Time Machine: An Efficient and Backend-Migratable Architecture for Defending Against Ransomware in the Hypervisor

ABSTRACT

Ransomware has caused escalating financial losses for individuals and companies, increasing annually. To combat this, we present Time Machine, a real-time, fine-grained sector-level live view navigation solution designed to safeguard filesystems from ransomware attacks at the hypervisor level. Time Machine offers several key advancements over existing solutions. Operating at the hypervisor level minimizes the risk of bypassing via privilege escalation and eliminates reliance on hardware-based solutions. Time Machine redirects I/O operations without altering the original storage disk. Utilizing local or cloud-based key-value store backends, it offers flexible storage spaces for live view navigation and the capability of backend migration. This approach ensures comprehensive filesystem protection without data loss, allowing users to browse and recover data to any specific timestamp. Time Machine is designed to operate independently of detection algorithms but can also integrate with them for enhanced protection. Evaluation results demonstrate that our prototype effectively safeguards the filesystem with minimal overhead. With a 256MB memory cache and affordable storage, Time Machine successfully defends against 12 ransomware variants on Windows and Linux platforms, incurring an average runtime overhead of less than 5%.

CCS CONCEPTS

• **Security and privacy → File system security**; **Malware and its mitigation**; **Usability in security and privacy**.

KEYWORDS

Cloud outsourcing, Filesystem protection, Hypervisor, Ransomware, Time Machine

1 INTRODUCTION

Ransomware remains one of the most prominent threats faced by individuals and enterprises, as highlighted by the global epidemic of Wannacry, which affected approximately 230,000 computers in 2017 [13]. According to a 2024 report from the World Economic Forum [35], ransomware activity increased by 50% during the first half of 2023, fueled by the rise of Ransomware-as-a-Service (RaaS). The accessibility of RaaS, with prices as low as $40, has democratized ransomware attacks, enabling even novice threat actors to carry them out.

Similarly, SOPHOS's 2023 global cross-industry ransomware survey [22] revealed alarming trends. The average ransom payment doubled from $812,380 in 2022 to $1,542,333 in 2023. Despite widespread distribution, only 34% of attackers demanded less than $100,000. Governments and numerous companies have increasingly become targets of ransomware attacks, with 66% of surveyed organizations reporting incidents. Using cryptocurrency for ransom payments ensures anonymity and facilitates cross-border extortion [11]. In 2023, ransomware payments exceeded 1 billion dollars, reaching record highs. As the attack landscape evolves, encompassing a broader array of devices, it becomes imperative to bolster defenses against ransomware.

This study discusses crypto-ransomware, which encrypts files in disk drives and requests ransoms from victims to unlock files. While it is infeasible to unlock encrypted files without a key, statistics [22] showed that 76% of victims suffered from data encryption, and 46% of victims paid the ransom to regain their data. Existing ransomware defense solutions face various challenges. Backup-based approaches may require a lot of storage spaces, and it may fail to protect every change instantly. Shadow-based approaches usually keep only the latest record. The offered protection granularity might be insufficient for recovering user data stored between the oldest and the encrypted filesystem state. Detection-based approaches may suffer from detection lag before an attack is conducted, and be vulnerable to unknown types of ransomware. Hardware-based approaches require additional hardware, which may restrict the scalability of the approaches and bring additional costs. Filesystem-based approaches (snapshot and log-based) usually implements in the same filesystem targeted by attackers. Attackers could remove the snapshots or log records. Nevertheless, all solutions deployed within the same machine compromised by ransomware could be bypassed if the system suffers from privilege escalation attacks.

In this paper, we propose Time Machine, a real-time, fine-grained sector-level live view navigation service with migratable backend storage for virtual machines to defend against crypto-ransomware. Time Machine leverages the hypervisor to monitor and redirect I/O payloads for real-time navigation. Instead of using local storage, we propose using a migratable key-value store as the backend storage of our proposed approach. The design of Time Machine allows the backend storage to be outsourced to a cloud service, providing even more robust and durable protection against ransomware attacks. The backend storage is composed of two parts: a *cache* in non-persistent storage (e.g., memory) and a *collection* in persistent storage (e.g., disks). Redirected I/O payloads are segmented into units of *sectors*, where a sector index serves as the key to the backend storage, and the creating timestamp and content is the corresponding record value.

Based on the proposed design, we successfully tackle the aforementioned challenges faced by existing solutions. Compared to backup-based approaches, segmenting data into smaller units makes it easier to optimize storage space. The I/O redirection design allows performing backup in real-time. Compared to shadow-based approaches, our approach preserves multiple copies of sectors of different timestamps, ensuring the filesystem state can be restored to any specific timestamp. Our proposed approach is not required to perform detection. Therefore, neither detection lag nor false negatives need to be considered. Our proposed approach is a software-based solution. Also, integrating our approach at the hypervisor level prevents it from being compromised by ransomware running

on the same OS. It can seamlessly protect any existing virtual machines without requiring modifications to the configuration of the guest OS, ensuring compatibility and ease of deployment. Besides the proposed novel design, we attempt to perform various optimizations to minimize the required backed storage spaces and improve the overall I/O performance.

Our proposed Time Machine can be deployed to various infrastructures. The most typical application is on servers providing virtual machines, which are widely used for web services, databases, and compute engines. Integrating the approach on base servers can extend robust protection to every virtual machine on them. Additionally, services offering remote desktop access, such as Windows 365, can integrate our approach to extend protection from the server side to the client side. Furthermore, local machines can be included by deploying their operating systems on hypervisors. As mentioned earlier, the flexibility of our database design allows our system to fit diverse infrastructures, from remote to local, based on user demands.

The rest of the paper is organized as follows. We introduce research works to detect ransomware and approaches to recover from ransomware attacks in Section 2. We present our proposed approach in Section 3, the implementation of the system in Section 4, and the evaluation of its performance in Section 5. A concluding remark is finally given in Section 6.

2 RELATED WORK

2.1 Ransomware Detection Research

Encryption imposes heavy computational loads on processors and introduces delays in file reading and complete disk erasure. Numerous ransomware detection systems [1–3, 25, 37] have been proposed to differentiate ransomware activities from benign software, aiming to identify and halt malicious operations promptly. Ransomware typically exhibits distinct behaviors compared to benign software, such as frequent file access, high-throughput encryption, and suspicious API calls. Machine learning approaches have been introduced into ransomware detection to enhance accuracy and reduce false positive rates. Since each detection approach is tailored for a specific execution platform and privilege level, they leverage different software features to distinguish ransomware. Both benign and malicious programs utilize encryption, memory management, file system, and network APIs, leading many detection systems to rely on API call parameters, sequences, and results. Furthermore, some studies [6, 32, 39, 41] suggest that machine learning-based approaches using Opcode sequences as features yield highly accurate detection. Encrypted files often exhibit relatively high entropy, a crucial characteristic of crypto-ransomware. While compression can also produce high entropy output, entropy is commonly employed as a feature in many detection systems [8, 15–17, 21, 26, 27, 36]. Additionally, certain ransomware establishes connections to Command and Control (C&C) servers to retrieve encryption keys. This behavior may be accompanied by data exfiltration, with domain names, IP addresses, packet counts, and packet payloads aiding in the construction of more accurate classifiers. Our work offers an efficient recovery mechanism that operates independently of detection functionality. Depending on specific performance requirements, users have the flexibility to combine various detection mechanisms.

2.2 Decryption-based Recovery

While ransomware detectors strive to halt attacks as swiftly as possible, some victim files may still become encrypted since distinguishing ransomware before an attack commences is often challenging. Decryption-based approaches have been proposed to rescue files without the need for a ransom payment. Notably, *No More Ransom* [10] is a reputable decryption service that has aided countless victims in file recovery. However, this service is only effective for known ransomware samples, and specific vulnerabilities within the ransomware design are required for decryption to succeed, such as breakable cryptographic schemes, predictable encryption key generation, or encryption key reuse. Additionally, a decryptor must be either leaked or released by the attackers or security analysts. PayBreak [19] and RWGuard [26] decrypt locked files by intercepting keys through hooks on crypto libraries like Microsoft Crypto API and Crypto++. Unfortunately, these methods are ineffective against ransomware employing self-implemented cryptographic functions.

2.3 Backup-based Recovery

ShieldFS [8] prevents files from being modified by redirecting malicious writes to a separate process space, ensuring zero file losses. However, it also introduces a performance overhead of 0.26x. Redemption [17] redirects I/O requests to a protected area and synchronizes writes based on their malicious scores, making the system resilient to new types of ransomware. CLDSafe [40], designed for cloud servers, offers a backup strategy that determines whether to preserve older file versions by assessing the similarity between the versions. Paik et al. [30] conceal backup storage from ransomware using the alternative data stream provided by NTFS, reducing the data transfer required for backing up files on remote machines. However, this approach still requires a huge disk capacity to store the backup in the same filesystem. Time Machine addresses these issues while retaining their advantages. It provides real-time protection and conceals the backend storage from ransomware, all while introducing minimal overhead and requiring only a small amount of additional storage.

2.4 Existing Filesystem Features

Several modern filesystem features can be used to counter ransomware attacks. Copy-on-write filesystems, such as Btrfs and ZFS, offer efficient snapshot capabilities that can be a basis for restoring data after a ransomware attack. When files are deleted or modified, these filesystems preserve the old data and only record the differences between the snapshot and new versions. However, snapshots have to be taken manually or periodically. Log-structured filesystems (LFS) [34] also natively preserve historical data. Since the physical distance between target sectors affects the writing speed to optical and magnetic disks, LFS maximizes sequential writes. An LFS maintains a log of the filesystem's history. Instead of replacing old data, new data and inodes (metadata) are appended to the end of the log. The latest inode's location is kept in an inode map, which is compact enough to be cached in memory. LFS can easily establish snapshots by keeping references to old inodes and data in the log [29]. NILFS [20], an implementation of a log-structured filesystem, supports creating snapshots with each synchronous

write. However, LFS requires garbage collection to recycle space storing outdated data, which deteriorates the overall performance and suffers from severe fragmentation over time [33]. Compared to Time Machine, relying on filesystem-based snapshots to defend against ransomware limits the sizes and the location of snapshot storage. Additionally, it is not secure enough since ransomware can remove snapshots if it gains admin privileges. Time Machine addresses these problems with hypervisor isolation and a migratable sector-level backend storage.

