separate page tables for applications, independent of the kernel, to protect against Meltdown attacks [32, 55]. Many systems use isolation as a key design principle, including microkernels [9, 31], web browsers [45, 46], and hypervisors/VMs [8, 51].

Our work aims to reduce HelmsDeep's TCB as much as possible and isolate it from the rest of the system, including the VS and storage device.

Orthogonal threats: There is a type of ransomware attack in which the attacker threatens to leak sensitive information. Such attacks only require read permission, unlike encryption ransomware, which requires both read and write access. The proposed defenses [21, 33, 49] against leaking/data exfiltration are orthogonal to HelmsDeep and thus can be used in conjunction with each other. For example, time-release encryption schemes [33, 49] can prevent plaintext leakage while HelmsDeep guarantees the ciphertext can't be overwritten.

Another threat model includes physical access to the storage system. These powerful adversaries can steal the physical storage device and ask for a ransom. Our work focuses on defending against digital security threats.

3 HelmsDeep Controller

This section presents the HD controller. It uses time as a defense mechanism, adding a last line of defense on top of conventional user credential-based security. It allows users to write data to an address and protect it using the timelock storage primitive. The timelock guarantees that no entity in the system, including privileged users, can modify that address for the specified duration of time. In Section 4, we detail a secure backup system design built on top of the HD controller.

3.1 HelmsDeep Interface

The HD controller is an interposer between the host and storage device with a formally verified interface. The only way to access the storage device is through the controller.

The **write** command is similar to a conventional disk write but includes an additional parameter specifying the timelock (τ). The timelock enforces a **transient immutability** constraint, preventing data modification for at least time τ.

An important interface design question is when the timelock countdown should begin. Starting the countdown at the time of writing assumes the user has foreknowledge of the data's lifetime. However, users typically do not know if or when they will need the data next. For example, when a version is created for a logical address, its lifetime lasts until the next version is written. This timing is impossible to know in advance without being able to predict the future.

In HD, each write's data are initially protected indefinitely. Later, a user can explicitly **unfreeze** the data. After the data are unfrozen, HD protects the data for the timelock duration specified at the time of write. This ensures that even if an adversary gains control shortly after a write and unfreezes that block's address, they will still have to wait for the timelock duration before modifying the data.

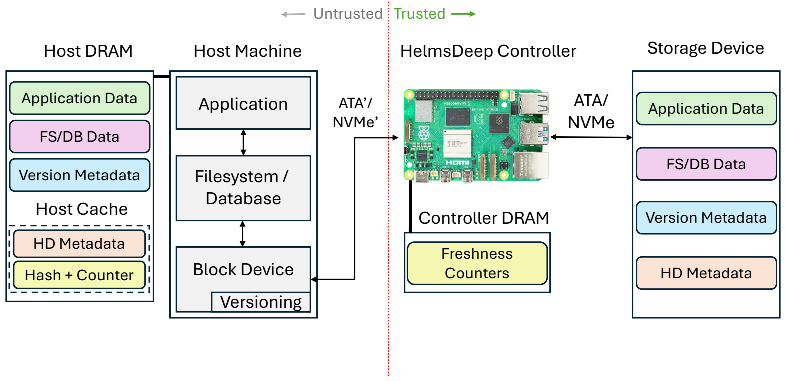

Figure 3a. HelmsDeep system overview. Table 1 lists the HD-ATA commands interface.

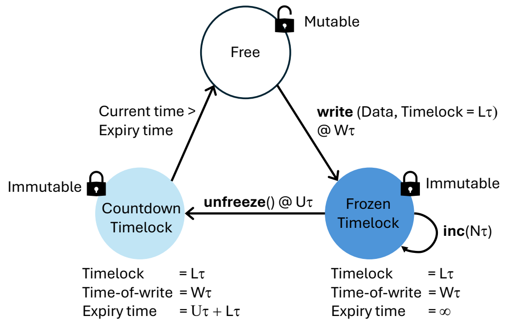

Figure 3b. States of a storage block.

Figure 3b shows the FSM for an address. When a free block is written, it transitions to the frozen timelock state, disallowing additional writes. When unfrozen, it moves to the countdown timelock state, starting the timelock countdown and still disallowing writes. Once the timelock expires, it transitions to the free state, where writes are permitted again.

Additionally, HD allows users to extend the timelock after a write using the **inc** command. This lets users strengthen the timelock guarantee when necessary. For instance, if the threat level rises due to difficulties in detecting an intrusion, a user may choose to increase the timelock for past writes. In Section 4.5, we will discuss how this command also facilitates powerful optimizations such as incremental checkpointing.

On a write, HD internally generates a time-of-write timestamp and stores it in the HD metadata of the address. The **read-MD** command allows users to read the time of the last write for an address, supporting HD's **time-of-write guarantee**, preventing adversaries from spoofing recovery (Section 4.4). Lastly, the **read, sync,** and **identify** commands function as expected, with no additional HD constraints.

3.2 Verified Trusted Controller

The controller is a simple microcontroller natively running the timelock logic. The inputs to the controller are storage interface commands; we use the set "HD-ATA" listed in Table 1. The controller approves or rejects commands based on the current timelock state of each block. The controller allows only valid commands that adhere to the timelock to pass through. HD is agnostic to the storage medium (HDD or SSD). Our design uses the ATA command interface due to its broad adoption and simplicity; however, it can be extended to work with any storage interface, including NVMe.

To perform timelock checks, the controller has a secure timer to generate the current time, which increases at a consistent rate. This secure timer is similar to non-volatile secure counters available in trusted platform modules (TPM) [37] that retain their state across power cycles.

The controller maintains a constant-sized HD metadata (8 bytes) containing the timelock state, timelock duration, time-of-write, and expiry-time for each physical address. The metadata for an address is updated for each state transition. The controller does not constantly update the metadata in the countdown timelock state; it is sufficient to check the metadata on a write, and determine if the expiry time (time-of-unfreeze + timelock) is less than the current time.

HD metadata blocks are interleaved with data blocks in physical storage. We use a static map (defined by an arithmetic function) from HD addresses to physical addresses (Figure 5).

HD incurs negligible time and space overhead. One physical block is used to store metadata for 512 data blocks, and so HD metadata incurs only about 0.2% storage space overhead. This can be cached on the host to eliminate auxiliary reads, as explained in Section 4.3.

