Rocky: Replicating Block Devices for Tamper and Failure Resistant Edge-based Virtualized Desktop Infrastructure

Beom Heyn Kim
University of Toronto
Toronto, Ontario, Canada
beomheyn.kim@gmail.com

Hyoungshick Kim
Sungkyunkwan University
Suwon, Republic of Korea
hyoung@skku.edu

ABSTRACT

Recently, edge-based virtual desktop infrastructure (EdgeVDI), which brings the power of virtualized desktop infrastructure to cloudlets closer to users, has been considered as an attractive solution for WAN mobility. However, ransomware and wiper malware are becoming more and more prevalent, which can impose serious cybersecurity threats to EdgeVDI users. Existing tamper-resistant solutions cannot deal with cloudlet failures. In this paper, we propose Rocky, the first distributed replicated block device for EdgeVDI that can recover from tampering attacks and failures. The key enabler is replicating to store a consistent write sequence across cloudlets as an append-only immutable mutation history. In addition, Rocky uses a replication broker to allow heterogenous cloudlets to control replication rates at their pace and reduces both disk space and network bandwidth consumption by coalescing writes for both uplink and downlink. To show the feasibility of Rocky, we implemented Rocky in Java. The experimental results show that Rocky's write and read throughputs are similar to those of a baseline device with 8.4% and 11.9% additional overheads, respectively. In addition, we could reduce repeated writes by 88.5% and 100% for editing presentation slides and a photo, respectively.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**.

KEYWORDS

Edge Computing, VDI, Replication, Block Device, Data Recovery

ACM Reference Format:
Beom Heyn Kim and Hyoungshick Kim. 2021. Rocky: Replicating Block Devices for Tamper and Failure Resistant Edge-based Virtualized Desktop Infrastructure. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3485832.3485886

1 INTRODUCTION

Having access to the same desktop environment from any device and any location has long been an ideal goal of personal computing users. Thus, virtualized desktop infrastructure (VDI) was proposed as an appealing solution that allows users to accomplish that dream. With VDI, a desktop environment is encapsulated by a virtual machine (VM) which runs on a remote server within a local area network (LAN), and its user connects to the VM via remote desktop protocol (RDP) from his or her personal computing device as a thin client. Indeed, there already exist many commercial VDI solutions such as VMWare Horizon, Citrix Virtual Apps and Desktops and Microsoft Azure Virtual Desktop. According to the report from Fior Markets, the VDI market size is expected to reach 38.41 billion US Dollars by 2027 [29]. However, existing VDI solutions are limited for users who need to access their VDI services over the wide-area network (WAN) because of long network latency.

Meanwhile, two newly emerging technologies, 5G and edge computing can shed some light on this problem. It is widely known and expected that, along with 5G network connection, cloudlets [26], defined as server-class computers (Tier-2) sitting between user devices (Tier-3) and the cloud (Tier-1), will significantly reduce the network latency for user devices to use cloud services. Thus, EdgeVDI has been proposed to provide VDI for users who need legacy applications and WAN-mobility by running and migrating VMs across cloudlets near users [33].

Nevertheless, EdgeVDI can be an attractive target for adversaries attempting to compromise the desktop environment with malware. These days, ransomware is one of the most frequently mentioned high-profile malware because it can be directly converted into a significant financial gain via ransoms. EdgeVDI remains as vulnerable as personal computers as vulnerabilities in RDP, legacy applications, or OS can be exploited in the same way. DarkSide, a criminal hacker group behind the Colonial Pipeline attack, compromised contractor's accounts and gain unauthorized access to the VDI [12, 37] to gain a foothold. In addition, ransomware may be embedded in a virtual machine image file by attackers to bypass endpoint detection mechanisms [40]. Although logging off the session may rollback the system image and remove the malware automatically, attackers may have plenty of time to encrypt or delete user data before logging off once they break into the desktop environment.

Many real-world incidents occurred where the victims of ransomware attacks were obliged to pay ransom to the attackers. Recently, Ukrainian police arrested six people behind the Clop ransomware attacks, and their attacks cost universities about 500 million US dollars for the past two years [16]. Last year, the cost due to ransomware incidents between the second half of 2019 and the first half of 2020 was estimated to be over 1 billion US dollars at the minimum [35]. Therefore, it becomes a challenging issue to prevent ransomware attacks.

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3485886

Several researchers have built various tamper-resistant storage systems to protect user data against ransomware [3, 19, 45]. However, those solutions all require modification on hardware architecture or need a special hardware device. Many previous studies have attempted to design ransomware detection methods [2, 22]. This line of works has mainly focused on quickly detecting ransomware activity, trying to reduce the damage done by the attacker but not on recovering data being tampered with or lost. More crucially, existing solutions do not work when a cloudlet on which the VM runs fails. For instance, if a natural disaster like fire destroys a cloudlet infrastructure, those systems protecting user data against ransomware become ineffective to prevent data loss. There can be human errors or component failures leading to catastrophic and cascading failures in a data center [1].

In this paper, we present Rocky, the first distributed replicated block device for tamper and failure-resistant EdgeVDI. It provides security and reliability guarantees related to data availability without requiring any special hardware. First, it can restore the block device back to the state before data is tampered with by performing ransomware or wiper malware attacks. Second, it can recover the coherent block device even if failures lead to the unavailability of one or more cloudlet infrastructures. The key enabler of those guarantees is carefully managing the replication of block devices on distributed cloudlets.

Our key observation is that *replicating a consistent totally-ordered sequence of writes on distributed cloudlets ensures the tamper and failure-resistant distributed block devices*. We designed Rocky's replication protocol based on this observation. Each Rocky runs on a cloudlet, and only one among the ensemble of replicas becomes active for its user to run the VM on it. As writes occur and mutate the state of the block device, Rocky replicates those writes in the same order to other cloudlets. Also, Rocky keeps the write sequence as the append-only immutable mutation history. Therefore, replaying the write sequence up to the point before a tampering attack begins can restore the version of block devices that have not yet been tampered with. In addition, failures of one or more cloudlets can be recovered by using replicas to rebuild the block device on a new cloudlet replacing the unavailable one.

Rocky leverages a special cloudlet called *connector-cloudlet* as a loosely coupled and asynchronous replication broker. The consistent totally-ordered write sequence is replicated across cloudlets via the connector-cloudlet. The connector-cloudlet is a logical component and can be backed by any cloudlet or by a cloud. The connector-cloudlet gives each regular cloudlet the freedom of determining the replication rate for better resource management as cloudlets may be heterogeneous because different service providers may operate them. Consequently, cloudlets may have different capabilities. Moreover, replicating through the broker is more efficient than broadcasting directly between cloudlets because it conserves network bandwidth consumption for uploading.