2.5 Hardware Supported Recovery

Ransomware defense systems typically ensure their authenticity by implementing functionality in kernel space, making them vulnerable to kernel attacks. To fully protect every file and every change, these systems often introduce high overhead or require substantial additional storage for backups. Several works [4, 5, 12, 27, 31] have addressed these issues using the out-of-place update characteristic of solid-state drives (SSDs). Embedding ransomware defenses in the SSD controller ensures the system is isolated from software and functions correctly, with outdated data awaiting erasure serving as a natural backup. This approach results in minimal backup and restoration overhead. However, the lack of rich context information limits the accuracy of detection systems embedded in the firmware. RansomTag [23] further migrates the detection mechanism to the hypervisor, leveraging the introspection capability of virtual machines to provide more accurate predictions using information such as file systems and network activities. Nonetheless, working with specific hardware incurs additional cost, making them less economical than traditional storage solutions.

3 APPROACH

3.1 Threat Model and Assumptions

Our threat model assumes that ransomware employs various techniques used by other types of malware to execute attacks. To evade detection, ransomware may encrypt only portions of files instead of entire files. It may exploit vulnerabilities to compromise guest systems, steal data, spread malware, and even act as a zombie within botnets. Additionally, some ransomware attempts to bypass detection and protection mechanisms through privilege escalation attacks. By obtaining administrator permissions, ransomware can bypass kernel-based defense systems or target snapshot data stored on storage devices. Furthermore, ransomware may disrupt Virtual Machine Introspection (VMI) techniques by obstructing the acquisition of accurate OS context information, thereby hindering VMI-based detection approaches. We also assume that attackers may possess prior knowledge of the detection and protection algorithms used in the system and attempt to circumvent them.

Given that our approach operates within a hypervisor environment, we assume the hypervisor itself, the associated remote servers, and the data storage on the VM host are trusted and protected against unauthorized access or modification. Conversely, the guest operating system is untrustworthy and could be compromised.

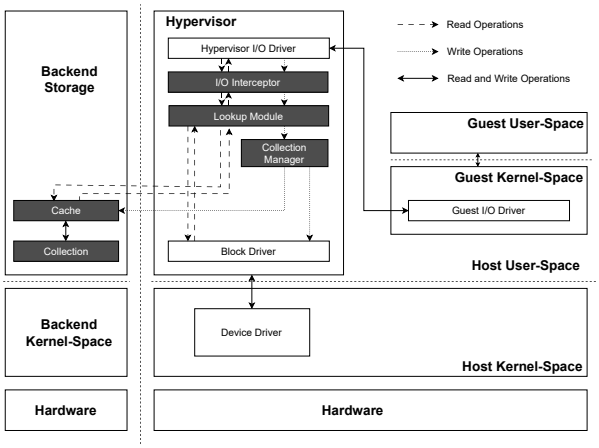


Figure 1: Overall Architecture.

3.2 Architecture Overview

The study aims to develop a sector-level live view navigation solution within the hypervisor with minimal overhead and manageable capacity requirements. Its objective is to monitor guest I/O behavior on the hypervisor and reroute I/O operations to the backend storage system, ensuring that any I/O activity by the guest does not directly impact the guest's hard disk. Recorded data encompasses the sector of write requests, timestamps, and the contents of write requests. For guest read requests, the system retrieves content from either the backend storage system or the hard disk, depending on the presence of the requested sector's content in the backend storage system. By logging all I/O behavior, the system enables browsing of the guest hard drive's state at any given timestamp and facilitates complete restoration to its state at that time.

Figure 1 illustrates the overall architecture of Time Machine. When the I/O driver of the guest OS initiates a read request, it is initially intercepted by the I/O interception mechanism within the hypervisor. This interception process involves dividing the request into sectors. The lookup module plays a pivotal role as a classifier, determining whether the sector-level read request should be routed to the collection, which serves as the backend storage in our proposed system, or to the disk driver, based on the existence of the requested data within the collection.

With the assistance of the lookup module, querying data from the collection becomes unnecessary if there is no corresponding information about the requested sector. For write requests, as they are transmitted to the hypervisor, they are also intercepted by the I/O interception mechanism and segmented into sectors. Subsequently, these requests are redirected to the collection, where the sector's value is recorded along with its timestamp. Notably, the collection manager oversees the maintenance of collection size, aiming to minimize storage requirements under normal and extreme conditions.

Before forwarding explicit requests to the collection, the collection manager evaluates factors such as the target sector, request timestamp, and information from the lookup module. It decides whether to record the request, replace old values with current ones,

or directly inform the disk for actual disk access. Furthermore, if the collection exceeds a predetermined threshold, it optionally updates data from a specified period to the disk and removes corresponding data from the collection.

To expedite read/write requests, we implement a key-value store caching strategy within the collection. This cache records the most recently accessed values using the requested sector as the cache key. When a sector's value is requested, the system first checks the cache for the result before querying the key-value store collection. This approach accelerates response to requests by minimizing the need to query the collection for frequently accessed data.

The main goal of this work is to facilitate the restoration of the disk to any specified timestamp. Prior to initiating the system recovery process, Time Machine introduces a READ mode, enabling users to navigate the filesystem at a designated timestamp. The operational framework of the READ mode closely resembles the original one, with the exception of the read content and the treatment of write requests. When handling read requests in READ mode, the system retrieves the most recent data preceding the given timestamp from the collection to present the filesystem as it appeared at that particular moment. It's crucial to note that in READ mode, all write requests are rejected to prevent any alterations to the content of both the collection and the disk.

Additionally, Time Machine offers a RESTORE mode to revert the state of the disk to a specified timestamp. Users can conveniently examine the filesystem's state at any given timestamp using the READ mode, allowing them to employ Time Machine's RESTORE mode effectively to revert the disk to that specific timestamp.

Please note that the design of Time Machine supports two distinct deployment models. The backend storage components depicted in Figure 1 can be located within the host operating system or the cloud environment. These deployment models alleviate constraints on storage spaces and provide more robust setups with limited performance overhead. In the subsequent subsections, we comprehensively explain how the performance and storage spaces are optimized in each module within the overall architecture.

3.3 Enhancing I/O Speed

As previously mentioned, within our framework, an I/O request is redirected to the collection. However, a drawback arises from the framework's inability to determine whether information for the requested sector exists in the collection. For example, in the case of a read request at the sector level, the framework must initially check the collection for the data. If the data is not found, the request is forwarded to the disk driver to access the value. With every request issued by the guest OS to the hypervisor, the framework is required to search for data in the collection, regardless of its presence.

To tackle this issue, Time Machine introduces a lookup module designed to manage the presence status of data within the collection. This module utilizes a bitmap to indicate which sectors are stored in the collection. Each bit in the bitmap corresponds to the presence of a sector, with a value of one indicating its existence. When a sector-level write request is intercepted by the I/O interception module and marked for recording in the collection, the corresponding bit in the bitmap is set to denote its presence. With the lookup module implemented, any request or operation no longer needs to query the collection for values if the bitmap indicates the absence of the sector. Interestingly, this mechanism involves a trade-off between time and space. Each bit in the bitmap can also represent multiple sectors, such as blocks. If any sector within a group of sectors is not recorded in the collection, the grouped sectors are still considered present, i.e., the bit mapped to the grouped sectors is set. In this scenario, the system initially queries the collection for data; if no results are found, the request is then forwarded for actual disk access. While this approach reduces the space required for the bitmap, it also introduces additional overhead.

An additional enhancement aimed at improving I/O speed is employing a cache strategy within the collection module of Time Machine. This involves integrating a cache system preceding the key-value store collection, which enhances the efficiency of queries made to the collection during I/O request handling. The cache system is implemented using an in-memory key-value store database. Similar to the key-value store collection, the cache system can be locally stored within the host OS or hosted on a remote server to leverage the benefits of cloud storage. In operation, the cache system utilizes the requested sector as the cache key and stores the most recently accessed values. When a request queries the key-value store collection and retrieves the required values, these sectors and values are cached within the cache system. Subsequent requests for the same sector to the key-value store collection can then directly obtain values from the cache system, thus bypassing the need for repeated queries. Additionally, any new records added to the key-value store collection are cached, ensuring that the cache system contains the latest values for the sectors. In practice, the cache system is allocated a memory limit to prevent resource exhaustion. It retains recently accessed keys and evicts the least recently used keys if the memory limit is at risk of being exceeded.

3.4 Maintaining Collection Size

This section discussed various automated and manual approaches that can be employed to manage and maintain the collection size of the backend storage.

3.4.1 Optional Auto-Commit for Releasing Collection Size. The collection manager module oversees the size of the key-value store collection. However, the space available for the key-value store collection is limited. Despite the ample space provided by cloud storage, it is not unlimited. Therefore, the system employs an auto-commit mechanism and establishes three thresholds as criteria for assessment.

Before adding a record to the collection, the system checks if the collection size exceeds a predefined notification threshold or upper bound threshold. Upon surpassing the upper bound threshold, the system initiates a process to write a specific period of data back to the disk at a time, starting from the oldest data, until the collection size is smaller than the initial threshold. Subsequently, the corresponding record is removed from the collection.