The controller and its state (timer and metadata) are isolated from the host computing system, except for communicating through the verified HD-ATA interface. We also formally verify (Section 6) that our controller implementation satisfies HD's security guarantees. As a result, the timelock constraint will always hold even if the host is compromised.

3.3 Small and Isolated TCB

The HelmsDeep controller is isolated from all other system layers. Privileged users, the filesystem, versioning software, drivers, and the OS are all excluded from the TCB. Our verified implementation in Dafny comprises ∼200 lines of code, excluding ghost statements used only to aid in verification.

3.4 HelmsDeep Policy and Use Cases

Prior S3 solutions [25, 53, 57] implement timelocks inside the VS, leading to one of two issues: (a) devices are restricted to inflexible, predefined versioning policies, or (b) modifying the versioning policy requires changes to the TCB logic, which risks introducing new vulnerabilities.

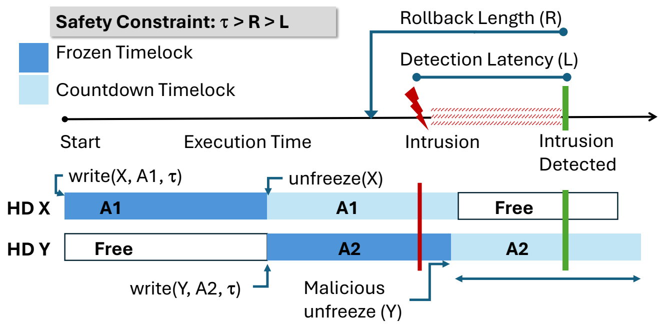

Figure 4: Secure backups using HelmsDeep

HD addresses these limitations by excluding versioning from the TCB. This approach allows users to define their own versioning policy without introducing TCB vulnerabilities.

We outline several use cases for HelmsDeep, beyond conventional versioning. A user with critical data (e.g., cryptocurrency keys) can store them in HD with an infinite timelock. An OS or middleware can automatically discover important files (based on contents/file type) and timelock them with different durations depending on importance. An organization may want to timelock its records to ensure regulatory compliance. HD can also help protect logs during an attack for forensics analysis and analyze intrusion [14, 17].

While there are many applications, we focus on one crucial use case for HelmsDeep: secure backups to defend against ransomware, which we discuss in the next section.

4 Secure Backups Using HelmsDeep

Secure backups enable a rollback to the pre-intrusion state, as explained in Section 2. This section details how HD builds a secure backup system with the VS outside the TCB.

4.1 Overview

The HD controller provides access to a portion of the physical address space (Figure 5), i.e. the HD address space. Our VS is a layer (block device driver) between the client filesystem[2] and the controller, exposing a portion of the HD address space to the filesystem (i.e. logical address space).

The VS allows the client filesystem to overwrite its logical address space. Under the hood, when the filesystem writes to a logical address, the VS writes the data to a free block in the HD address space while retaining the old version (and unfreezing it). The VS manages version metadata, mapping logical addresses to HD addresses where each version is stored. Once an old version has been retained longer than the rollback length, its HD address can be reclaimed.

[2]Any system using a block device works with HD, including filesystems, databases, object stores, etc. We use a filesystem as an illustrative example.

In the following sections, we explain how the timelock protects data versions. Next, we describe how the VS's metadata are secured with a timelock, involving careful design to ensure both time and space efficiency. We then introduce our recovery software, which, after an intrusion, uses only the HD state to revert to a pre-intrusion state. Lastly, we discuss how conventional incremental checkpointing optimizations can be implemented within the context of HD.

4.2 Timelocking Versions

When the filesystem issues a write, the VS stores the data in a free block in HD and secures it with a timelock set to the rollback length (R). Even if an intrusion happens immediately after the write, the worst an adversary can do is attempt to unfreeze the data. However, due to the timelock, HD ensures the data are protected for the duration of R. Therefore, if the intrusion is detected in less than R, the data can be recovered.

Once data are written to an HD address, that HD address remains immutable until explicitly unfrozen by the user. VS unfreezes that HD address only when the filesystem writes a new version to the corresponding logical address.

Figure 4 shows an example. When the filesystem writes to a logical address (A), the VS stores the data in a free block (X) in the HD, applying a timelock of R. On a subsequent write to the same logical address (A), the VS writes the new version (A2) to another free HD address (Y) and immediately unfreezes the old version (A1) stored in HD address X. The unfrozen HD address then enters a countdown timelock state, and stays protected for the duration of R.

Version Integrity Guarantee: To establish that we will always be able to safely recover the latest pre-intrusion version for a logical address, let us consider all possible storage states at the time of intrusion for an address.

First, for HD addresses in a free state, there is no valid data and therefore no guarantee is needed.

Second, we consider addresses in a countdown timelocked state. For instance, in Figure 4, HD address X is in this state at the time of an intrusion. This HD block was unfrozen by our VS before the intrusion. While the VS is untrusted, its behavior before an intrusion is benign. Therefore, it would have issued an unfreeze command for block X only after writing a newer version (A2) to another HD block (Y). As a result, even if block X expires during the attack and loses protection, a more recent version (A2 in Y) can still be recovered for the corresponding logical address.

Finally, an HD address (Y) could be in the frozen timelock state, implying that it contains the most up-to-date version for a logical address and was not unfrozen before the intrusion. In this state, the worst an adversary can do is unfreeze the HD address shortly after the intrusion, as shown in Figure 4. This would only transition the HD address into a counting timelock state, where it remains protected for the duration of the timelock (R). Therefore, as long as the timelock duration is set long enough to outlast the time needed to detect the intrusion, we can recover a valid version for A.

4.3 Timelocking Version Metadata

We assume that an attacker can gain full control over the VS after an intrusion. To enable safe recovery, we must apply the same timelock protection to both data and the version metadata. This is a novel challenge, as prior S3 systems do not timelock their version metadata. All prior VS we are aware of, including "append-only" systems [13, 20, 40, 50], modify version metadata. For example, the LSFS periodically overwrites checkpoint regions every 30 seconds [50].

Our VS metadata is a map from logical addresses to HD addresses. Since we cannot overwrite the map when an update occurs, a naive solution is to track changes of version metadata with more versioning. However, to do this, we would need a second layer of metadata for our version metadata. Future updates encounter the same issues, necessitating the addition of a third layer of metadata, followed by a fourth, and so on. This becomes an infinite recursion problem.