Rocky employs two novel replication-related techniques: (1) *periodic mutation snapshot update* and (2) *periodic prefetch with snapshot merging*. For periodic mutation snapshot updates, a Rocky cloudlet batches a sequence of writes performed for a period and asynchronously flushes the snapshot of blocks mutated with the batched writes ("dirty blocks"). Because repeated writes to the same block are reduced to the latest write to that block for a period, Rocky can efficiently be used in terms of performance and space.

The periodic mutation snapshot update replicates dirty block snapshots to the connector-cloudlet from which other cloudlets can asynchronously fetch them in advance before they become active ("periodic prefetch"). Because both periodic mutation snapshot and periodic prefetch allow configuring their period, it allows Rocky nodes to determine the replication rate at their own pace considering their resource utilization status. Periodic prefetch uses snapshot merging. Rocky nodes fetch the meta-data about dirty blocks uploaded over the last few periods, reduce multiple mutation snapshots into a single merged mutation snapshot ("snapshot merging"), and fetch the merged snapshot. Thus, repeated writes are even further reduced, which consequently saves the cost of replicating dirty blocks. Rocky keeps multiple versions of blocks so that it can guarantee tamper and failure resistance against malware and failures.

Rocky is implemented in 3K lines of code in Java. We evaluated our prototype by running block I/O workload directly to the block device. The experimental results show that there were 8.4% and 11.9% additional throughput overheads for writes and reads compared to the raw network block device. We also measured how many repeated writes can be reduced through periodic mutation snapshot updates and mutation merging for our simulated workflow editing presentation slides and photos. Consequently, we found repeated writes were reduced 88.5% and 100%, respectively. In addition, we provide the proof that Rocky can restore coherent block device in attack scenarios and that it can recover from failures with minimal data loss.

We make the following novel contributions in this work:

(1) We develop the first tamper-resistant and failure-resistant block device for EdgeVDI.
(2) We devise a novel replication protocol involving a replication broker and employing periodic mutation snapshot update and periodic prefetch with snapshot merging.
(3) We design and implement a prototype distributed replicated block devices implementing the proposed techniques (https://github.com/Kaelus/Rocky).
(4) We evaluate the prototype to show that it is efficient in terms of performance and storage space.

In Section 2, we describe details of the target environment, threat model, failure model, and coherence problem. Then, Section 3 discusses the architecture of Rocky. Subsequently, recovery procedures are presented in Section 4, and Section 5 shows the evaluation results of our empirical study with our prototype system. Section 6 compares our work with related works and we conclude in Section 7.

2 PROBLEM STATEMENT

2.1 Target Environment

This section describes the target environment for which we designed Rocky. Figure 1 illustrates the high-level view of the target environment. Each Rocky block device is installed on a cloudlet, and a VM uses a Rocky block device as a passthrough device. A Rocky block device receives and processes VM's block I/O. For writes, the Rocky block device replicates the VM's disk writes to

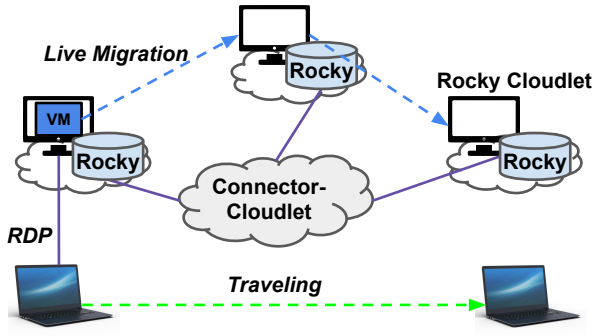

Figure 1: Target Environment. A user accesses the desktop environment contained in a VM via RDP. As the user travels (the green dashed line), the virtual machine follows the user via live migration (the blue dashed line).

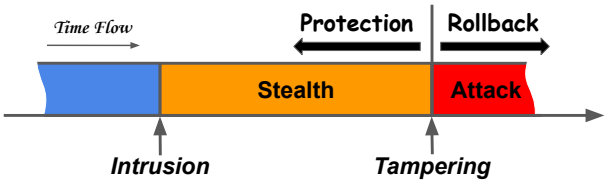

Figure 2: Tampering Attack Problem. The blue bar indicates the writes made while there is no attacker in the system. The orange bar represents the writes made during the stealth phase of malware. The red bar shows malware's tampering writes.

other cloudlets' Rocky block devices via a connector-cloudlet. A set of Rocky block devices forms an ensemble and collectively provides the coherent block device for the VM running on top of it. There is only one active Rocky block device at any given time for an ensemble. As the VM migrates following its user, the Rocky block device of the cloudlet which the VM runs on becomes active, and the previously active device becomes inactive. Thus, only the active Rocky block device can handle VM's block I/O requests and therefore can serialize writes. Unlike conventional VDI solutions, Rocky intends to support a user who requires WAN mobility. The user may bring or use any personal computing device (e.g., desktop, laptop, etc.) to access their desktop environment anywhere beyond a single LAN.

Threat Model. We assume a powerful active adversary. An adversary is assumed to be capable of compromising a victim's thin client device or the desktop environment contained in a VM on a cloudlet. The adversary may mount privilege escalation attacks and can compromise kernels or perform arbitrary privileged operations. The attacker's primary purpose of intrusion is to install malware such as ransomware or wiper malware that degrades the availability of user data aiming the financial gain. We assume that the victim's VM is installed with anti-malware software that can notify any suspicious activity before the attacker taking it down. Although the attacker can be privileged, he or she cannot escape the VM. That is, we trust cloudlets, and the attacker cannot compromise the underlying hypervisor or any other cloudlet infrastructure outside the VM. The connector-cloudlet will not be compromised as it does not expose any interface for attackers inside the VM to exploit.

Failure Model. In this paper, we assume that cloudlets may suffer from benign hardware and software failures that are not adversarial but harmful enough to prevent us from reading or writing disk blocks. For example, hard disks of cloudlets may fail and, therefore, servers may crash. The network can also fail due to Ethernet cable issues or WiFi adapter failures. In addition, software bugs can also cause catastrophic failures and crash servers of cloudlets. Thus, we suppose that failed servers may not be recoverable. For example, a hard disk can be worn out and cannot be booted from. More crucially, cloudlets may suffer from irrecoverable service outage due to disasters such as fire or flood. Because cloudlets are located more closely to each other than cloud data centers are, we think that it will be more likely that a natural disaster can break down multiple cloudlets at the same time. Thus, we assume multiple cloudlets may fail simultaneously. Nevertheless, we expect that there will be at least one correct cloudlet. For example, we suppose at least one cloudlet is located somewhere the natural disaster did not affect. We note that Byzantine faults of cloudlets are out of scope.