If the collection size exceeds the notification threshold, a notification is sent to the user, prompting them to inspect the file system to free up space in the collection. In the prototype implementation, the initial threshold, notification threshold, and upper bound threshold are set to 20 GB, 25 GB, and 30 GB, respectively. This auto-commit mechanism reclaims excess space when the collection

size becomes too large. However, the data being written back to the disk is not guaranteed to be unencrypted. Extended operational periods or intentional attacks leading to excessive redundant operations could breach threshold limits, resulting in the update of encrypted content to the disk.

Note that the auto-commit feature is optional because ransomware may escape detection and fill up the backend storage. In a typical usage, the collection should be filled slowly and a regular manual commit would work for releasing collection spaces. When the feature is disabled, all write operations to the drive are rejected when the collection is full and the user is notified.

3.4.2 Bitmap for Handling Unused Blocks. To counteract the impact of targeted attacks aimed at filling up the collection, Time Machine implements protective measures within the collection manager module. Firstly, it introduces the concept of a block bitmap in the EXT4 file system. This bitmap maintains a record of the usage status of blocks on the disk. Any write request targeting an unused block can directly access the disk without being logged into the collection. This prevents attackers from compromising storage efficiency by filling up unused blocks, such as creating files or downloads. Each bit in the bitmap corresponds to the status of a single data block.

In cases where the block bitmap has not been initialized, Time Machine examines the disk to ascertain the filesystem's usage status before booting the guest system. Alternatively, it retrieves the block usage status from the existing bitmap. When a write request is directed at an unused block, Time Machine marks it as one in the bitmap. Consequently, subsequent requests pertaining to the block can be logged to prevent data loss.

3.4.3 Request Consolidation for Saving Collection Size. Attackers may still attempt to continuously write to existing content blocks, i.e., data blocks marked as one in the block bitmap. As these requests are all logged in the collection, they effectively increase its size, potentially leading to overflow and requiring the flushing of collection content back to the disk. Despite the hypervisor's potential utilization of write coalescing or other optimization techniques to enhance efficiency and reduce overhead, consistent writing to used blocks could still result in collection overflow, albeit possibly requiring more write operations to achieve. Consequently, the second protection mechanism is devised to streamline existing content block records by merging records with matching sectors and requesting timestamps falling within a designated time threshold.

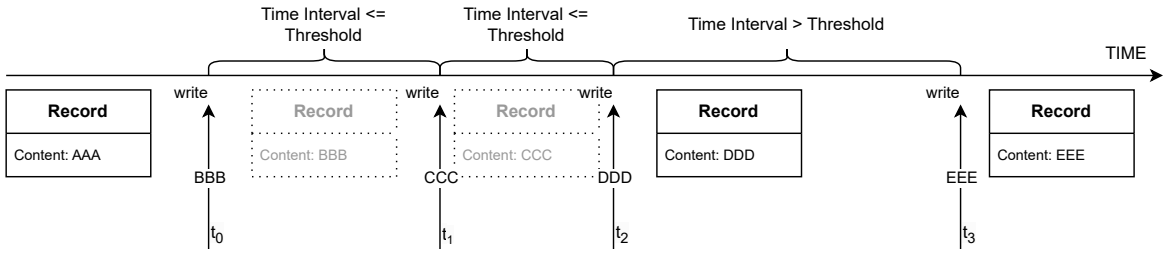
Time Machine provides support for two strategies to merge records within the key-value store collection. The first strategy utilizes the concept of interarrival time: when two records relating to the same sector have a time interval between them within a specified threshold, the system merges them by retaining the more recent record and removing the older one. The second strategy involves slicing time segments: fixed time segments are continuously connected in a series. For each sector, only the most recent record within a time segment is retained. All other records, except the newest one, are removed from the key-value store collection. These strategies mitigate the impact of multiple write requests to the same sector and help reduce the overall collection size. Figure 2 shows the schematic diagram illustrating the two strategies.

- **Interarrival Time Strategy** Figure 2a demonstrates the strategy utilizing interarrival time. Let us consider a scenario where there exists a record with the value 'AAA' stored for a specific sector in the key-value store collection. At timestamp t_0, a write request was recorded with the value 'BBB'. Subsequently, at timestamp t_1, a write request with the value 'CCC' is transmitted to the collection manager module. Given that the time interval between t_0 and t_1 falls within the defined threshold, the record with the value 'BBB' is first removed from the collection, followed by the insertion of the new record with the value 'CCC'. Similarly, records with the value 'CCC' are deleted before inserting the new record with the value 'DDD' since the time interval between t_1 and t_2 is smaller than the threshold. Conversely, at timestamp t_3, only the new record with the value 'EEE' is inserted into the collection, while the old records remain unchanged due to the prolonged time interval between t_2 and t_3.
- **Time Segment Strategy** Figure 2b demonstrates the strategy employing time segments. Let us consider a scenario where there exists a record with the value 'PPP' in the collection. At timestamp t_0, a write request is sent to the collection module. Given that the request timestamp falls within the time segment s_1 and no record has been inserted within s_1, the record with the value 'QQQ' is inserted into the collection. However, within time segment s_2, two distinct write requests are received. Only the latest one will be retained. Specifically, at timestamp t_1, the record with the value 'RRR' will be inserted into the collection. Therefore, at timestamp t_2, it will be replaced by the record with the value 'SSS'.

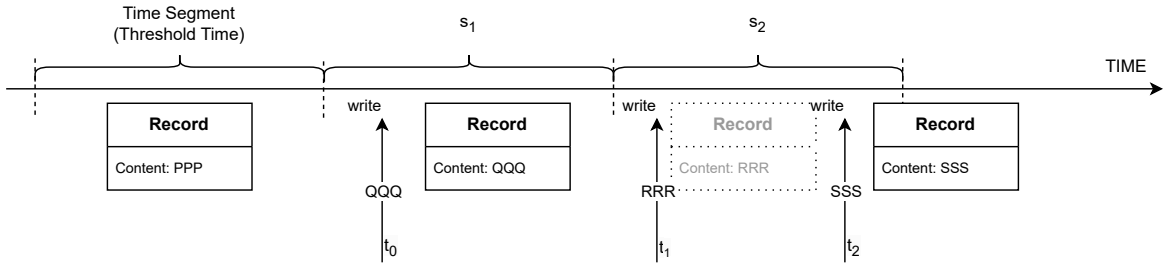
The choice between the two strategies involves a trade-off. A time segment strategy ensures a higher probability of retaining data for each interval but may store more redundant information for long-lasting and repetitive write operations. Conversely, the interarrival time strategy retains only a few data points during rapid and extensive write operations. However, it may result in significantly fewer records when dealing with a repetitive data stream.

Due to the protection mechanisms in place, attackers face significant challenges in filling up the collection to its threshold size and triggering the auto-commit mechanism. Their only exploitable avenue lies in manipulating existing content blocks. However, the rate at which the collection size grows is controlled by the threshold time implemented within the protection mechanism. For each sector, attackers are limited to inserting only one record within a time segment or interarrival time threshold. Additionally, their ability to write values to sectors is severely restricted; they cannot write values to any sectors in the guest OS.

The duration of the threshold time can vary significantly, ranging from minutes to several hours, depending on the users' desired granularity of restoration. A prolonged threshold period causes the collection's size to grow slowly, with records merging for each sector, thus making exploitation more difficult for attackers. Conversely, a shorter threshold time results in the collection containing more detailed information, thereby enhancing the degree of restoration achievable.



(a) The strategy of utilizing interarrival time.



(b) The strategy of utilizing time segments.

Figure 2: Schematic diagram of the two strategies to streamline the records of existing content blocks.

3.4.4 User Notification. To address the inability to determine whether data is encrypted, Time Machine employs a user notification mechanism within the collection manager module. File encryption operations often involve writing to different sectors in a short period. This mechanism alerts users when there is a significant increase in the collection size over a short period, indicating potential encryption activity. The notification mechanism operates on the premise that if numerous write requests target several used sectors within a brief period, it suggests potential file encryption within the guest operating system. This pattern resembles processes that alter numerous files or substantially modify the content of large files. The mechanism notifies users upon detecting such behavior. Additionally, it provides information on which sectors are currently undergoing writing. While the warning mechanism effectively signals potential threats, it may yield false positives. For instance, system updates or data compression activities could trigger alerts. However, such occurrences are infrequent during normal operations or can be anticipated in advance.

It is important to note that our entire framework operates independently of the detection mechanism. The notification mechanism described earlier can be seamlessly substituted with advanced detection algorithms, offering enhanced protection against ransomware at the hypervisor level. For instance, employing state-of-the-art approaches [9] allows for reliable differentiation between compressed and encrypted data fragments. Additionally, leveraging virtual machine introspection (VMI) techniques enables the acquisition of necessary contextual information from the guest operating system at the hypervisor level, facilitating the implementation of ransomware detection algorithms [38]. By adopting such strategies, our framework provides a more comprehensive defense against ransomware attacks, enhancing security for virtualized environments.

3.5 Timestamp-based State Retrieval and Recovery

In contrast to other related works, Time Machine offers the capability to restore the system state to any specific version based on a timestamp. Leveraging the information stored in the key-value store collection, it becomes feasible to revert the system to its state before an attack occurs. A critical aspect to consider is that users may prefer to examine the filesystem initially to determine the timestamp at which files within the guest operating system were not yet encrypted. This preliminary check aids users in making an informed decision regarding the optimal timestamp for system restoration.

To enhance user-friendliness, Time Machine offers a READ mode, enabling users to inspect the filesystem at any specific timestamp without affecting the disk or the collection module. In READ mode, all I/O requests issued by the guest's storage driver are intercepted. Subsequently, the I/O interception module rejects write requests to prevent any alteration of the collection module or disk contents. Following the standard flow depicted in Figure 1, read requests in READ mode are partitioned into sectors and directed to either the collection module or the disk driver. The collection module, operating in READ mode, retrieves the corresponding values at the specified timestamp by seeking the latest value preceding the timestamp. By employing READ mode, users can explore the filesystem at any timestamp, similar to using the system normally but with read-only characteristics and access to previous states.