We solve this recursive problem by using an append-only log that tracks the chronological updates of versions. When the filesystem writes to a logical address, the VS will create a new version, which it stores in a free HD address. At the same time, a new metadata entry is added to the end of the log that describes the mapping between the logical and HD addresses. When enough log entries are created (or enough time has passed), a block of log entries is stored in HD with a timelock greater than or equal to the underlying data. Each log block is linked together through a series of forward pointers with the head of the log stored at a predetermined location in the HD address space. This data structure allows us to track all changes to our metadata (versions of versions) without triggering the recursive update problem [50].

Each log entry is small, so we coalesce multiple log entries into a single metadata block. When the VS is under a heavy load of writes, the metadata block fills quickly, resulting in almost no delay between writing data and the metadata log. To account for periods of low write activity, we enforce that our VS writes the metadata block at least once a second, even if the block is not filled. This way, in the event of a power failure, we lose at most one second of activity.

The VS frees the metadata log stored on HD infrequently (user-defined, e.g. once a month). Before unfreezing the old list, the VS quickly computes a new log that only contains the most recently created version for each logical address. The VS accomplishes this by scanning its local copy of the metadata stored in memory. The VS then stores the new log in HD with the same timelock duration as the underlying data. The head of the new log is stored at a second pre-defined location on HD. The garbage collector alternates between the two predefined head locations.

Note that HD is agnostic to data versus version metadata, as it is completely isolated from versioning logic, and protects them using the same mechanisms. The garbage collector that frees VS's linked list of version metadata is also part of the untrusted VS. To free an old list, it issues unfreeze commands to the HD addresses containing the old list after it has written the new state of the log to a different location. The old list will remain immutable in the countdown timelock state. Thus, an intruder cannot corrupt the VS metadata stored on HD for at least the timelock duration.

Optimizations: I/O related to HD metadata is expensive (Figure 7). Caching solves this, however, the controller has limited memory, and HD metadata demands a large cache due to poor temporal locality. We address this by caching metadata in the untrusted host's large memory and protecting it using cryptographic techniques inspired by trusted hardware (Section 5.2.3). We also optimize metadata writes for spatial locality (Section 5.2.4), eliminating nearly all auxilary I/O.

4.4 Spoof-free Guarantee for Recovery

All prior S3 systems [25, 53, 57] assume that their versioning software will behave correctly during an intrusion. This makes recovery after an attack simple. For instance, FlashGuard [25] maintains a backward list of pointers between versions of the same page. During recovery, it uses these pointers to retrace to earlier states [25]. However, with an untrusted VS, the attacker can create new versions with backward pointers to garbage data, making the recovery process complicated or impossible. This section discusses how HD enables rapid recovery after an intrusion without trusting the versioning software.

We assume that the time of intrusion can be estimated. Prior work has designed forensic techniques for reconstructing and analyzing intrusions [14, 17, 60]. Alternatively, one can use a simple method like binary searching over possible intrusion times and analyzing if the recovered state is valid. Besides the time of intrusion, the only other input that our recovery software relies on is the content of the storage device.

The first step is to determine what data was written after the intrusion by reading their time-of-write using the read-MD command through the HD interface. Post-intrusion blocks are ignored and not used for recovery. This eliminates spoofing attacks where the intruder attempts to trick our recovery through new state created after the intrusion.

As discussed in Section 4.2, versioning integrity is guaranteed for the most recent version of all logical addresses. We scan the version metadata linked list to locate the recent version for each logical address (Section 4.3). Each log entry has a logical address and a pointer to an HD address that contains one of its versions. We determine the most up-to-date log entry by reading the time-of-write of the log (using read-MD). This log entry provides us with the HD address for the most recent version of that logical address.

After fully resetting the untrusted components of the system, the VS is initialized with the logical to HD address map constructed during recovery. The system can safely restart from this point onwards.

4.5 Incremental Checkpoint using Timelock Increments

Creating a version for every write can greatly increase storage overhead. A conventional solution is to divide execution into epoch intervals and take a copy of the checkpoint at the end of each interval. An incremental checkpoint [22, 41] typically employs an optimization to maintain only the most recent version written to a logical address executed within its epoch. If an address was not written within an epoch, then the checkpoint has no versions for it.

Our VS implements incremental checkpoints as follows. For the first write to a logical address within an epoch, the VS allocates a new HD block and stores that version with a timelock of zero. Future writes to the same address within the epoch overwrite the address. This is possible because the writes all have a timelock of zero and will immediately become free when unfrozen. At the end of the epoch, the timelock of all newly allocated HD blocks within the epoch is incremented (using HD inc command) to rollback length (R).

Incremental checkpointing does not weaken our security guarantees. The only limitation is that rather than always recovering to the last write before intrusion, we instead roll back to the most recent checkpoint/epoch before intrusion. We use small epoch intervals (one hour), which is negligible compared to the intrusion detection latency (weeks). Given that we may lose all versions produced after an intrusion, incremental checkpointing would additionally lose at most one epoch worth of data, which again is negligible.

5 Implementation

5.1 System Organization

Users create and write data to files using a standard Linux ext4 filesystem. To make files persistent, the filesystem communicates with a block device driver. We implement a new block device driver using the BDUS framework [19]. Our new driver implements a VS and manages version metadata. Our versioning driver communicates with the HD controller through sockets using the new interface in Figure 3a. We implement our HD controller logic in Dafny to formally verify the properties defined in Section 6. Our Dafny code transpiles to Rust, which is compiled using rustc.

Our proposed design is a microcontroller running our code natively, which can intercept storage interface commands. We could not find a commercial device with this capability, so we prototyped the HD controller on a Raspberry Pi.

5.1.1 Storage Interface Extension

HD provides a modified ATA command interface, HD-ATA, to the host based on the ATA specification. HD-ATA (described in Table 1) extends ATA by adding fields for timelock duration to write commands using reserved and obsolete bits, as well as the two new commands unfreeze and increment. We have only included commands that we required for our experiments. In the future, more ATA commands can be proven safe and therefore included in HD-ATA, e.g. power management, SMART, streaming configuration, etc. The controller interprets the commands along with the provided fields and current timelock state of each storage address to determine if it should accept or reject the command. Accepted ATA commands are passed through to the storage device.

5.2 Trusted Controller

We make several optimizations to improve the performance and portability of the trusted controller.