2.2 Challenges

Each write is applied to a block device by making a corresponding mutational change to update a data block. For synchronization, each replicated block device must consistently apply the same set of writes in the same order. That means there exists a specific sequence of writes $(w_0, w_1, \ldots, w_{n-1})$ and the state of a block device (S_k, for some $k \in \mathbb{Z}$) sequentially changes as those writes in the sequence are applied.

$$S_0 \xrightarrow{w_0} S_1 \xrightarrow{w_1} \ldots \xrightarrow{w_{n-1}} S_n$$

Tamper Resistance. Data tampering attacks having ransomware or wiper malware generate a series of destructive writes. Malware's activity is divided into two phases: (1) stealth phase and (2) attack phase. Suppose the malware starts its attack phase to tamper user data after the last benign write w_{k-1}. Then, the block device is tampered with $w_k, \ldots, w_{n-1}$. We can recover the block device to the state before tampering begins by not applying $w_k, \ldots, w_{n-1}$ but applying $w_0, \ldots, w_{k-1}$. Figure 2 illustrates this, aiming to protect user data by excluding all those tampering writes issued during the attack phase. Pinpointing the point when the first tampering attack began is out-of-scope, and we rely on anti-malware software to find out that point.

Without an append-only immutable mutation history kept securely, naively replicating and applying writes to backups cannot defend user data against malware. Accordingly, cloud-based file synchronization solutions will replicate tampering writes to each file out-of-order, and, therefore, those solutions cannot appropriately restore the coherent block device state that is not tampered with. In addition, equipping cloudlets with special hardware, such as TEE supports, a specially customized SSD, or a fancy self-encrypting drive, can increase the cost unnecessarily. Rocky is designed to

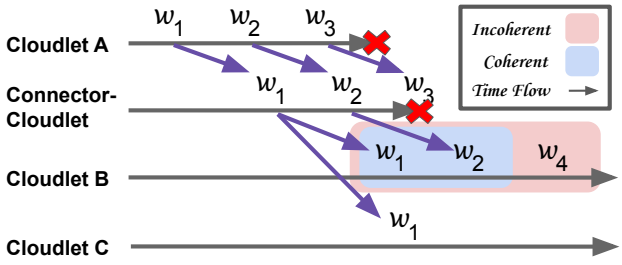

Figure 3: Coherence Problem. Failures can lead to writing loss (w_3), which can make the block devices incoherent. Therefore, to recover coherent block devices, we need to remove recent writes such as w_4 and apply the contiguous write sequence, i.e., w_1, w_2.

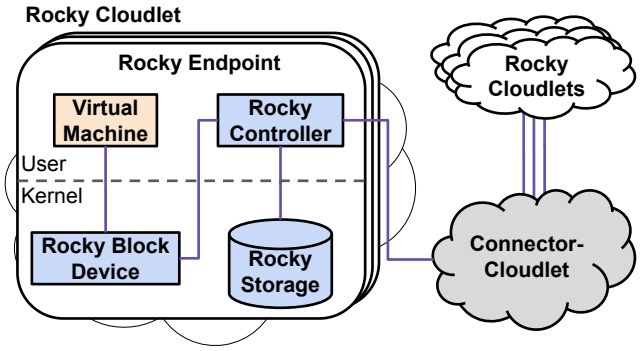

Figure 4: Rocky Cloudlet Architecture. Each cloudlet supporting EdgeVDI is installed with Rocky (components colored in blue) and a VM can run on it. The connector-cloudlet mediates replication between Rocky cloudlets.

defend against a tampering attack by keeping the append-only immutable mutation history across cloudlets without using special hardware.

Failure Resistance. When a block device's state has been changed by a contiguous write sequence, we say the block device is *coherent*. However, if we apply a write sequence that may include some prefix that is not contiguous, we cannot assure that the block device is coherent. Maintaining a coherent block device is one of the most critical properties that block device abstraction should ensure. For example, when a file system is updated by creating a file in some directory, writing the content to the file occurs first, and then writing to the directory occurs next in order to keep the file system consistent even if a failure unexpectedly occurs. If we applied the latter write to the block device but not the former write, then we have a block device that is not coherent.

Although replicating the write sequence to block devices guarantees that replicated block devices are synchronized at a coherent state, it is challenging to know every write sequence in advance that is going to be generated by applications. Therefore, most existing solutions let applications generate block I/O and apply them to one of the replicated block devices first, then asynchronously replicate the write that happened on one block device to another block device to let other devices apply the identical write sequences. Similarly, Rocky's approach is replicating the sequence of writes from a Rocky block device on which VM runs to other Rocky block devices via a connector-cloudlet.

However, if failures occur, a *coherence problem* can occur as the example illustrated in Figure 3. In this example, Rocky on the cloudlet A wrote w_1, w_2, w_3 which were replicated to the connector-cloudlet on time. However, if cloudlet A crashed due to hard disk failure and the connector-cloudlet became unavailable due to a service outage. Meanwhile, the user traveled and used the cloudlet B performing w_4. Cloudlet B replicated w_1, w_2, but failed to replicate w_3 due to the connector-cloudlet's service outage. Cloudlet C has been even slower and replicated only w_1. Here, cloudlet B does not have a coherent block device because it does not apply w_3 but apply w_4 to its block device. Meanwhile, cloudlet C is coherent but stale, as it only applies w_1. We can restore the latest coherent block device by replaying w_1, w_2 but not w_4.

Rocky is designed to provide a function for users to get the latest coherent block device restored only with a given set of correct Rocky block devices.

Note that the problem described in Figure 3 can occur even if we say that w_3 was not a benign write lost due to a failure but a tampering write corrupting some data block. Essentially, both tampering attacks and failures cause the situation where we need to discard all subsequent writes after the write either tampered or lost.

3 ROCKY ARCHITECTURE

3.1 Overview

Rocky is a replicated distributed block device to enhance EdgeVDI's security and reliability against malware attacks inside a VM and various failures of cloudlet infrastructures. To that end, Rocky enables reconstructing a coherent block device that is not tampered.

Rocky Cloudlet. Each Rocky cloudlet maintains *Rocky endpoints*. Each Rocky endpoint consists of a *Rocky block device* (RDB), a *Rocky controller* (RC) and a *Rocky storage* (RS) as illustrated in Figure 4. Each Rocky endpoint of the Rocky cloudlet stores the replica of a VM image containing a user's desktop environment. An ensemble of Rocky endpoints keep those replicas across cloudlets in sync via Rocky's replication protocol. There exists only one Rocky endpoint for an ensemble that can run the user's VM. Other endpoints in the ensemble are periodically replicating new disk writes generated by the VM.

I/O Handling. RDB exports a device file for a VM to run on top by using it as a passthrough device. The VM sends block I/O requests to RDB. Then, RDB redirects it to RC. Each block I/O request is made to either read or write fixed-size disk blocks. Each disk block is associated with a unique integer type identifier (*block ID*). Thus, each block I/O request must specify which disk block it wants to read or write. With the block ID requested, RC looks up meta-data indicating whether the blocks are located locally or remotely. Then, RC forwards requests to either RS or the connector-cloudlet. RS or the connector-cloudlet handles requests and returns a response. RC returns the response to RDB which in turn relays it to the VM.