Once users become aware that the system has been compromised by ransomware or other malware or receive a notification to update the disk due to the collection size surpassing the notification threshold, they can utilize Time Machine's RESTORE mode. This mode enables them to update the disk to a specified timestamp. In RESTORE mode, Time Machine first searches for the latest value of each sector with data preceding the given timestamp in the key-value store collection. It then updates these values to the disk based on the recorded sectors. Using RESTORE mode to update the disk status triggers an initialization process, which involves clearing all records in the collection module and removing the bitmaps in both the lookup and collection manager modules. To utilize RESTORE mode effectively, users can employ the READ mode to examine the previous state of the filesystem and identify the timestamp at which the disk was in the desired state. Subsequently, they can use RESTORE mode to revert the disk to that specific timestamp.

4 IMPLEMENTATION

To validate our approach, we developed a proof-of-concept prototype of Time Machine. This prototype was implemented by modifying the open-source QEMU hypervisor [7], specifically version 7.2.91, running on an Ubuntu 20.04/x86-64 system. The guest operating systems utilized in our prototype are Ubuntu 18.04/x86-64 and 64-bit Microsoft Windows 10. QEMU, being a type II hypervisor, typically resorts to software emulation when native hardware support is unavailable. However, traditional software-based virtualization methods, such as full virtualization or emulation, often incur substantial CPU overhead due to the need for instruction translation and emulation, leading to significant performance penalties.

We leverage QEMU's hardware virtualization support to mitigate this issue and enable a Kernel-based Virtual Machine (KVM) [18]. By enabling KVM, guest operating systems can execute code directly on the CPU with minimal overhead. This approach achieves near-native performance levels, making KVM particularly suitable for performance-sensitive workloads. The prototype introduces approximately 2800 lines of additional C code to QEMU.

In our prototype, we utilize MongoDB as the key-value collection. To ensure compatibility with MongoDB, we selected Remote Dictionary Server (Redis) as the cache system. Moreover, we employ the Least Recently Used (LRU) algorithm to manage key replacements in Redis. Due to space limitations, detailed introductions of the involved components have been moved to Appendix A.

5 EVALUATION

5.1 Experimental Setup

Recently, Windows and Linux have emerged as the dominant operating systems worldwide. Windows, in particular, holds a strong presence in the desktop market, making it a primary target for attackers. Consequently, there has been a significant increase in ransomware incidents targeting Windows systems. On the other hand, Linux has traditionally been the preferred operating system for web servers, including those hosted on public cloud platforms. As a result, ransomware targeting Linux systems is on the rise.

In our experiments, ransomware samples were executed on Ubuntu 18.04/x86-64 and 64-bit Microsoft Windows 10 systems, both with 2 GB RAM. The host machine, running on VirtualBox, is an Ubuntu 20.04/x86-64 system equipped with 12 GB RAM and 8 processors. To enable the KVM accelerator in QEMU, the Nested VT-x/AMD-V option has been turned on. To further simulate the performance of Time Machine within an internal network environment, we migrated the collection to a different system, as detailed in Section 5.5. This system consists of an Ubuntu 20.04/x86-64 setup, equipped with 12 GB of RAM and a single processor, serving as the backend storage server.

The primary reason for employing a two-layer VM architecture is for greater flexibility and convenience in performing experiments. However, this approach may also lead to lower guest I/O throughput. The threshold time used in the protection strategy for used blocks, including interarrival time and time segment, was set to 5 minutes.

Based on the characteristics of ransomware, we assume that all ransomware samples either encrypt user files or lock the user's system. To simulate real-world scenarios, we created a user document directory containing various file types, such as documents, media files, program source files, and archive files. Additionally, to ensure the successful execution of ransomware samples, we disabled the antivirus software, firewalls, and any other processes that might obstruct malicious operations within the guest machine. Apart from these adjustments, we did not alter any guest OS settings.

Since certain ransomware variants require network connectivity to transmit encryption keys or potentially steal data, they may not execute successfully without network access. To address this, we granted the guest machine unrestricted network access through Network Address Translation (NAT). Each ransomware sample was executed with administrator privileges to ensure its ability to carry out attacks on user files. Prior to running each sample, we reverted both the guest and host machines using a snapshot to ensure that the samples were not influenced by previous executions.

We systematically verify the capability of our prototype to recover the guest's filesystem to any given timestamp through the following procedure. Firstly, we power on the guest machine and perform random operations (OP) on files within the user document directory, including file creation, writing, and content appending. Next, we execute the ransomware sample for a duration of 30 minutes. Subsequently, we examine whether the ransomware has encrypted any files in the directory. If no encryption is detected, we proceed to the next step. Otherwise, we proceed to the restoration process. If no files are encrypted, we assess whether the filesystem can be restored to its state prior to the OP operations performed on the files. We utilize the RESTORE mode of the prototype for this assessment. If encryption is detected, we browse the filesystem at two timestamps: one preceding the encryption but after the OP operations, and another prior to the OP operations. We utilize the READ mode of the prototype for this browsing process. Following the browsing process, we employ the RESTORE mode to revert the filesystem to its state before encryption but after the operations were conducted. Finally, we assess whether the filesystem has reverted successfully after restoration.

5.2 Effectiveness of Recovery

We gathered real-world Linux and Windows ransomware samples from MalwareBazaar [24] and the Github [28] to assess the recovery

effectiveness. Subsequently, each ransomware sample was executed on its corresponding operating system for 30 minutes.

To confirm Time Machine's capability to browse the filesystem at any timestamp, we utilized the READ mode of Time Machine to inspect the filesystem state at timestamps preceding the encryption but after the operations and the timestamp before the operations, as discussed in Section 5. Success was determined by examining the files within the user document directory and ensuring they matched their expected state.

To validate Time Machine's ability to recover to any given timestamp, we utilized the RESTORE mode to revert the disk to the timestamp preceding the encryption but after the operations. Subsequently, we compared the files in the user document directory with their original versions (after random operations), ensuring they remained unchanged. If the files were identical, we concluded that Time Machine could recover from the ransomware samples.

Table 1 presents the results of the browsing and recovery tests, where "Browsing" denotes the capability to navigate the filesystem at any given timestamp, and "Recovery" indicates the ability to restore the disk to a specified timestamp. Throughout the experiment, we conducted tests on 11 Linux[1] and 12 Windows[2] ransomware samples. However, only 6 Linux and 6 Windows ransomware samples exhibited encryption behavior. Results for ransomware samples without encryption behavior were excluded from Table 1. Nevertheless, we conducted recovery and browsing tests for all samples, even if the ransomware samples were invalid. In such cases, we restored the disk to its state before any operations occurred. In summary, Time Machine successfully enables browsing the filesystem at any timestamp and facilitates fine-grained recovery of the entire disk to a specified timestamp for all tested ransomware samples.

We further investigated the growth behavior of the collection size after ransomware encryption through case studies. Leveraging the characteristics of specific ransomware that alter file extensions after encryption, we calculated the total number of files encrypted in the file system and retrieved their original filenames. We compared the number of files encrypted, the total size of these files, and the resulting increase in the collection size. After executing Buhti, approximately 219 files were encrypted, amounting to roughly 72.91 MB of data. The execution resulted in a collection size increase of about 57.66MB. Similarly, for Linux's Cylance, approximately 242 files were encrypted, amounting to roughly 78.25 MB of data. The execution led to an increase in the collection size of around 76.35MB.

5.3 Effectiveness of Handling Collection Size

As outlined in Section 3.4, Time Machine implements several protection mechanisms to thwart targeted attacks on the collection size. To evaluate the efficacy of these protective measures, we conduct three types of attack experiments and analyze their respective effects on the collection size. Note that the baseline in the experiments indicates that Time Machine is configured without employing unused block bitmap, interarrival time, and time segment strategies.

[1]Linux Samples: Erebus, Kuiper, RansomEXX, Hive, Qilin, Lockbit, GonnaCry, AvosLocker, Buhti, Cylance, IceFire.
[2]Windows Samples: Phobos, WastedLocker, WannaCry, Thanos, TeslaCrypt, Satana, RedBoot, Radamant, Petya, Mamba, Locky, Cylance.

Table 1: List of ransomware samples that perform encryption.

Linux Samples		
Ransomware	Recovery	Browsing
Erebus	✓	✓
Kuiper	✓	✓
AvosLocker	✓	✓
Buhti	✓	✓
Cylance	✓	✓
IceFire	✓	✓
Windows Samples		
Ransomware	Recovery	Browsing
Phobos	✓	✓
WastedLocker	✓	✓
WannaCry	✓	✓
RedBoot	✓	✓
Thanos	✓	✓
Cylance	✓	✓

5.3.1 Altering Existing Files. In the first scenario, we envisage attackers attempting to overflow the collection by consistently altering existing files within the guest environment. Here, we initiate the guest, execute a program to iteratively write to a file (50K, 100K, 200K, and 300K times), power off the guest, and analyze the resulting increase in the collection size. The outcomes of this initial experiment are depicted in Figure 3. When a file is modified within the guest, it typically generates write requests aimed at utilized blocks on the disk. Consequently, the protective mechanism addressing unused blocks has minimal impact on reducing the collection size. Conversely, the protective mechanism targeting used blocks demonstrates a notable reduction in the collection size during the experiment.

Figure 3 further evaluates the effectiveness of the two protection strategies for used blocks. Both strategies display efficient management of used blocks, resulting in similar increases in the collection size irrespective of the number of modifications performed. From the standpoint of minimizing the collection size, the interarrival time strategy proves advantageous if the duration between each write request lengthens but remains within the merge threshold. However, this approach compromises restoration precision. If users continually modify a file while ransomware encrypts it, only the latest version will be recorded in the collection. Nevertheless, files are typically locked during modification, preventing ransomware from directly encrypting them. Therefore, the drawback of the interarrival time method is tolerable in such scenarios, and the prototype adopts the interarrival time as the protective strategy for used blocks.