5.2.1 HelmsDeep Metadata

Each address requires 8 bytes of HD metadata. 4 bytes are dual-purpose and store either the timelock value or expiry time, depending on the state. The other 4 bytes are split, with the upper 31 bits used for a ToW timestamp and the lowest bit used for a "free bit." We store metadata for adjacent addresses together, meaning we can fit $\frac{blocksize}{8}$ metadata in a single block. In our system, a storage block is 4KB, meaning exactly 512 metadata entries fit in a single HD metadata block.

5.2.2 HD Metadata Impact on Performance

A write operation in a standard system will incur one storage I/O, which writes the data to an address. A naive implementation of HD, however, adds a high cost to write operations. Every time a write is issued, we must first check the timelock state of an address by checking its HD metadata to ensure the write is permissible. This involves an auxiliary storage read. If the command is successful, the metadata will be updated to reflect the new state. The data and updated metadata must then be written to storage. With no optimizations, our HD controller issues 3 I/O for every write.

5.2.3 HD Untrusted Host Cache for HD Metadata Reads

HD metadata reads are non-contiguous (Section 5.2.4) and costly. Caching metadata in DRAM reduces auxiliary reads. We find that the cache has to be large enough to hold all HD metadata to improve performance, as they lack temporal locality. Storing all metadata (1 block per 512 addresses) would require $\frac{1}{513}$th of the total storage size. This is an issue for a small controller attached to a large storage device.

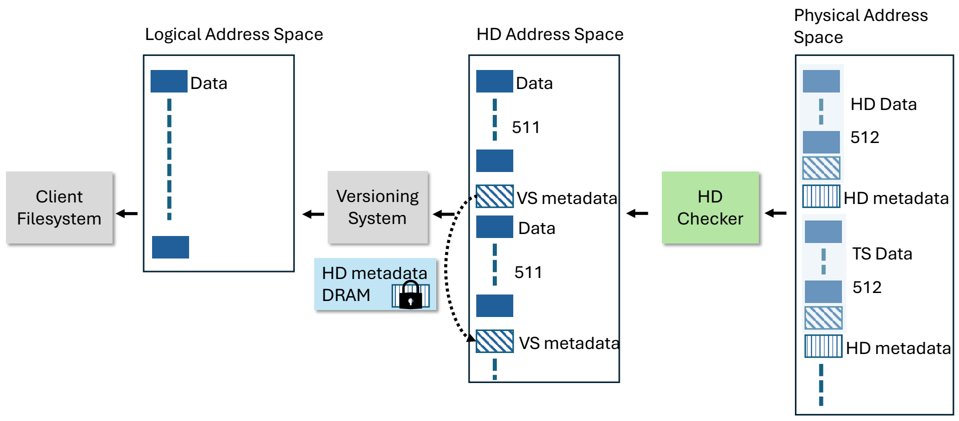

Figure 5: Address space layouts.

We instead let the untrusted host manage an HD metadata cache. We secure the cache using a cryptographic hash (BLAKE-3) computed over the HD metadata, HD address, a freshness counter, and a 256-bit secret salt. On a write/unfreeze/increment, the host sends the corresponding metadata to the controller. The controller verifies the hash and freshness to ensure its integrity.

To track freshness, the controller keeps a 4-byte freshness counter for each HD metadata block. If the counter overflows, a new salt is selected, and all hashes are recomputed. Given that a non-contiguous write operation on an HDD has a latency of ~10 ms, we calculate it would take 1.36 years of non-stop writes to a single block to cause an overflow, meaning re-hashing is an exceedingly rare event. This solution shifts the controller's memory requirement from $\frac{1}{513}$th to $\frac{1}{525312}$th of disk size (2 MB per 1 TB), making it scalable.

5.2.4 HD Metadata Write Optimizations

As explained in Section 5.3, our VS generates sequential writes to storage, which is important for performance, especially on disk drives. However, because HD metadata is modified whenever a new version is written, we risk ruining the sequential pattern if HD's metadata is not written contiguously with the data. To address this issue, we lay out our physical address space as shown in Figure 5. 512 data blocks are contiguous, and the 513th block holds all of their HD metadata. We then add a metadata write coalescing optimization using a coalescing metadata cache with a size of 1. We coalesce all HD metadata updates to a single block and immediately flush the block when the 512th update is made. This means that the HD metadata block will be written sequentially with the data in the common case. In effect, this amortizes HD metadata updates across 512 writes.

By interspersing our metadata with data in the physical address space, we minimize write overheads but break the contiguous address space abstraction. To remedy this, the HD interface provides the illusion of contiguity in the HD address space exposed to users. Internally, it uses a static mapping between HD and physical address space. The HD address space is .2% smaller than the size of the physical address space, due to the HD metadata space overhead.

5.2.5 HD Metadata Persistency

In Section 5.2.4, we describe how we coalesce updates to a HD metadata block for contiguous addresses. This optimization introduces a problem: without writing the rest of the entries in the block, we have no reliable way of forcing HD metadata to be persistent. This opens us up to the possibility that in the case of a power failure, our HD metadata state on the storage device will be incorrect. Prior S3 solutions that cache/coalesce metadata claim that without a battery backup, they may lose metadata during a power outage [25, 26] or do not discuss a power loss scenario [57].

To address this problem, we extend the functionality of the sync command to flush our coalesced metadata block along with the disk caches. This allows users to enforce their own persistence policies for data and HD metadata. Prior implementations of versioning block drivers make all metadata persistent every 30 seconds [20]. We use a more conservative window of 1 second. This offers an improvement over prior works, which offered no way to guarantee metadata persistence, which can lead to security vulnerabilities and consistency issues.

5.3 Versioning System

5.3.1 Optimizing Version Metadata

A key data structure used by our VS is a logical-to-HD map. It maps a logical address to a set of HD addresses that contain versions of data. We use an append-only log to track updates to this map (Section 4). Each log entry is 8 bytes, and so we can fit $\frac{BlockSize-8}{8}$ logs in one storage block, which amounts to 511 entries per 4KB storage block.