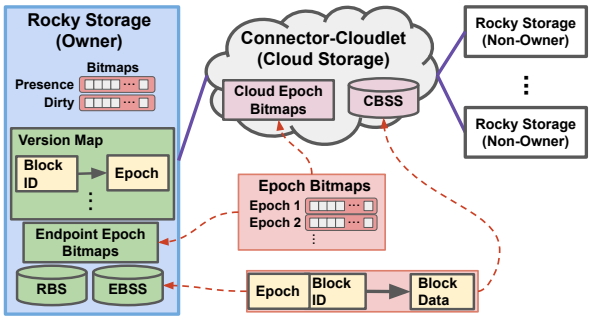

Figure 5: Rocky Storage and Connector-Cloudlet Architecture. Meta-data and block snapshots are stored for tamper and failure resistance. (RC which actually manages communication between cloudlets is not shown.)

Write Serialization. An ensemble of Rocky endpoints must collectively provide a logically single coherent block device. To that end, writes must be serialized. Rocky requires an endpoint to obtain an exclusive ownership from the ensemble. The endpoint with the ownership is the only one that can run a VM and, therefore, can serialize VM's block writes into a consecutive, consistent, totally-ordered write sequence that needs to be replicated to all other endpoints to maintain coherence of a block device. As a user moves, we need to migrate user's VM across cloudlets. To transfer the ownership for the VM between cloudlets, we devised the *ownership-transfer protocol*, which we will explain in more detail below. We define the Rocky endpoint with the exclusive ownership as *owner* and all other endpoints as *non-owner*.

Replication Protocol. Rocky owner batches writes. Then, the owner periodically flushes writes to the connector-cloudlet. The connector-cloudlet stores the write sequence as the immutable mutation history. From the connector-cloudlet, non-owners periodically retrieves new writes in the sequence they have not yet replicated. Therefore, write sequence is asynchronously and periodically streamed by the owner to non-owners via the connector-cloudlet. In short, Rocky's replication protocol is publish-subscribe pattern where the owner publishes new writes, the connector-cloudlet performs the broker role and non-owners subscribes to any new writes published by the owner. Because the owner uploads and non-owners download the streamed write sequence via the connetor-cloudlet, they can adjust the replication rate at their will depending on their own resource utilization status.

Storage Architecture. Figure 5 shows the architecture of Rocky storage and connector-cloudlet (hereafter, "cloud storage"). Rocky storage contains the *raw block store* (RBS), *version map*, *endpoint block snapshot store (EBSS)* and *dirty/presence/epoch bitmaps*. Cloud storage is used in a way that it contains epoch bitmaps and *cloud block snapshot store (CBSS)*. We will explain how each component contributes to Rocky's replication protocol for tamper and failure resistance in more details in the following sections.

3.2 Presence/Dirty Bitmaps

For block I/O handling, RC uses two types of bitmaps: a *presence bitmap* and a *dirty bitmap*. The presence bitmap is used to indicate the freshness of blocks in Rocky storage. If a n-th bit of the presence bitmap is set to 1, then it means the Rocky endpoint has the up-to-date n-th block in Rocky storage. However, if the bit is 0, then it means the copy of the block in Rocky storage is stale. Thus, the presence bitmap directs whether the requested block is stored in its local Rocky storage or should be fetched from cloud storage. The dirty bitmap is mainly used by the owner to record which blocks have been written ("dirtied") in Rocky storage but has not yet been replicated to cloud storage. Similar to the presence bitmap, if the n-th bit of the dirty bitmap is set to 1, then it means the n-th block has been written and therefore is a dirty block.

RC manages its presence bitmap and dirty bitmap to read the latest blocks and replicates writes to other Rocky endpoints. When RC runs on the owner, it accordingly sets or resets presence and dirty bitmap for each incoming block I/O. For an incoming read request, RC looks up the presence bitmap to determine if it can serve a block from Rocky storage. If so, RC reads the block from RBS. Otherwise, RC should get the block from cloud storage. After getting the block from cloud storage, RC stores the block in RBS. Then, RC sets the corresponding bit in the presence bitmap to 1. For an incoming write request, RC directly writes to RBS after setting the corresponding bit in the dirty bitmap. RC does not read a block for a write request because the block will be overwritten by the write operation. RC sends the dirty bitmap along with writes to the cloud storage to notify other non-owner endpoints about new writes. Then, the corresponding indexes of the dirty bitmap are reset after flushing those updates to cloud storage. Meanwhile, when RC runs on a non-owner, it resets bits in the presence bitmap as it receives the meta-data about new writes from the cloud storage.

3.3 Periodic Mutation Snapshot Update

To ensure a coherent block device, Rocky replicates a contiguous write sequence across distributed Rocky endpoints. However, if we replicate every write, resource consumption is wastefully increased by unnecessarily sending blocks that are going to be overwritten by new writes. To solve this issue, Rocky hoards writes and sends only the latest version. More specifically, Rocky periodically updates cloud storage with the snapshot of dirty blocks at the end of each period. Rocky allows the owner to adjust the period of update, called *epoch*. Therefore, Rocky sequentially sends each fragment of a write sequence at the end of each epoch. For example, suppose we have a long sequence of writes, W:

$$w_{11}, w_{12}, \ldots, w_{1n}, w_{21}, w_{22}, \ldots, w_{2m}$$

With this, the state of the block device mutates from its initial state S_0 to S_{n+m}:

$$S_0 \xrightarrow{w_{11}} S_1 \xrightarrow{w_{12}} \ldots \xrightarrow{w_{1n}} S_n \xrightarrow{w_{21}} S_{n+1} \xrightarrow{w_{22}} \ldots \xrightarrow{w_{2m}} S_{n+m}$$

, or simply

$$S_0 \xrightarrow{W} S_{n+m}$$

Suppose W is fragmented into two epochs, which can be varied depending on the configuration. The first epoch sends a write sequence fragment, W_1:

$$w_{11}, w_{12}, \ldots, w_{1n}$$

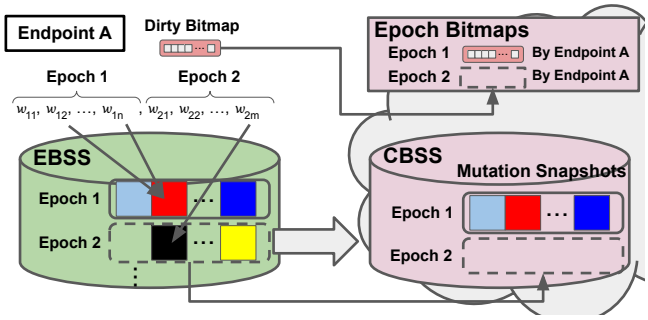

Figure 6: Periodic Mutation Snapshot Update. The RC writes the snapshot of dirty blocks to EBSS and then replicates it to CBSS for each epoch. Dirty bitmaps for epochs are also replicated to the cloud as epoch bitmaps.