It is important to note that to ensure the modification operations are directed to the hypervisor, we utilize the *fsync* system call to compel it to flush data to the hypervisor before each write request. It ensures that the number of modifications equals the number of write requests corresponding to the sector number of the file.

5.3.2 Generating Large Files. In the second scenario, we explore the potential for attackers to fill the collection size by generating

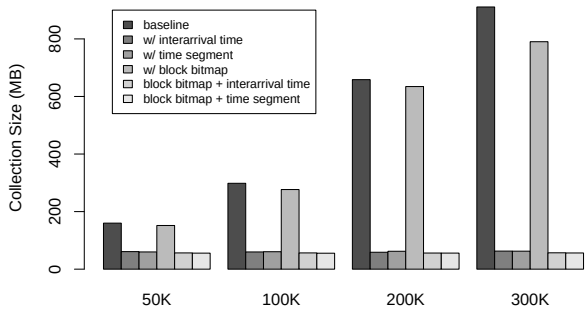


Figure 3: The increased collection size corresponding to writing a file many times in Linux.

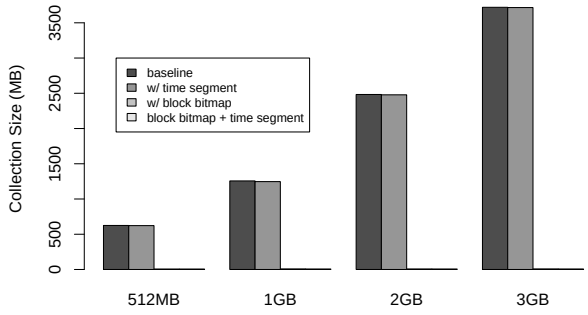


Figure 4: The increased collection size corresponding to creating a large file in Linux.

large files within the guest environment. Here, we power on the guest, create a sizeable file of various sizes (0.5GB, 1GB, 2GB, and 3GB), power off the guest, and examine the resulting increase in the collection size. The outcomes of this experiment are depicted in Figure 4.

Without protective mechanisms, the collection logs all write requests to the disk, resulting in a size increase closely aligned with the size of the created file. Given that file creation within the guest typically triggers write requests targeting unused blocks on the disk, the protective mechanism for used blocks has minimal impact on reducing the collection size. However, the safeguard for unused blocks enables write requests directed toward such blocks to access the disk directly. Consequently, most of the I/O requests generated by file creation within the guest can be written directly to the disk, exerting minimal influence on the collection size.

5.3.3 Generating Numerous Files. In the third scenario, we investigate where attackers seek to fill the collection size by generating

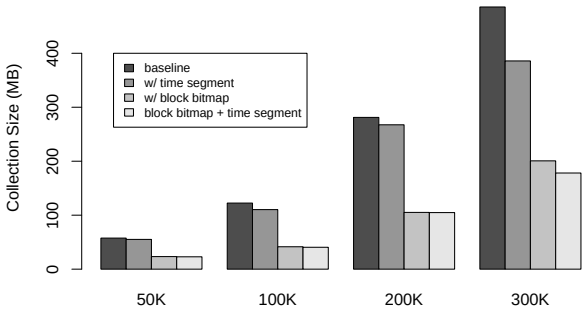


Figure 5: The increased collection size corresponding to creating many files in Linux.

numerous files within the guest environment. Here, we initialize the guest machine, execute a program designed to create varying numbers of files within the guest (50K, 100K, 200K, 300K), power off the guest, and subsequently analyze the resulting increase in the collection size. Each file is populated with 512-byte random values generated using the *rand* function in C. The experimental outcomes employing different protection strategies are illustrated in Figure 5.

Similar to the second scenario, file creation within the guest typically triggers write requests to unused blocks on the disk, resulting in limited effectiveness of the protection mechanism against used blocks. However, as the number of created files increases, the collection size expands regardless of the employed strategies. It is due to file creation within the guest leading to write requests to its respective data blocks and incurring write requests to the inode block of the directory for file creation. In the EXT4 filesystem, file/directory creation necessitates modification of the extent tree structure or the corresponding inode content, thereby contributing to the growth of the collection size. Despite the inevitable increase in collection size, both protection mechanisms for used and unused blocks effectively mitigate the issue.

5.4 Resource Constraints of Cache

In this section, we investigate the impact of the cache system on Time Machine. We begin the experiment by clearing all records in both the key-value store collection and the cache server, eliminating any extraneous factors apart from the maximum memory constraint of the cache system. Subsequently, we power on the guest and utilize IOzone [14] to assess the I/O performance of Time Machine across different memory limits. By configuring various parameters of IOzone, we conduct tests covering six categories of I/O performance:

- Write: Evaluate the throughput of writing a new file.
- Re-Write: Assess the writing throughput to an existing file.
- Read: Measure the throughput of reading an existing file.
- Re-Read: Gauge the throughput of reading a recently accessed file.

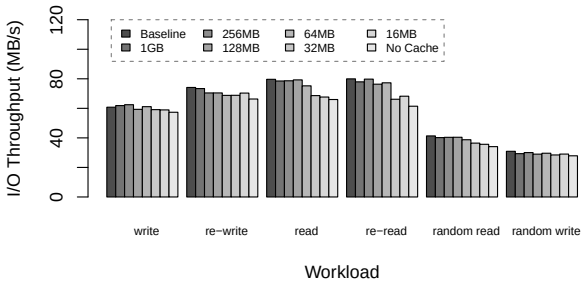


Figure 6: I/O throughput with different cache sizes.

- Random Read: Determine the throughput of reading a file with accesses occurring at random locations within the file.
- Random Write: Assess the throughput of writing a file with accesses occurring at random locations within the file.

Figure 6 shows the results of the experiment. The memory limit ranges from 1GB to 16MB. Across different memory limits, the categories related to write operations exhibit similar throughput compared to the guest in QEMU without Time Machine. Write operations typically involve write requests to unused blocks, which directly access the disk in our mechanism. For re-write and random write operations, since the initial collection size was reset to zero, there are fewer instances requiring the handling of used blocks. Most write requests are directly logged in the collection, resulting in throughput close to the baseline. Moreover, as write-related operations and the cache mechanism are less correlated, the memory limit's impact on the three writing operations is minimal.

In contrast, read-related operations show a relatively noticeable downward trend compared to write-related operations. There is a more pronounced relationship between cache size reduction and throughput decline for re-read operations, which involve retrieving recently read files. We hypothesize that when the memory size is limited, the data in the cache may be overwritten by the system's I/O operations, rendering the cache less effective for re-reads and decreasing throughput. Without the assistance of the cache system, the I/O throughput of all categories experiences a further decline, with read-related operations being more affected. This decline is attributed to the impact on the system's operational I/O operations, which consequently affect the performance of all I/O operations.

Based on the findings illustrated in Figure 6, Time Machine demonstrates comparable I/O throughput to the baseline when configured with both 1GB and 256MB memory. We used the average overhead across the six categories of I/O operations as the selection criterion. Considering resource conservation, we opt for a maximum memory limit of 256MB.

5.5 Impact on I/O Performance

To evaluate the impact of Time Machine on I/O performance, we conduct benchmarks in two environments: (1) an operating system running on the hypervisor as the baseline, and (2) an operating system running on the hypervisor with Time Machine enabled. We utilize both Linux and Windows operating systems in the experiment. Despite not implementing the protection mechanism for unused blocks in Windows due to filesystem dependencies, we include an evaluation of Windows in this section. This analysis explores the relationship between I/O throughput and the collection size. We employ IOzone to evaluate the I/O throughput in both environments and analyze the associated I/O overhead with various collection sizes.

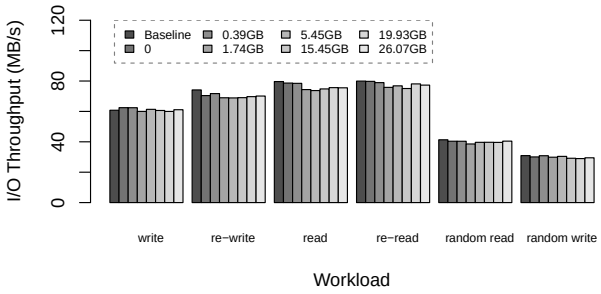
In each round of testing, we record the collection size before booting a guest virtual machine, which is labeled in the legend of the figures. After booting, we measure the I/O throughput and prepare for the next test round. We run a program in the guest that periodically modifies the files to fill the collection size. By varying the number of modifications and the size of modified files, we fill the collection to ensure the size keeps increasing for the sake of performance comparison under different sizes.

5.5.1 Linux OS. The results measured in Linux are depicted in Figure 7. Figure 7a shows the results when Time Machine employs local backend storage for the collection. The bars, from left to right, represent the baseline and the throughputs with increased collection size. As the collection size increases, the I/O throughputs of all testing categories exhibit a slight decrease compared to the baseline. We attribute this non-linear decline in I/O throughput to the characteristics of the key-value store collection. Overall, the increase in collection size does not significantly affect I/O throughput. On average, the overhead is approximately 3.5%. Notably, the write operation demonstrates relatively lower overhead than other testing categories due to the protection mechanism for the unused blocks. For other operations, the overhead ranges from approximately 2% to 7% across all collection sizes compared to the baseline, which remains within acceptable limits.

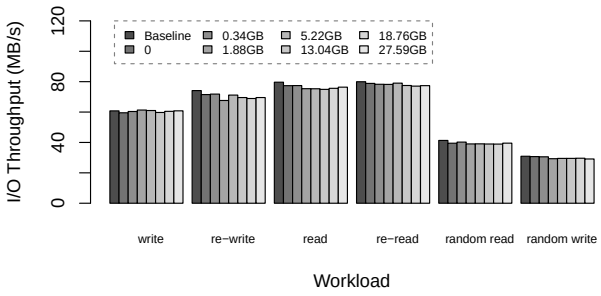
On the other hand, Figure 7b presents the results under the condition where Time Machine interacts with the backend storage collection over the network. The I/O throughput performance is similar to that of the local setup. The results indicate that the I/O overhead compared to the local setup is less than 1% on average, which is negligible.