We reserve 8 bytes in a metadata block for a pointer that allows us to implement a linked list of metadata blocks, enabling faster recovery. Before storing a version metadata block, VS reserves an HD address for the next metadata block. A pointer to this preallocated metadata block is stored in the actively written metadata block. However, this raises an issue as the current block will point to an invalid next block, which is yet to be written with valid metadata. This is a concern if the system crashes due to a power failure. We solve this problem by using the time-of-write. The nodes in the metadata linked list are chronologically ordered, except for the last unwritten pre-allocated metadata block. Thus, if we find that a metadata block was written before the parent during recovery, we have reached the end of the list.

5.3.2 Free-List of HD Blocks

The VS maintains a list of HD blocks in the free state. Initially, this is the set of all blocks in HD. On a write to a logical address, VS allocates an HD block from the free list to store a new version. The HD block containing the older version for that logical address is unfrozen and moved to a countdown list. Periodically, VS scans the countdown list to move HD blocks with an expired timelock to the free list.

5.3.3 Delayed Defragmentation

The VS maintains high write performance by issuing sequential writes to free up blocks. Over time, the set of free blocks will become fragmented, making sequential writes less frequent, which hurts performance. This problem is explained well by Rosenblum et al. in the design of the LFS [50].

The solution to fragmentation is a periodic "cleaning" of the log, which involves copying valid data to the head of the log, allowing the fragmented segment to be reclaimed. Our VS differs slightly from the LFS - because of the timelock constraints, we must wait for the timelock to expire before we can reclaim the fragmented segment. Our VS implementation features a timelock-aware defragmentation daemon that adheres to the constraint while reclaiming sequential addresses, thereby ensuring future performance remains fast.

6 Formally Verified Security Guarantees

We formally verify four primary properties of HD to prove strong security guarantees. Our proofs ensure: (1) our model of HD refines to the state machine seen in Figure 3b; (2) only the controller can directly modify HD metadata; (3) the controller never modifies the wrong metadata; (4) coalesced updates to metadata are correct and will be persisted. By proving these four properties, we ensure that timelock immutability is always guaranteed. To formally prove properties about HD we create a model of it in Dafny. Dafny is a programming language with a built-in automatic proof generation for the functional correctness of imperative code [29].

Refinement. The timelock state of each HD address is a function of the metadata that describes that block and the current time. For all possible metadata states and at all points in time, we compute the correct state and disallow state transitions not specified in the state machine. This proves that our model of HD refines the state machine seen in Figure 3b [2].

Protecting Metadata. While we have proven refinement, if an attacker can freely modify HD metadata, they can bypass the timelock logic altogether. To prove that only the controller can modify the HD metadata, we use Dafny's static type checking. We create a type for disk blocks based on their physical address and assign each one to be either a "data" block or a "metadata" block, based on a static mapping. Dafny's type-checking system enforces that methods capable of modifying HD metadata are invoked only by the controller.

No Side Effects. We prove that when a data block is modified, the controller appropriately updates the metadata for the HD address and no other metadata. This strategy for reasoning about shared mutable state has a similar structure to the techniques used in separation logic [47].

Persistency. Lastly, we prove that our coalescing/delayed write-back optimization correctly persists the right changes to metadata. We also prove that if this block is flushed, it is fully coherent with the disk and no data is lost.

6.1 Disk Model and Timer

Our proof for HD represents storage as an array of 4KB blocks (i.e., a 4KB array of byte objects). We model the storage interface commands for read and write with functions that copy blocks to/from our storage array. We assume that our model accurately reflects real storage read/write behavior. Converting our Dafny model into a real, executable program that can interface with a physical storage device requires replacing three lines of Dafny code with unverified, trusted methods that issue real ATA commands to a storage device, instead of copying data to/from our storage model.

To make correct decisions about block expiration and to ensure our time-of-write (ToW) guarantee, our controller must use a timer that increments monotonically at a constant and fixed rate. To implement this in our executable code, we utilize a trusted method that generates a timestamp for our program using the upper 32 bits of the RDTSC instruction on x86 processors and the upper 26 bits of the CNTVCT_EL0 instruction on ARM cores. By our calculations, this will not overflow for over 136 years. This *simulates* our ideal design of a non-volatile secure timer described in Section 3.2.

7 Evaluation

Our evaluation shows (1) HD has low overheads for I/O, execution, and bandwidth when evaluated across a variety of

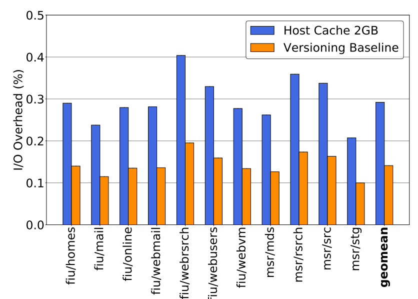

Figure 7a. Disk IO overhead for FIU and MSR traces.

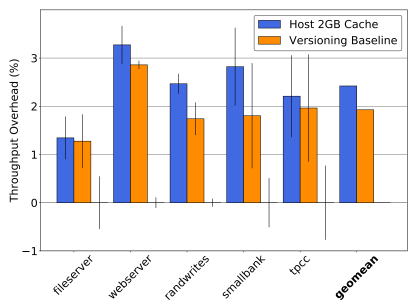

Figure 7b. Throughput overhead for Postgres and Filebench workloads.

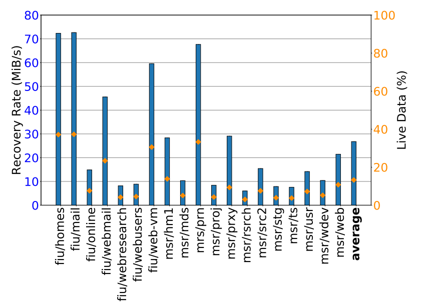

Figure 7c. Bars show the recovery rate of live versions. Diamonds indicate the percentage of all recovered data that is live.

Table 2: Workloads Used for Evaluating HD.

Name	Description
FIU [28]	20 days of Storage traces collected from FIU servers.
MSR [36]	6 days of Storage traces collected from Microsoft enterprise servers.
Filebench [54]	Benchmark framework designed to stress test filesystems. Chosen benchmarks simulate serving files, serving webpages, and random writes.
Postgres [42]	OLTP workloads running on Postgres database. Workloads generated include TPCC and SmallBank.