The other epoch sends another write sequence fragment, W_2:

$$w_{21}, w_{22}, \ldots, w_{2m}$$

Thus, the state mutation can be seen as:

$$S_0 \xrightarrow{W_1} S_n \xrightarrow{W_2} S_{n+m}$$

To store old write sequence fragments as an immutable mutation event for tamper-resistance, Rocky packages each fragment of write sequence into a *mutation snapshot*. A mutation snapshot is a collection of dirty blocks written during an epoch. For example, among a write sequence W_1, if w_{12} and w_{1n} write to the same block whose block ID is 1, then we reduce them to just w_{1n} as it is the latest write to the block ID 1 for the epoch. Then, Rocky just replicate the latest snapshot of the block ID 1 that is the block applied with w_{1n} at the end of the epoch. By overwriting the block device with blocks packaged in mutation snapshots, we can mutate the state of the block as if a corresponding write sequence is applied to the block device. Suppose that the mutation snapshot of the first epoch, M_1 and that of the second epoch, M_2. Then, applying M_1 to the block device with the initial state S_0, we can mutate the state of the block to S_n, and so on, as if we have applied W_1 and W_2:

$$S_0 \xrightarrow{M_1} S_n \xrightarrow{M_2} S_{n+m}$$

This technique is used by a procedure called *Periodic Mutation Snapshot Update (**PMSUP**)* which Figure 6 depicts. PMSUP works as follow: (1) for dirty blocks referring to the dirty bitmap, RC first needs to translate the dirty block's ID to a key that is the combination of the current epoch appended by the corresponding block ID delimited by a colon; (2) RC sends those dirty blocks to EBSS and CBSS as values to those translated keys; (3) RC writes its dirty bitmap to the cloud storage service by writing that bitmap to the key `<Epoch>-bitmap` where the `<Epoch>` is the current epoch during which those changes occurred—the dirty bitmap uploaded to the cloud storage service for each epoch is simply called *epoch bitmap*; (4) RC resets the corresponding bits in the dirty bitmap for those dirty blocks.

Epoch is essentially a version for each block. It indicates when the block was taken as a block snapshot and made "public" to others by writing them to the cloud. This is why Rocky indexes each block in EBSS or CBSS by the combination of epoch and block ID. On an

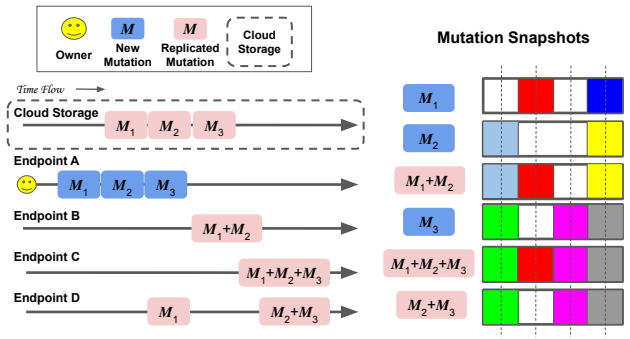

Figure 7: Periodic Prefetch. Endpoint A is the owner generating a sequence of mutation snapshots M_1, M_2 and M_3. Other endpoints periodically prefetch at their own pace. Blank squares in mutation snapshots indicate that there is no write for the corresponding block.

incoming read request, RC may need to fetch blocks from CBSS but need to know which version of blocks it needs to request. To that end, Rocky endpoint maintains *version map* consisting of a map between block ID and the latest version (i.e., epoch) for that. When RC periodically fetches epoch bitmaps, it updates its version map to reflect new writes made since the last period.

3.4 Periodic Prefetch and Snapshot Merging

Non-owner's RC periodically gets new block snapshots before becoming the owner (*periodic prefetch*) to minimize the overhead of getting blocks from the cloud storage on-demand when it becomes the new owner. To begin, RC gets epoch bitmaps from cloud storage—those epoch bitmaps should be ones newly uploaded by the owner. Once epoch bitmaps are downloaded, RC calculates what version of block snapshots it needs to get to not download block snapshots overwritten by the latest one. For example, suppose mutation updates M_1 and M_2 were made by the owner endpoint A since the endpoint B's last periodic prefetch. The state of the raw block device on endpoint A has been changed:

$$S_0 \xrightarrow{M_1} S_n \xrightarrow{M_2} S_{n+m}$$

If some block snapshots in M_1 are overwritten by block snapshots in M_2, it is wasteful to download those block snapshots in M_1 that are overwritten by M_2's. We devised *snapshot merging* technique to solve this issue. To construct the merged mutation snapshot, RC picks the latest snapshot of each block from mutation snapshots. In this way, RC gets the "merged" mutation snapshot consisting of the latest block snapshots for all dirty blocks needed to be replicated.

Figure 7 shows how snapshot merging works with an example scenario. If endpoint B calculates the mutation snapshot, $M_1 + M_2$, then by applying $M_1 + M_2$ endpoint B can sync with endpoint A more efficiently than downloading them separately:

$$S_0 \xrightarrow{M_1+M_2} S_{n+m}$$

Also, note how each endpoint may vary the rate of periodic prefetch and the resulting mutation snapshot becomes different based on that rate. Merged snapshots for endpoints B, C and D are $M_1 + M_2$, $M_1 + M_2 + M_3$ and $M_2 + M_3$, respectively.

The periodic prefetch procedure (***PPP***) is as follow: (1) RC begins with prefetching by downloading all epoch bitmaps uploaded by the owner since the last prefetch; (2) RC reflects them into its presence bitmap by resetting bits in the presence bitmap if the corresponding bit is set in at least one of the epoch bitmaps; (3) RC starts getting blocks from the cloud storage; (4) newly retrieved blocks are written to RBS and EBSS; (5) RC updates the version map accordingly by referring to epoch bitmaps and sets the presence bitmap accordingly.

3.5 Ownership Transfer

For coherence, we allow only one owner at any given time. To conduct the ownership-transfer protocol, Rocky endpoints use cloud storage for the coordination. The RC allocates the key `owner` to indicate both the existence of the owner. If the key does not exist, no owner is currently active. Therefore, the RC of the non-owner can become a new owner simply by creating the key. However, if the key exists, there exists an active owner currently. Then, to demand the ownership transfer, the RC sends a signalling message to the current owner via a cloud messaging service. After that, RC waits until the key `owner` gets deleted by the current owner. If RC waits too long, it times out and sends a notification message to its user via screen, email or SMS.

Upon the arrival of the ownership-transfer request, the current owner performs: (1) stops accepting any more block I/O during the ownership transfer, (2) flushes all writes to raw block storage, (3) performs the last periodic mutation snapshot update (4) deletes the key `owner` on the cloud, and (5) sends a signalling message to the endpoint requesting the ownership to notify the completion of the ownership relinquishment. Subsequently, RC requesting the ownership transfer receives the signalling message and finally become a new owner by creating the key `owner` with its own endpoint ID. If the owner waits for the key `owner` to change but time out, then it sends a notification message to its user. Lastly, the new owner downloads all epoch bitmaps that the owner did not see, creates a new dirty bitmap, and by referring to epoch bitmaps, resets bits in the presence of bitmaps and updates the version map.