5.5.2 Windows OS. Figure 8a depicts the results for Windows OS when Time Machine employs local backend storage for the collection. Note that the handling of unused blocks was not activated in this experiment. Similar to the findings in the Linux OS, the I/O throughput slightly decreased for all tested categories compared to the baseline. With the deactivated handling unused blocks feature, all write requests must be processed and preserved in the collection. Always writing data to the disk directly resulted in a decline in the I/O throughput for write operations compared to Linux. For other operations, the trends are similar to those in the Linux OS. In summary, the overhead is approximately 4.5% on average. All tested operations show an overhead of approximately 3% to 9% for various collection sizes compared to the baseline.

Figure 8b presents the results of Time Machine interacting with the backend storage collection over the network. We observe that the I/O throughput is not significantly different from the local setup for all tested categories. The average overhead is less than 1%.

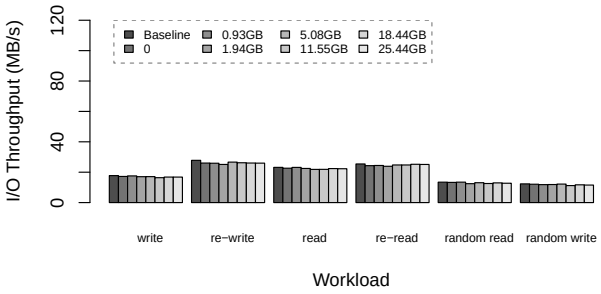


(a) I/O throughput on Linux (local backend setup).

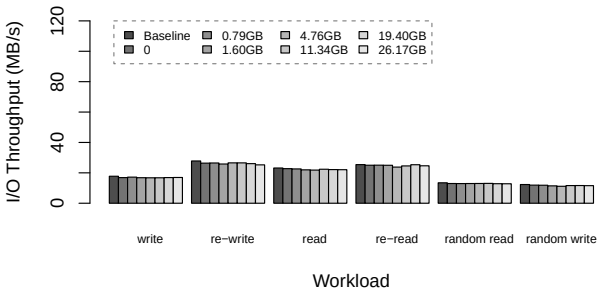


(b) I/O throughput on Linux (networked backend setup).

Figure 7: I/O throughput measured on Linux.



(a) I/O throughput on Windows (local backend setup).



(b) I/O throughput on Windows(networked backend setup).

Figure 8: I/O throughput measured on Windows.

5.5.3 Restoration Time. The restoration time for Linux and Windows with various collection sizes is shown in Figure 9. It is evident that the time required for Time Machine's RESTORE mode to revert the disk is closely proportional to the collection size. For instance, when the collection size reaches approximately 25GB, the restore time for Windows and Linux OSes is about 101 and 104 seconds, respectively.

5.6 Attack Scenario

This section introduces the potential attacks targeting Time Machine and discusses the corresponding countermeasures.

5.6.1 Attacks Targeted on the Collection Module. Time Machine stores its backend data in the collection module, thus eliminating the hardware dependency associated with SSD-based approaches by leveraging software solutions. Although Time Machine benefits from the expansive storage capacity of cloud storage, its storage space is still finite. Consequently, Time Machine cannot indefinitely record all I/O activities. If attackers understand the mechanism of Time Machine, they might intentionally execute numerous redundant I/O operations. Such actions could increase the collection size, reduce the I/O efficiency of the guest OS, and potentially cause overflow in the collection module. It, in turn, could result in malicious operations being written to the disk.

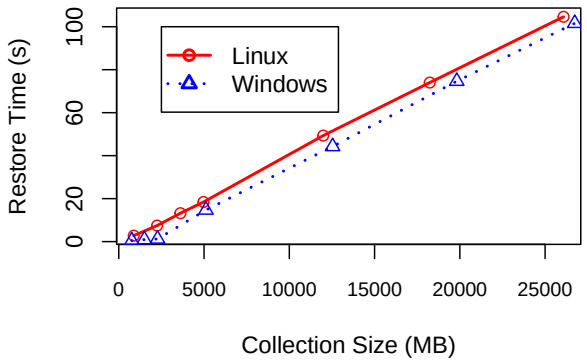


Figure 9: Restoration Time.

As mentioned in section 3.4, Time Machine utilizes a protection mechanism in the collection manager module to mitigate such conditions. Increasing the collection size significantly is not easy

for attackers. Even if they manage to do so, Time Machine issues a warning to users if the collection size grows rapidly within a short period. Therefore, attackers would need to fill the collection size gradually to avoid detection. However, this process would take a long time, during which users might proactively update data from the collection to the disk, necessitating a refill. Moreover, if the collection is filled slowly, the impact on I/O performance would likely be minimal or negligible.

5.6.2 *Attacks Targeted on the Collection Manager Module.* .

Time Machine relies on an auto- or manually-commit mechanism to flush data to the disk. The effectiveness of the user notification mechanism depends on how rapidly the collection is filled. Given the characteristics of Time Machine, attackers could gradually encrypt a few files over an extended period. In such cases, the user notification mechanism may not promptly detect the malicious activity. Users might also fail to notice abnormal file encryption and continue using the computer until the collection size reaches its limit.

Encrypting a few files gradually over an extended period means that users and other ransomware detection tools have more time to discover the attack in its early stages. It allows users to take countermeasures and thwart the attack before encrypting important files. Additionally, requiring users to inspect the filesystem status once the collection size exceeds a notification threshold can help prevent malicious payloads from being updated to the disk. Furthermore, the attacker portals used for receiving ransom payments are usually unstable. These portals and payment methods may be outdated after the encryption is completed. Therefore, from the ransomware's perspective, extending the encryption period to counter Time Machine offers no real benefit.

Our proposed approach, with Time Machine residing in the hypervisor, offers a unique defense against common privilege escalation attacks within the operating system. These attacks, which often seek to bypass defenses by acquiring administrator privileges, are inherently thwarted by our system's architecture. Our approach operates independently of the detection algorithm, allowing it to incorporate state-of-the-art methods to enhance protection against ransomware.

5.7 Limitation

The limitation of Time Machine lies in the potential data loss caused by the auto-commit mechanism and I/O overhead. However, the primary contribution of this work is not the development of a robust ransomware detection algorithm at the hypervisor level. Instead, our work introduces a framework that employs a sector-level live view navigation approach at the hypervisor level to protect user files within the guest operating system comprehensively. This framework operates independently of detection mechanisms and can integrate with existing detection approaches to mitigate the risk of data loss attributed to the auto-commit mechanism.

Despite this limitation, a detection mechanism can also help improve storage efficiency. If a potent detection mechanism achieves 100% accuracy in detecting ransomware, there would be no need to continuously record I/O requests for extended periods to prevent potential data loss. The collection size can be minimized by setting a smaller upper bound threshold.

Time Machine is an architecture-agnostic system that operates independently of the guest operating system and disk, eliminating hardware dependencies by offloading storage tasks to a software-based storage server. Although this approach incurs slightly higher I/O overhead compared to SSD-based solutions, the overhead remains within an acceptable range. Further optimizations to the storage server/backend efficiency could help mitigate this overhead. Future work includes integrating detection algorithms and exploring more efficient storage algorithms.

6 CONCLUSION

We proposed Time Machine, a real-time fine-grained sector-level live view navigation approach designed to protect filesystems against ransomware attacks at the hypervisor level. Time Machine require no hardware dependency and instead leverages the capability of key-value store collection to record the I/O requests. Additionally, it incorporates two optimization strategies to improve the performance of the proposed approach. With all the I/O requests being recorded, Time Machine provides complete protection and enables users to easily navigate the filesystem at any given timestamp and recover the filesystem content to the selected timestamp. The evaluation results demonstrate that Time Machine can effectively secure user data from ransomware attacks while maintaining minimal overhead (<5%). We believe the proposed approach pinpoints an innovative direction for defending against ransomware attacks.

REFERENCES

[1] Muhammad Shabbir Abbasi, Harith Al-Sahaf, and Ian Welch. 2020. Particle Swarm Optimization: A Wrapper-Based Feature Selection Method for Ransomware Detection and Classification. In *Applications of Evolutionary Computation: 23rd European Conference, EvoApplications 2020, Held as Part of EvoStar 2020, Seville, Spain, April 15–17, 2020, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 181–196. https://doi.org/10.1007/978-3-030-43722-0_12

[2] Yahye Abukar Ahmed, Barış Koçer, Shamsul Huda, Bander Ali Saleh Al-rimy, and Mohammad Mehedi Hassan. 2020. A system call refinement-based enhanced Minimum Redundancy Maximum Relevance method for ransomware early detection. *Journal of Network and Computer Applications* 167 (2020), 102753. https://doi.org/10.1016/j.jnca.2020.102753

[3] Bander Al-rimy, Mohd Maarof, Yuli Prasetyo, Syed Zainudeen Mohd Shaid, and Aswami Ariffin. 2018. Zero-Day Aware Decision Fusion-Based Model for Crypto-Ransomware Early Detection. *International Journal of Integrated Engineering* 10 (11 2018). https://doi.org/10.30880/ijie.2018.10.06.011

[4] SungHa Baek, Youngdon Jung, Aziz Mohaisen, Sungjin Lee, and DaeHun Nyang. 2018. 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)*. 875–884. https://doi.org/10.1109/ICDCS.2018.00089

[5] Sungha Baek, Youngdon Jung, David Mohaisen, Sungjin Lee, and DaeHun Nyang. 2021. SSD-Assisted Ransomware Detection and Data Recovery Techniques. *IEEE Trans. Comput.* 70, 10 (2021), 1762–1776. https://doi.org/10.1109/TC.2020.3011214

[6] James Baldwin and Ali Dehghantanha. 2018. *Leveraging Support Vector Machine for Opcode Density Based Detection of Crypto-Ransomware*. Springer International Publishing, Cham, 107–136. https://doi.org/10.1007/978-3-319-73951-9_6

[7] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[8] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. 2016. ShieldFS: A Self-healing, Ransomware-aware Filesystem. In *Proceedings of the 32nd Annual Computer Security Applications Conference* (Los Angeles, USA, 2016-12). ACM.