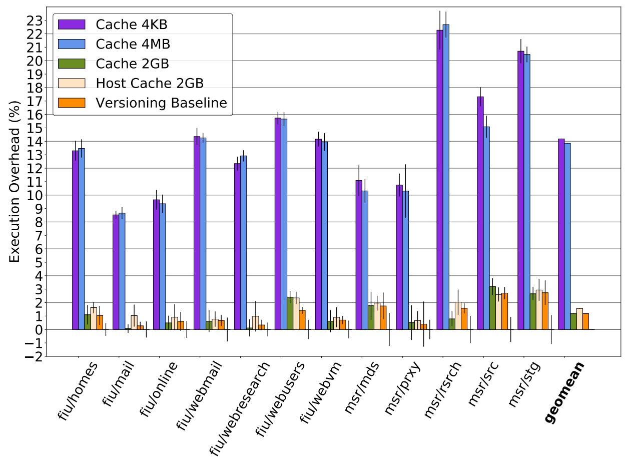

Figure 7: Execution overhead for FIU and MSR traces.

traces and workloads; (2) HD can recover all versions quickly after an attack; (3) The storage overhead of our HD metadata and version metadata is negligible.

7.1 Experimental Setup

To measure the performance of HD we use the workloads in Table 2. These include (1) traces collected from servers at Microsoft Research and the Computer Science department at FIU. (2) Filesystem benchmarks created using the Filebench framework [54], designed to simulate the workloads of a file server, a web server, and random writes. (3) OLTP workloads (TPC-C and SmallBank), generated using Benchbase [15] running on a Postgres database [42]. Before each experiment, we warm up the system by running a portion of the workload for one minute. We average all results from multiple runs and display the standard error where variance is present.

We have six configurations, their label is listed in brackets: [baseline] A standard storage system, [versioning baseline] A storage system that creates versions, [Host Cache 2GB] HelmsDeep, which manages versions, enforces timelock, and securely caches all HD metadata on the host, [Cache 2GB] Same as previous but all metadata is cached on the controller, [Cache 4MB] same as previous with 4MB of cache, [Cache 4KB] same as previous with 4KB of cache.

Our host machine is an Intel Xeon E3-1240 CPU, which communicates with the HD controller running on a Raspberry Pi 5 over an Ethernet connection. The storage device used in our evaluation is a 4TB Seagate ST4000DMZ04 hard disk.

7.2 Security Analysis

Our formal verification of the HD controller in Section 6 proves our transient immutability guarantee. To demonstrate our secure backup system, we deploy 18 ransomware samples from well-known "families" of ransomware attacks [27]. HD successfully recovered the state of the filesystem for every sample. Measuring the speed of recovery for this experiment is uninformative, as we do our recovery scan from the *start* of the log (Section 4.4). Thus, the recovery time for every sample is identical. Recovery speed in HD is a function of write intensity, which we explore in Section 7.4

7.3 Performance Analysis

We run a variety of I/O traces on HD to understand the performance overhead it adds compared to conventional storage. Each trace contains a list of reads and writes of different sizes to various addresses. We run the first million operations for each trace with no delay between operations.

7.3.1 I/O Overhead

While HD has 2x the I/O overhead of the versioning baseline as seen in Figure 6a, the actual performance difference between the two is quite small. Due to our layout of HD metadata (Figure 5) explained in Section 5.2.1, version metadata and HD metadata are written sequentially with data.

7.3.2 Execution Overhead

Figure 7 shows that for our versioning baseline, which has no security protections, the execution overhead is on average 1.5%. HD using the host cache adds only 0.3% overhead on top of the versioning baseline. Additionally, the two configurations are within a standard error of each other for all traces. We observe that when all metadata is cached on the controller (Cache 2GB), this overhead disappears, meaning the cost of HD is communicating the HD metadata with the host.

7.3.3 Caching HD Metadata

Due to the timelock constraint, there is no temporal locality for HD metadata. Therefore, there will be frequent misses unless the cache is large enough (2GB in our experiments) to fit all metadata blocks. This impact can be seen in Figure 7, where both the 4KB and 4MB caches incur significant overheads for all traces.

7.3.4 Throughput

In addition to the traces, we mount a filesystem on our versioning driver and run several applications to measure HD's impact on throughput. Versioning adds a small average throughput overhead of about 1.9%. HD on average adds an additional 0.5% overhead (Figure 6b).

7.4 Efficiency of Data Recovery

Figure 6c shows the recovery speed for HD. We can recover versions at a constant rate of 195 MiB/s. We only need the most recent checkpointed version before intrusion, which we call "live versions". The older (dead) versions are not helpful. Figure 6c shows the rate at which we recovered the live versions. To contextualize this rate, we also plot the percentage of live versions recovered out of all data read during the recovery. We observe that the recovery speed of live data is directly proportional to the fraction of live data in the system.

7.5 Storage Space Overhead

Storing data versions incurs space overhead; the VS decides which versions to keep or free. There is a rich body of literature on optimizing this overhead using techniques like delta-compression [10, 34], application-specific heuristics [25], etc. In our VS, the choice of checkpoint interval and rollback length impacts this space overhead. HD's timelock constraint does not further increase this versioning data overhead; instead, it *enforces* the VS's policy. Therefore, we do not consider HD responsible for the storage overhead of versioning.

HD introduces two sources of storage overhead. The first is from HD metadata. This overhead is a small constant fraction (.2%) of the physical storage space. The second is version metadata; in a VS without timelock guarantees, version metadata can be reclaimed along with old versions.

However, because HD must protect the version metadata with timelock guarantees, we used an append-only log (Section 4.3), which is not efficient to free instantly. Instead, we perform an infrequent garbage collection (user-defined) for version metadata. This results in a space overhead due to stale VS metadata. We observed that across all 18 traces we analyzed, this overhead never exceeded .5 GB of total space, even for the write-intensive 20-day-long traces.

8 Related Work

This paper is the first to isolate timelock checks from versioning. HD enables a small TCB, where even the versioning software is untrusted, which was not before shown to be feasible. We also formally verify security guarantees of HD.

Backups and Versioning. The traditional ransomware defense is data backups. Common policies include versioning [13, 20, 40], snapshotting [11], and delta backups [10, 12, 61]. These solutions are implemented in software and are thus vulnerable to common malware attacks that compromise the OS [25, 38], and are thus inadequate for ransomware defense. However, the versioning policies and optimizations they use can be implemented in HD's untrusted VS.