3.6 Scheduled Checkpoint

Rocky can perform a old block snapshots removal via scheduled checkpoints. During the scheduled checkpointing, the owner should stop user's VM, read and write every block, and flush writes for the epoch called checkpoint epoch, e_c. Then, endpoints finish with prefetching. Subsequently, endpoints confirm that an anti-malware and failure detector that there has been no tampering attack and no cloudlet failure. After that, endpoints and the cloud storage can clean up their old epoch bitmaps and old block snapshots in EBSS/CBSS added prior to e_c except for those for last checkpoint epochs. Each checkpoint epoch's bitmap will be marked as a scheduled checkpoint so that it does not get garbage-collected during the future scheduled checkpointing. The period of scheduled checkpoints can be determined based on user's policy on storage space utilization.

4 RECOVERY PROCEDURES

Tampering attacks and cloudlet failures both result in degrading data availability. Once data gets corrupted by malware's tampering attacks, users cannot access their data. If cloudlet failures occurred, user data on failed cloudlets cannot be accessed. If the failure is permanent like hard disks destroyed by natural disasters, then data may not be recovered forever. Under our threat and failure models, there are three possible scenarios of suffering from tampering attacks and cloudlet failures: (1) tampering attacks with no failure, (2) failures but no tampering attack and (3) tampering attacks and failures. Rocky's recovery procedures can deal with those situations.

We rely on the assumption that an conventional anti-malware and a failure detector are installed in-place, properly configured and started running already. There exists anti-malware that can report tampering attacks after seeing some data gets tampered by malware [2]. We suppose such an anti-malware can correctly pinpoint and notify the specific epoch when tampering attacks began. Additionally, we expect to see an additional component that periodically pinging Rocky endpoints with a heartbeat message to detect failures. Thus, we suppose there will be notification from either the anti-malware or a failure detector regarding tampering attacks or cloudlet failures. They broadcast to Rocky endpoints. Once Rocky endpoints receive notification, they immediately stops block I/O processing, periodic updating and prefetching. Then, Rocky endpoints start running the recovery procedure accordingly.

4.1 Tampering Attack Recovery

Rocky runs tampering attack recovery procedure when an anti-malware sends the notification to Rocky endpoints. When the notification from a anti-malware is received by a Rocky endpoint, the endpoint retrieves the epoch e_a specified by the anti-malware as the epoch when tampering attacks started. Rocky endpoint updates their data structures, that are epoch variable for the last epoch e_p it finished with prefetch for, its version map and its dirty and presence bitmaps. More specifically, each endpoint starts with comparing e_p with e_a. If $e_a <= e_p$, then the endpoint will set its $e_p := e_a - 1$. Also, they download epoch bitmaps from the cloud storage, and also update their version map using epoch bitmaps updated up until, $e_a - 1$. In addition, for the case $e_a <= e_p$, the endpoint sets its presence bitmap to one for all indices. Additionally, the owner endpoint should flush existing dirty block snapshots and reset bits in its dirty bitmap for those batched dirty block snapshots. For a connector-cloudlet, Rocky discards epoch bitmaps that are for epochs greater than $e_a - 1$. Then, Rocky can continue operating without data being tampered as long as the anti-malware was accurate about the beginning time of tampering attacks.

4.2 Tamper-Resistant Failure Recovery

Rocky triggers failure recovery procedure when a failure detector notifies Rocky endpoints about the failures of cloudlets. Rocky additionally consider tampering attack recovery as well. If a tampering attack was mounted, an anti-malware should notify with the epoch, e_a, when the attack began. If a cloud does not fail, but endpoints are failed, necessary data is all contained on the cloud storage in its epoch bitmaps and CBSS. The procedure is trivial as follows: (1) replace failed component with new ones, (2) remove every bitmap

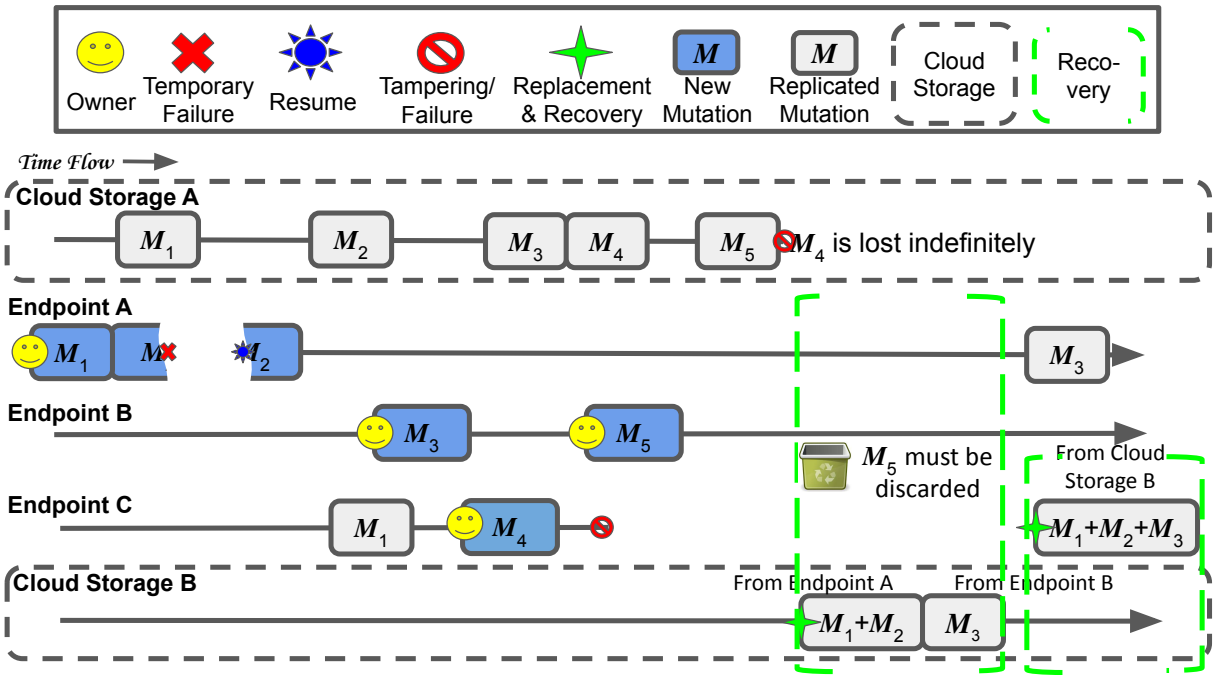

Figure 8: Failure Recovery. Endpoint A generates mutation snapshots M_1 and M_2 although it may temporarily fail during the upload. A user travels to endpoint B to generate M_3, travels to endpoint C to generate M_4, and travels back to endpoint B to generate M_5. Meanwhile, endpoint C fails permanently after downloading M_1 and generating M_4 due to a hardware problem and then cloud storage A fails due to a devastating failure resulting in data loss. The user switches to cloud storage B and triggers Rocky failure recovery procedure. Endpoint A sends cloud storage B its snapshot $M_1 + M_2$, which is the most up-to-date among endpoints A and B. Then, endpoint B sends M_3 but discard M_5 because M_4 is completely lost with endpoint C and cloud storage A's failures (note that M_4 may have been the beginning of a tampering attack). Endpoint C catches up by replicating the merged snapshot $M_1 + M_2 + M_3$ after the failure recovery.

added after $e_a - 1$ from the cloud epoch bitmaps. (3) let new endpoints reconstruct their version map, endpoint epoch bitmaps, RBS, and EBSS using epoch bitmaps and CBSS on the cloud storage by having new endpoints perform prefetch from the beginning.