[9] Fabio De Gaspari, Dorjan Hitaj, Giulio Pagnotta, Lorenzo De Carli, and Luigi V Mancini. 2020. Encod: Distinguishing compressed and encrypted file fragments. In *Network and System Security: 14th International Conference, NSS 2020, Melbourne, VIC, Australia, November 25–27, 2020, Proceedings 14*. Springer, 42–62.

[10] Europol. 2016. No More Ransom. (2016). https://www.europol.europa.eu/media-press/newsroom/news/no-more-ransom-law-enforcement-and-it-security-companies-join-forces-to-fight-ransomware

[11] Danny Yuxing Huang, Maxwell Matthaios Aliapoulios, Vector Guo Li, Luca Invernizzi, Elie Bursztein, Kylie McRoberts, Jonathan Levin, Kirill Levchenko, Alex C. Snoeren, and Damon McCoy. 2018. Tracking Ransomware End-to-end. In *Proceedings - 2018 IEEE Symposium on Security and Privacy, SP 2018 (Proceedings - IEEE Symposium on Security and Privacy)*. Institute of Electrical and Electronics Engineers Inc., 618–631. https://doi.org/10.1109/SP.2018.00047

[12] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K. Qureshi. 2017. FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2231–2244. https://doi.org/10.1145/3133956.3134035

[13] Gizmodo International. 2017. Today's Massive Ransomware Attack Was Mostly Preventable; Here's How To Avoid It. *Gizmodo Australia* (05 2017). https://gizmodo.com.au/2017/05/todays-massive-ransomware-attack-was-mostly-preventable-heres-how-to-avoid-it/

[14] IOzone . 2018. IOzone Filesystem Benchmark. online. Last accessed: 2024.04.30, http://www.iozone.org/.

[15] Brijesh Jethva, Issa Traoré, Asem Ghaleb, Karim Ganame, and Sherif Ahmed. 2020. Multilayer ransomware detection using grouped registry key operations, file entropy and file signature monitoring. *J. Comput. Secur.* 28, 3 (jan 2020), 337–373. https://doi.org/10.3233/JCS-191346

[16] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. 2016. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware.

[17] Amin Kharraz and Engin Kirda. 2017. Redemption: Real-Time Protection Against Ransomware at End-Hosts. In *Research in Attacks, Intrusions, and Defenses*, Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.). Springer International Publishing, Cham, 98–119.

[18] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, Vol. 1. Dttawa, Dntorio, Canada, 225–230.

[19] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. 2017. PayBreak: Defense Against Cryptographic Ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 599–611. https://doi.org/10.1145/3052973.3053035

[20] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. 2006. The Linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.* 40, 3 (jul 2006), 102–107. https://doi.org/10.1145/1151374.1151375

[21] Kyungroul Lee, Jaehyuk Lee, Sun-Young Lee, and Kangbin Yim. 2023. Effective Ransomware Detection Using Entropy Estimation of Files for Cloud Services. *Sensors* 23, 6 (2023). https://doi.org/10.3390/s23063023

[22] SOPHOS Ltd. 2023. *The State of Ransomware 2023.* Technical Report. SOPHOS. https://www.congress.gov/118/meeting/house/116406/documents/HHRG-118-GO12-20230927-SD003.pdf

[23] Boyang Ma, Yilin Yang, Jinku Li, Fengwei Zhang, Wenbo Shen, Yajin Zhou, and Jianfeng Ma. 2023. Travelling the Hypervisor and SSD: A Tag-Based Approach Against Crypto Ransomware with Fine-Grained Data Recovery. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 341–355. https://doi.org/10.1145/3576915.3616665

[24] MalwareBazaar. [n.d.]. MalwareBazaar Database. online. Last accessed: 2024.04.29, https://bazaar.abuse.ch/browse/.

[25] May Medhat, Samir Gaber, and Nashwa Abdelbaki. 2018. A New Static-Based Framework for Ransomware Detection. 710–715. https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00124

[26] Shagufta Mehnaz, Anand Mudgerikar, and Elisa Bertino. 2018. *RWGuard: A Real-Time Detection System Against Cryptographic Ransomware: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings.* 114–136. https://doi.org/10.1007/978-3-030-00470-5_6

[27] Donghyun Min, Donggyu Park, Jinwoo Ahn, Ryan Walker, Junghee Lee, Sungyong Park, and Youngjae Kim. 2018. Amoeba: An Autonomous Backup and Recovery SSD for Ransomware Attack Defense. *IEEE Computer Architecture Letters* 17, 2 (2018), 245–248. https://doi.org/10.1109/LCA.2018.2883431

[28] Mohsen KHashei. 2021. Ransomware-Samples. online. Last accessed: 2024.06.07, https://github.com/kh4sh3i/Ransomware-Samples.

[29] Pradeep Padala. 2005. A Log Structured File System with Snapshots. (08 2005).

[30] Joon-Young Paik, GeunYong Kim, Seoyeon Kang, Rize Jin, and Eun-Sun Cho. 2022. Data Protection Based on Hidden Space in Windows Against Ransomware. In *Proceedings of Sixth International Congress on Information and Communication Technology*, Xin-She Yang, Simon Sherratt, Nilanjan Dey, and Amit Joshi (Eds.). Springer Singapore, Singapore, 629–637.

[31] Jisung Park, Youngdon Jung, Jonghoon Won, Minji Kang, Sungjin Lee, and Jihong Kim. 2019. RansomBlocker: a Low-Overhead Ransomware-Proof SSD. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[32] Subash Poudyal, Kul Prasad Subedi, and Dipankar Dasgupta. 2018. A Framework for Analyzing Ransomware using Machine Learning. *2018 IEEE Symposium Series on Computational Intelligence (SSCI)* (2018), 1692–1699. https://api.semanticscholar.org/CorpusID:59554080

[33] J.T. Robinson and P.A. Franaszek. 1994. Analysis of reorganization overhead in log-structured file systems. In *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*. 102–110. https://doi.org/10.1109/ICDE.1994.283000

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

[35] Scott Sayce. 2024. 3 trends set to drive cyberattacks and ransomware in 2024. *World Economic Forum Annual Meeting* (2024). https://www.weforum.org/agenda/2024/02/3-trends-ransomware-2024/

[36] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin R. B. Butler. 2016. CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 303–312. https://doi.org/10.1109/ICDCS.2016.46

[37] Shaila Sharmeen, Yahye Abukar, Shamsul Huda, Baris Kocer, and Mohammad Hassan. 2020. Avoiding Future Digital Extortion Through Robust Protection Against Ransomware Threats Using Deep Learning Based Adaptive Approaches. *IEEE Access* PP (01 2020), 1–1. https://doi.org/10.1109/ACCESS.2020.2970466

[38] Fei Tang, Boyang Ma, Jinku Li, Fengwei Zhang, Jipeng Su, and Jianfeng Ma. 2020. RansomSpector: An introspection-based approach to detect crypto ransomware. *Computers & Security* 97 (2020), 101997.

[39] Wentao Xiao, Bin Zhang, Xi Xiao, Arun Kumar, Weizhe Zhang, and Jiajia Zhang. 2019. Ransomware classification using patch-based CNN and self-attention network on embedded N-grams of opcodes. *Future Generation Computer Systems* 110 (09 2019). https://doi.org/10.1016/j.future.2019.09.025

[40] Joobeom YUN, Junbeom HUR, Youngjoo Shin, and Dongyoung Koo. 2017. CLD-Safe: An Efficient File Backup System in Cloud Storage against Ransomware. *IEICE Transactions on Information and Systems* E100.D (09 2017), 2228–2231. https://doi.org/10.1587/transinf.2017EDL8052

[41] Hanqi Zhang, Xi Xiao, Francesco Mercaldo, Shiguang Ni, Fabio Martinelli, and Arun Kumar Sangaiah. 2019. Classification of ransomware families with machine learning based on N-gram of opcodes. *Future Generation Computer Systems* 90 (2019), 211–221. https://doi.org/10.1016/j.future.2018.07.052

A IMPLEMENTATION DETAILS

A.1 Monitor Guest IO

The key component of the prototype is the I/O interception module within the hypervisor. To accomplish this, interception of I/O requests before their actual access to the disk in QEMU is necessary. While KVM benefits from virtualization support provided by physical hardware for CPU, memory, and other functions to improve efficiency, it lacks built-in I/O device support. Nevertheless, I/O devices that are emulated by QEMU through pure software emulation result in longer paths for each I/O operation and decreased performance when communicating between virtual machines and the host machine. Thus, the introduction of VirtIO addresses this issue. VirtIO serves as a universal virtual I/O device protocol, defining two primary aspects: device configuration and initialization on the control plane, and data transfer on the data plane. It allows hypervisors to export a standardized set of simulated devices and provides access to virtual machines through an Application Programming Interface (API). The VirtIO frontend driver resides within the guest machine as the guest I/O driver, while the VirtIO backend driver is implemented in QEMU as the backend I/O driver. Virtual queues (virtqueue), typically in the form of ring buffers (virtio-ring), facilitate communication between the guest and the virtual machine manager. These queues can be implemented in various ways as long as coherence between the guest and the hypervisor is maintained. Virtual queues serve as storage for information exchanged between the guest I/O driver and the hypervisor I/O driver. They can buffer multiple I/O requests from the guest I/O driver simultaneously, which are then processed in batches by the hypervisor I/O driver.