Several prior works use the intrinsic out-of-place write structure of an SSD to implement data versioning that defends against ransomware without trusting the OS [25, 26, 57]. These works modify garbage collection in the flash transition layer (FTL) to retain old versions of data and enable rollbacks. BVSSD [26] retains all versions indefinitely, which implements a continuous data protection policy [59, 62] inside an SSD. Storage on an SSD is finite, which is why RSSD [44] modifies FTL garbage collection to backup flash blocks to the cloud through NVMe over Ethernet before erasing them, providing the illusion of infinite capacity. To truly address capacity limitations, we must eventually reclaim space. FlashGuard [25] and Project Almanac [57] allow for garbage collection and modify the FTL to add time-delayed versioning.

The use of time as a defense in FlashGuard and Project Almanac is similar to our work, however, we differ in several key ways. Both works use time retention as a security policy in their versioning logic. We are the first to show that time retention can be decoupled from versioning, and that untrusted versioning software can use the HD interface to build a secure backup system. Additionally, because both prior works implement versioning in the FTL, the entire modified firmware is included in the TCB. Verifying that changing the FTL does not introduce vulnerabilities is a significant challenge. Tripathy et al. formally verify several, *but not all*, properties of a ransomware-resistant FTL [56], improving confidence in the correctness of FTL-based solutions.

RSSD shows that prior S3 works, including FlashGuard and Project Almanac, are vulnerable to a "trimming attack" that bypasses security checks in the FTL to erase data [44]. In contrast, our work decouples versioning from the TCB, which is a non-trivial feat as explained in Section 4.3. Our isolated HD is only responsible for enforcing timelock guarantees and is separated from the FTL, making it easier to prove strong security guarantees. We prove that HD has no equivalent to the "trimming attack" [44].

Integrating versioning and timelock checks within SSD firmware makes it harder to change the versioning and timelock policies, which in turn restricts their use cases. By excluding versioning from the TCB, our approach enables easy updates to versioning policies and optimizations in untrusted software, without altering TCB code or risking security vulnerabilities. As a result, HD enables several uses beyond encryption ransomware, as detailed in Section 3.4.

Commercial products, such as Exagrid and Oracle, implement "retention timelocks" for ransomware defense in databases [52] and object stores [18], by retaining data for a duration in the application backup logic. In contrast, our approach isolates timelocks from versioning logic, resulting in a minimal TCB that improves security and flexibility.

Ransomware Detection and Prevention. A different defense against ransomware is to detect and prevent further damage. These approaches monitor I/O accesses and data contents to identify patterns commonly found in ransomware [7,35,39]. However, even if an attack is detected and halted, some files are already corrupted [44]. Additionally, while these techniques may work for existing ransomware, attackers aware of the heuristics and detection models can create new ransomware that bypasses detection. Ransomware detection is an essential first step in speedy recovery, but detection alone cannot prevent data loss. Secure backups are needed to complement detection.

Replay for Forensics Analysis. To answer questions about system state before an attack [14] or replay the events leading to an attack [17], a system can create logs of all events and store them on a disk. With these logs, an intrusion analysis [60] can determine how to improve system defenses and generate patches for the vulnerabilities discovered. However, ransomware may encrypt or delete these logs to prevent users from analyzing the attack. HD can be used to enhance intrusion analysis by protecting logs with timelocks.

Time for Defending Reads. Time-release cryptography enables publicly sharing encrypted data that is decrypted at a predetermined time in the future [33,49]. This is used in applications such as sealed bidding and digital time capsules [33,49]. This can be implemented with trusted third parties or computational puzzles that act as a proxy for time [33,49]. HD utilizes time to protect data integrity (writes), while time-release cryptography uses time for confidentiality (reads). The two can be used together to obtain both benefits.

9 Conclusion

While significant advancements have been made to address hardware and software security vulnerabilities, solutions targeting human errors remain limited. Given that nearly two-thirds of ransomware attacks exploit human weaknesses, there is a pressing need for defenses that do not rely solely on managing data access through user credentials.

HelmsDeep (HD) is a significant step toward achieving this goal. It is the first work to isolate the timelock defense completely, enabling the construction of a secure backup system with the smallest known TCB — one that doesn't trust the user or the VS. The HD interface is simple enough to allow formal verification, yet flexible enough to support a wide range of versioning optimizations and data retention policies. These benefits come with negligible performance and storage overhead, making HD an efficient and robust solution.

References

[1] M-trends 2024 special report. Technical report, Mandiant, 2024.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[3] Steve Alder. Ascension ransomware attack hurts financial recovery. 2024. `https://www.hipaajournal.com/ascension-cyberattack-2024/`.

[4] Omar Alhazmi, Yashwant Malaiya, and Indrajit Ray. Security vulnerabilities in software systems: A quantitative perspective. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 281–294. Springer, 2005.

[5] Zainab Alkhalil, Chaminda Hewage, Liqaa Nawaf, and Imtiaz Khan. Phishing attacks: A recent comprehensive study and a new anatomy. *Frontiers in Computer Science*, 3:563060, 2021.

[6] Mohamed Alsharnouby, Furkan Alaca, and Sonia Chiasson. Why phishing still works: User strategies for combating phishing attacks. *International Journal of Human-Computer Studies*, 82:69–82, 2015.

[7] SungHa Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and DaeHun Nyang. Ssd-insider: Internal defense of solid-state drive against ransomware with perfect data recovery. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 875–884. IEEE, 2018.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[9] Simon Biggs, Damon Lee, and Gernot Heiser. The jury is in: Monolithic os design is flawed: Microkernel-based designs improve security. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–7, 2018.

[10] Randal C Burns and Darrell DE Long. Efficient distributed backup with delta compression. In *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 27–36, 1997.

[11] Hoi Chan and Trieu Chieu. An approach to high availability for cloud servers with snapshot mechanism. In *Proceedings of the industrial track of the 13th ACM/IFIP/USENIX international middleware conference*, pages 1–6, 2012.

[12] Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. Protecting file systems: A survey of backup techniques. In *Joint NASA and IEEE Mass Storage Conference*, volume 99. Citeseer, 1998.

[13] Brian Cornell, Peter A Dinda, and Fabián E Bustamante. Wayback: A user-level versioning file system for linux. In *Proceedings of Usenix Annual Technical Conference, FREENIX Track*, pages 19–28, 2004.

[14] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 525–540, 2014.