When the cloud storage fails, Rocky recovers using data stored on endpoints after removing tampering effects if attacks were mounted. Here, some of endpoints may be failed but Rocky's recovery procedure below can recover the block device state with minimal data loss with those endpoints still not failed. One of endpoints becomes a *coordinator* which can be specified by the user and known to every endpoints in advance. The user may specify the priority for each endpoint to become a coordinator in case some endpoints fail and cannot become the coordinator. Then, the coordinator learns from the failure detector about endpoints not failed which forms a *recovery committee*.

Then, the coordinator starts running the initialization procedure (***IP***): (1) every endpoint receives e_a from an anti-malware; (2) every endpoint uploads the epoch of their last endpoint epoch bitmap, e_1, that is $e_1 < e_a$; (2) every endpoint uploads the epoch for which they finished with prefetching, e_2, that is $e_2 < e_a$; (3) the coordinator reads all e_1's and e_2's; (4) the coordinator designates the epoch leader l_1, which is the endpoint uploaded the largest e_1; (5) the coordinator designates the prefetch leader l_2, which is the endpoint uploaded the largest e_2.

Subsequently, the following initial recovery procedure (***IRP***) runs to recover epoch bitmaps and CBSS that is the most coherent one prior to e_a: (1) l_1 uploads endpoint epoch bitmaps to the cloud storage as a new cloud epoch bitmaps; (2) l_2 uploads EBSS to the cloud storage as a new CBSS; (3) the coordinator downloads epoch bitmaps from the cloud storage and scans epoch bitmaps starting at e_2 and sequentially add each subsequent epoch (greater than the previous epoch only by one) into the list L if there is an endpoint in the recovery committee which wrote the epoch bitmap for; (4) stop as soon as we found the epoch for which there is no owner endpoint in the recovery committee.

Finally, we further forward recovery (***FRP***) to minimize data loss by having endpoints to write more recent mutations which may exist across endpoints' EBSS: (1) the coordinator uploads L and deletes epoch bitmaps for epochs newer than those included in L; then notifies all other endpoints via the messaging service; (2) endpoints download L; (3) endpoints upload dirty blocks for the epoch in L for which they were the owner; (4) endpoints notify the coordinator once they are done; (5) once the coordinator is notified by all endpoints in the recovery committee, it notifies all others about the completion of the recovery procedure.

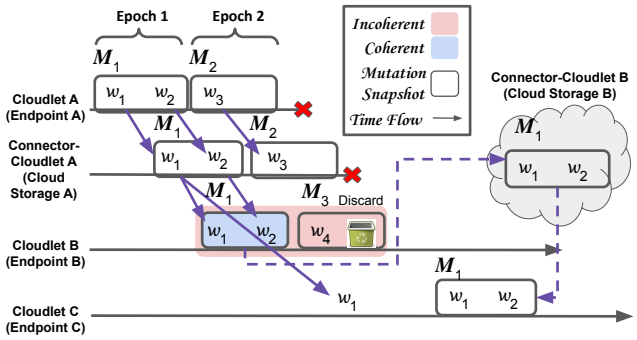

Figure 9: Revisiting the example of the coherence problem, which can be solved by utilizing mutation snapshots. Each write is divided into a mutation snapshot, and a contiguous sequence of mutation snapshots is needed to recover the coherent block device. Endpoint C and cloud storage B recover the coherent block device by applying M_1, as M_2 gets lost and M_3 is not contiguous. M_3 must be discarded on endpoint B.

Figure 8 gives an example that illustrates the high-level of Rocky's failure recovery procedure. Suppose that endpoint C and cloud storage A create a hole in the sequence of mutation snapshots because the mutation M_4 is lost permanently (either due to a tampering attack or a failure). In this case, M_5 must be discarded on endpoint B during the failure recovery to recover the coherent block device that has not been tampered. Consequently, the coherent block device state we can recover is the one with a mutation snapshot $M_1 + M_2$ from endpoint A and M_3 from endpoint B applied sequentially. Once cloud storage B gets the mutation snapshots uploaded, endpoint C can replicate the merged snapshot $M_1 + M_2 + M_3$ at once.

Any component of Rocky can be tampered or failed at any point in time and unexpectedly become unavailable. If a Rocky endpoint temporarily fails due to network connection failures for a while or rebooting the machine, it can simply resume the remaining work it was doing before the failure. Because a Rocky endpoint keeps prefetching blocks for each mutation snapshot sequentially, any Rocky endpoints contain a coherence snapshot of the block device.

Figure 9 illustrates how Rocky's periodic mutation snapshot updates, periodic prefetch, and snapshot merging features can solve the coherence problem described in Section 2. The key idea is that each write sequence can be divided into multiple chunks of mutation snapshots. Then, endpoints can periodically merge mutation snapshots and prefetch to mutate endpoints' block storage states coherently. On the recovery, we start with finding the snapshot that is the most up-to-date among available endpoints. We then make the block storage state fast-forwarded by applying subsequent mutation snapshots sequentially until we find any discontinuity.

5 EVALUATION

5.1 Prototype Implementation

Rocky is implemented in about 3K lines of Java code (https://github.com/Kaelus/Rocky). It is based on a variant of NBD implementations backed by FoundationDB (https://github.com/spullara/nbd.git). The

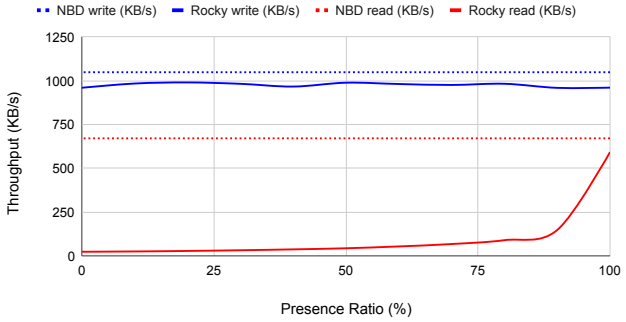

Figure 10: Read/Write Throughput Comparison between Rocky and NBD.