Subsequently, the actual I/O operations are executed physically by invoking the device driver on the host machine. This approach enables batch processing based on agreements rather than handling each I/O request from the guest machine individually. As a result, it improves the efficiency of information exchange between the guest machine and the hypervisor.

The Simplified Function Call Graph (CFG) of the I/O mechanism with VirtIO in QEMU is illustrated in Figure 10. Typically, in the guest machine, I/O requests are pushed to the virtual queue, and it then notifies the hypervisor to handle the I/O requests. The start point of the I/O request to the VirtIO block device in QEMU is the *virtio_blk_handle_output* function. To receive I/O requests from the guest machine, it calls the *virtio_blk_get_request* function to pop the I/O request from the virtual queue and encapsulates the request into a data structure called *VirtIOBlockReq*. The *virtio_blk_handle_request* function consolidates I/O requests into a *MultiReqBuffer* data structure. When the *MultiReqBuffer* reaches its capacity or when traversal of the virtual queue (available ring) is complete, it invokes the *virtio_blk_submit_multireq* function to submit I/O requests to the VirtIO backend driver. Depending on whether the request is for read or write, the *submit_requests* function calls the corresponding function to handle the requests. For read requests, it ultimately invokes the *blk_co_do_preadv_part* function to issue the read request to the block driver by using the *bdrv_co_preadv_part* function for actual disk access. Conversely, for write requests, the *blk_co_do_pwritev_part* function is invoked to initiate the write request to the block driver by using the *bdrv_co_do_pwritev_part* function.

During these processes, the content of the I/O requests is stored in the *QEMUIOVector* data structure. In practice, the Time Machine prototype intercepts the I/O request within the *blk_co_do_pwritev_part* and *blk_co_do_preadv_part* functions. It redirects each read/write request to achieve the expected functionality and protect the guest machine from ransomware attacks.

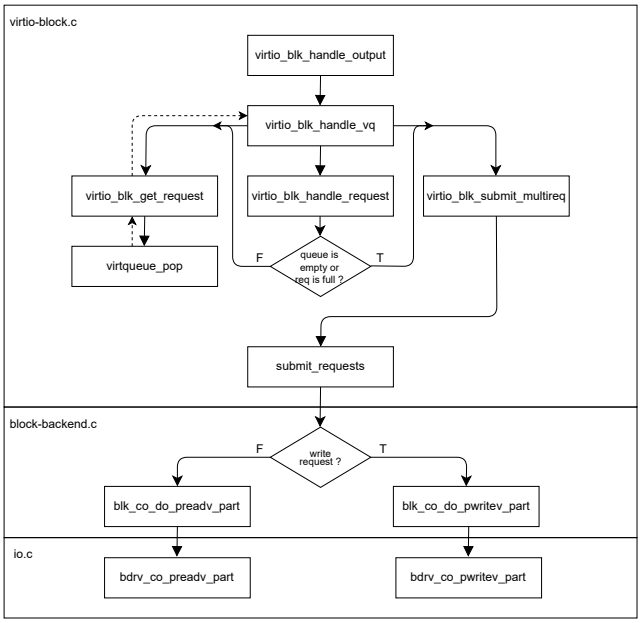


Figure 10: Simple Function Call Graph for VirtIO in QEMU.

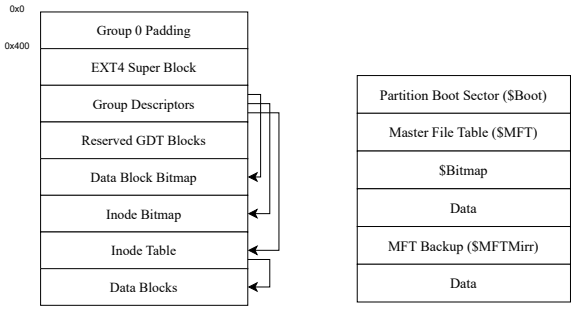
A.2 Collection Module

As mentioned in Section 3, Time Machine employs a key-value store collection to archive information regarding redirected requests managed by the I/O interception module. This collection dynamically records changes in disk status over time and retrieves values to respond to read requests. It stores three variables: target sector, request timestamp, and the corresponding value. In our prototype, we utilize MongoDB as the key-value collection. MongoDB was chosen due to its extensive and active community, which offers abundant resources such as documentation, tutorials, and community forums. Furthermore, MongoDB boasts a robust ecosystem of tools and libraries that seamlessly integrate with various programming languages and frameworks, simplifying development and maintenance efforts. Its scalability allows for the handling of databases with terabyte-level data, and it is compatible with cloud platforms like Amazon Web Services (AWS) and Google Cloud Platform (GCP). In MongoDB, a record is represented as a document, a structured data format comprising field-value pairs. Each document in the prototype's MongoDB consists of an unsigned 64-bit variable representing the sector, another unsigned 64-bit variable representing the timestamp, and a 512-byte variable representing the value.

To ensure compatibility with MongoDB, we selected Remote Dictionary Server (Redis) as the cache system in our prototype. The data structure stored in Redis's in-memory storage mirrors that of MongoDB. We designate the sector as the key in Redis, allowing it to store only the latest value for each sector. To mitigate resource utilization, we impose memory usage restrictions on the Redis server. Following an evaluation of the relationship between the I/O overhead of the guest operating system and memory limits, as detailed in section 5.4, we cap the maximum memory Redis can utilize at 256MB. This limitation has little impact on the I/O overhead of the guest operating system. Moreover, we employ the Least Recently Used (LRU) algorithm to manage key replacements in Redis. This means replacing the least-recent used keys with newer ones, ensuring optimal memory utilization. In essence, keys that have been least recently accessed compared to others are prioritized for replacement. The prototype communicates with both the MongoDB server and the Redis server to record and retrieve disk information.

A.3 Usage of Blocks in Filesystem

As mentioned in section 3.4, Time Machine introduces the concept of the block bitmap within the EXT4 filesystem to track which blocks are not in use, enabling direct data writing to these blocks without any adverse effects. Basically, most common file systems incorporate such functionality. Unix operating systems typically utilize the EXT4 filesystem, with older versions employing EXT2/EXT3. On the other hand, Windows operating systems commonly employ file systems such as FAT, NTFS, or exFAT. This subsection is dedicated to acquiring block usage information for NTFS and EXT4, which are the primary file systems utilized in Windows and Linux,



(a) EXT4 filesystem layout. **(b) NTFS filesystem layout.**

Figure 11: Layout of EXT4 and NTFS filesystem.

respectively. Specifically, we implement such an analysis mechanism for the EXT4 file system in our prototype.

The Fourth Extended File System (EXT4) is split into a sequence of block groups. Each group, typically with a default block size of 4KB, encompasses 32768 blocks, totaling 128MB in length. The total number of block groups is determined by dividing the size of the device by the size of a block group. The simplified layout of a typical EXT4 is depicted in Figure 11a. The superblock contains essential information about the filesystem, including block counts, inode counts, supported features, maintenance data, and more. Meanwhile, the group descriptors are the descriptors associated with each block group and record the locations of data block bitmaps, inode bitmaps, and the inode table. Typically, each group descriptor spans 0x40 bytes. The data block bitmap monitors the utilization status of data blocks within the block group, while the inode bitmap tracks the occupancy of entries in the inode table. In both bitmaps, each bit represents the usage status of a data block or inode table entry. In a regular UNIX filesystem, the inode serves as a data structure for all metadata related to a file. To retrieve information associated with a file, it can navigate through directory files to locate the directory entry corresponding to the file, then access the inode to retrieve metadata and the location of the data blocks for that file.

Based on the layout of the EXT4 file system, we extract information about data block usage from the data block bitmap. Due to potential misalignment between the filesystem and the disk, block numbers in the filesystem do not directly map to those seen in the hypervisor. To resolve this discrepancy, the prototype initially parses the start offset of the filesystem on the disk. It then retrieves the location of the data block bitmap from the corresponding group descriptors and constructs the entire block bitmap based on their block group numbers, adjusted by the offset. With the block usage bitmap, write requests can directly access the disk for unused blocks without influencing the size of the collection.

The New Technology File System (NTFS) consists of several components, including the partition boot sector (PBS), master file table (MFT), a series of metafiles, and data areas. A simplified layout of NTFS is depicted in Figure 11b. The partition boot sector stores boot information, while the master file table contains metadata about all files, directories, and metafile data, similar to the inode data structure in EXT4. In other words, each file on an NTFS is represented by a record in the master file table. The MFT backup serves as a partial copy of the MFT and is utilized if corruption occurs. Additionally, NTFS includes several files that define and organize the file system. The metafile $Bitmap tracks cluster usage on the NTFS filesystem, with each bit in the $Bitmap representing the usage status of a cluster. According to the layout of the NTFS, the data block usage information can be extracted from the $Bitmap, with the need to convert the unit of clusters to blocks. Although we only introduced two filesystems here, other filesystems can also retrieve data block usage information. Only the protection mechanism for unused blocks needs to understand the layout of a filesystem, and we believe that all the other approaches can work in all operating systems/filesystems.

Note that the I/O interception module splits requests into sector-level operations, which may not align with the block size used in the bitmap. Traditionally, for spinning disks, consolidating related blocks nearby minimizes the movement required by the head actuator and disk to access data blocks, leading to improved disk I/O performance. While SSDs lack moving parts, achieving data locality can still increase transfer request sizes while reducing the overall number of requests. Additionally, this locality may concentrate writes on a single erase block, significantly boosting file rewrite speeds. Hence, minimizing fragmentation is essential for optimizing performance. Essentially, data locality is a highly desirable characteristic of a filesystem, and each read/write request to a data may consist of several consecutive sectors with high probability. By leveraging the characteristic of the filesystem, the prototype applies a compromise method wherein a block is marked as used if and only if all the sectors within it are written during a write request initiated by the guest operating system or if a write request is completed that has modified sectors within the block.