[15] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[16] Nesara Dissanayake, Mansooreh Zahedi, Asangi Jayatilaka, and Muhammad Ali Babar. Why, how and where of delays in software security patch management: An empirical investigation in the healthcare sector. *Proceedings of the ACM on Human-computer Interaction*, 6(CSCW2):1–29, 2022.

[17] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[18] ExaGrid. Retention time-lock for ransomware recovery. https://www.exagrid.com/wp-content/uploads/ExaGrid-Retention_Time-Lock_for_Ransomware_Recovery_DS.pdf, 2023.

[19] Alberto Faria, Ricardo Macedo, José Pereira, and João Paulo. Bdus: implementing block devices in user space. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–11, 2021.

[20] Michail Flouris and Angelos Bilas. Clotho: Transparent data versioning at the block i/o level. In *MSST*, pages 315–328. Citeseer, 2004.

[21] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. Vanish: Increasing data privacy with self-destructing data. In *USENIX security symposium*, volume 316, pages 10–5555, 2009.

[22] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 9–9. IEEE, 2005.

[23] Brij B Gupta, Aakanksha Tewari, Ankit Kumar Jain, and Dharma P Agrawal. Fighting against phishing attacks: state of the art and future challenges. *Neural Computing and Applications*, 28:3629–3654, 2017.

[24] Ryan Heartfield and George Loukas. A taxonomy of attacks and a survey of defence mechanisms for semantic social engineering attacks. *ACM Computing Surveys (CSUR)*, 48(3):1–39, 2015.

[25] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K Qureshi. Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2231–2244, 2017.

[26] Ping Huang, Ke Zhou, Hua Wang, and Chun Hua Li. Bvssd: Build built-in versioning flash-based solid state drives. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, 2012.

[27] Mohsen Khashei. Ransomware-samples. https://github.com/kh4sh3i/Ransomware-Samples, 2022.

[28] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Trans. Storage*, 6(3), September 2010.

[29] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[30] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.

[31] Jochen Liedtke. On micro-kernel construction. *ACM SIGOPS Operating Systems Review*, 29(5):237–250, 1995.

[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[33] Jia Liu, Tibor Jager, Saqib A Kakvi, and Bogdan Warinschi. How to build time-lock encryption. *Designs, Codes and Cryptography*, 86:2549–2586, 2018.

[34] Josh MacDonald. *File system support for delta compression*. PhD thesis, Masters thesis. Department of Electrical Engineering and Computer Science . . . , 2000.

[35] Donghyun Min, Donggyu Park, Jinwoo Ahn, Ryan Walker, Junghee Lee, Sungyong Park, and Youngjae Kim. Amoeba: An autonomous backup and recovery ssd for ransomware attack defense. *IEEE Computer Architecture Letters*, 17(2):245–248, 2018.

[36] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical power management for enterprise storage. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[37] Justin D Osborn and David C Challener. Trusted platform module evolution. *Johns Hopkins APL Technical Digest (Applied Physics Laboratory)*, 32(2):536–543, 2013.

[38] Harun Oz, Ahmet Aris, Albert Levi, and A Selcuk Uluagac. A survey on ransomware: Evolution, taxonomy, and defense solutions. *ACM Computing Surveys (CSUR)*, 54(11s):1–37, 2022.

[39] Jisung Park, Youngdon Jung, Jonghoon Won, Minji Kang, Sungjin Lee, and Jihong Kim. Ransomblocker: A low-overhead ransomware-proof ssd. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[40] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage (TOS)*, 1(2):190–212, 2005.

[41] James S Plank, Jian Xu, and Robert HB Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical report, Citeseer, 1995.

[42] PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org, 2008.

[43] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[44] Benjamin Reidys, Peng Liu, and Jian Huang. Rssd: Defend against ransomware with hardware-isolated network-storage codesign and post-attack analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 726–739, 2022.

[45] Charles Reis and Steven D Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 219–232, 2009.

[46] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1661–1678, 2019.

[47] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[48] John Riggi. A look at 2024's health care cybersecurity challenges. 2024. https://www.aha.org/news/aha-cyber-intel/2024-10-07-look-2024s-health-care-cybersecurity-chall

[49] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

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

[51] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Van Doorn, John Linwood Griffin, Stefan Berger, Reiner Sailer, Enriquillo Valdez, Trent Jaeger, et al. shype: Secure hypervisor approach to trusted virtualized systems. *Techn. Rep. RC23511*, 5, 2005.

[52] Kelly Smith. Retention lock your cloud database backups for increased ransomware protection. https://blogs.oracle.com/maa/post/retention-lock-for-increased-ransomware-protection, 2023.

[53] John D Strunk, Garth R Goodson, Michael L Scheinholtz, Craig AN Soules, and Gregory R Ganger. Self-securing storage: Protecting data in compromised systems. In *OSDI*, pages 165–180, 2000.

[54] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *login Usenix Mag.*, 41, 2016.

[55] The Linux Kernel Documentation. *Page Table Isolation (PTI)*. Documentation for PTI (Page Table Isolation) on x86.

[56] Shivani Tripathy, Debiprasanna Sahoo, Manoranjan Satpathy, and Madhu Mutyam. Formal modeling and verification of security properties of a ransomware-resistant ssd. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(8):2766–2770, 2022.

[57] Xiaohao Wang, Yifan Yuan, You Zhou, Chance C Coats, and Jian Huang. Project almanac: A time-traveling solid-state drive. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[58] Sophos Whitepaper. The state of ransomware in healthcare 2024. Technical report, 2024. https://assets.sophos.com/X24WTUEQ/at/4bk9xt4h7gsm4xs6mfzh3k/sophos-state-of-ransomware-healthcare-2024.pdf.

[59] Weijun Xiao, Jin Ren, and Qing Yang. A case for continuous data protection at block level in disk array storages. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):898–911, 2008.

[60] Yulai Xie, Dan Feng, Zhipeng Tan, and Junzhe Zhou. Unifying intrusion detection and forensic analysis via provenance awareness. *Future Generation Computer Systems*, 61:26–36, 2016.

[61] Yucheng Zhang, Ye Yuan, Dan Feng, Chunzhi Wang, Xinyun Wu, Lingyu Yan, Deng Pan, and Shuanghong Wang. Improving restore performance for in-line backup system combining deduplication and delta compression. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2302–2314, 2020.

[62] Ningning Zhu and Tzi-cker Chiueh. Portable and efficient continuous data protection for network file servers. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 687–697. IEEE, 2007.