NBD kernel module is equivalent to Rocky block device, which passes block I/O to the user-level Rocky controller, which is the main engine of the system implementing periodic mutation snapshot updates and periodic prefetch and snapshot merging. For endpoint epoch bitmaps, version map, endpoint block snapshot store, Rocky uses LevelDB. For the cloud storage, Rocky uses the AWS DynamoDB service, located in Seoul, Republic of Korea (ap-northeast-2).

5.2 Throughput Measurement

Figure 10 demonstrates the benefit of prefetching. All workloads for our experiments run on a machine equipped with Intel Core2 Quad CPU where each core runs at 2.83 GHz, 8 GB RAM, and Samsung SSD 860 EVO 500GB. We ran a workload, writing and reading 2 MB of data to the block device directly using the well-known 'dd' utility tool. To remove the buffer cache effect, we flushed all buffer caches to the disk before running the workload. NBD is the baseline that does not involve network communication with the AWS DynamoDB service and Rocky's implementation.

With Rocky, we varied the percentage of blocks locally present by setting and resetting bits in the presence bitmap accordingly. When every block is present locally, there were 8.4% and 11.9% additional throughput overheads for writes and reads, respectively. Write performance of Rocky is almost the same as NBD's all the time because processing writes in Rocky does not cause fetching from the cloud storage. However, read performance is affected dramatically depending on the percentage of blocks locally present. We found that the major performance bottleneck is fetching from the remote cloud storage. The performance drops exponentially as the number of blocks fetched from the cloud storage increases. Therefore prefetching as many blocks as possible is very important to reduce perceivable performance degradation.

5.3 Reduction Ratio Measurement

We also measured the benefit of reducing repeated writes to the same blocks by mutation snapshot and snapshot merging. We simulated a workflow involving photo editing and presentation slides updating to see how many repeated blocks could be reduced. The result of this study is presented in Table 1. Assuming Rocky uploads a mutation snapshot after running the workflow, we analyzed the

Table 1: Mutation generated by a workflow of photo editing and presentation slides creating. We indicate the number of new 'WRITES' generated at each step of the workflow and the new 'BLOCKS' written by those writes.

Metric	Copy-P	Edit-P	Create-S	Edit-S
WRITES	8600	48	2096	1040
BLOCKS	8600	0	2056	120

ratio of the number of writes to the number of blocks newly added to the mutation snapshot as our measurement metric, *reduction ratio*, for the entire workflow and each task.

Initially, we started with an empty block device and created a file system on it by running 'mkfs.ext4' on the device. Then, we mounted the device on a host file system. Subsequently, our workflow started with copying a high-resolution photo of size 4.3 MB to the mount point of the Rocky block device. This phase is labeled as *Copy-P*. The photo was then edited using the 'ImageMagick' photo editing application. Editing includes cropping, rotating, adding frames, drawing a few lines and applying several miscellaneous special effects, which is labeled as *Edit-P*.

After that, we created new presentation slides using 'LibreOffice.' We inserted a high-resolution photo of size 438.9 KB into the slide. Then, we added several new slides with randomly typed texts along with few new shapes added. This step is labeled as *Create-S*. Lastly, we did some more editing to the slides by adding more slides with more texts and shapes, which is represented by *Edit-S*.

Overall, 11784 writes were generated, while 10776 blocks were newly included in the mutation snapshot uploaded after executing the aforementioned workflow. Therefore, the reduction ratio is 8.6% in total, but the reduction ratio is varied greatly depending on the type of tasks. For *Edit-P*, there was the 100% reduction ratio because even though there were 48 writes, no blocks were newly written after *Copy-P*. Also, for *Edit-S*, we could gain the 88.5% reduction ratio. Thus, we observe the tendency that tasks updating existing files may gain significant advantages from mutation snapshot and snapshot merging. However, we also like to note that tasks creating a new file are not taking a significant benefit from our techniques, as those tasks are likely to lead to a multitude of blocks that need to be newly included in the mutation snapshot. For instance, *Copy-P* shows 0% reduction ratio and *Create-S* shows only 1.9% reduction ratio.

6 RELATED WORK

There have been many ransomware detection works [2, 4, 10, 15, 18, 22, 23, 30, 31, 34, 36]. However, those previous works do not discuss a recovery mechanism. Many works also provide means to prevent or recover for tamper-resistant storage systems against malware [8, 19, 24, 26, 39, 44, 45]. Nonetheless, those previous proposals do not work for replicated block devices but for a single block device.

Several storage systems placing data on the edge for endpoints have been studied recently [13, 17, 41]. Nevertheless, those systems usually mainly provide a key-value store interface and do not solve the coherency problem. Although these systems can provide low-latency over a wide-area network over the cloud, they cannot provide a recoverable coherent block device abstraction in the presence of failures of for both endpoints and the cloud and cannot provide tamper-resistance against malware.

A couple of research projects have explored how to build a system image that can be replicated over a shared storage infrastructure for endpoints [25, 32]. The Collective has developed their solution for the enterprise environment where all endpoints are connected within the enterprise's private network, so it has not considered the problems occurred by the shared storage server hosted on the shared infrastructures like cloudlets. In addition, the Internet Suspend/Resume project performed several empirical studies for realizing this idea. Also, ISR project maintainers proposed to use edge computing for legacy applications via EdgeVDI [33] Nonetheless, their proposals do not consider security and reliability aspects of EdgeVDI.

Also, there have been many existing works for securing distributed file systems in the presence of the untrusted server component, such as SUNDR, Sirius, Plutus [14, 21, 27]. However, they cannot provide availability on the server failure. Hourglass, PDP, POR, DepSky, and Hail [5–7, 20, 42] are exploring server-side solution for untrusted cloud either by applying cryptography techniques in a novel way or by using multiple cloud service providers. However, they cannot provide block device abstraction and therefore do not support coherency.

In terms of protecting consistency guarantee, Depot, Sporc, and Venus [11, 28, 38] are related to protecting data consistency, similar to Rocky's coherency guarantee. However, their solutions are not suitable for edge/fog computing environments and they are not tamper-resistant solutions against malware. Salus and Windows Azure [9, 43] ensures a strong consistency guarantee, but they are not for edge computing but for the enterprise environment.

7 CONCLUSION

As 5G and edge computing technologies are emerging, we will see the increasing number of applications taking benefits of short network latency. EdgeVDI has been proposed recently as an application that can provide a desktop environment to users needing legacy applications and WAN-mobility. Among many challenges on the road, two most significant problems are how to protect against data tampering malware and failures, affecting data availability. We propose a distributed replicated block device, Rocky, that enables tamper and failure resistant EdgeVDI. Rocky stores a totally-ordered contiguous write sequence as an append-only immutable mutation history and replicates it across multiple cloudlets. Evaluating our prototype, we found only about 10% performance overhead is required to provide a recoverable coherent block device abstraction along with 88.5% to 100% reduction ratio for repeated writes.

ACKNOWLEDGMENTS

The authors would thank anonymous reviewers. Hyoungshick Kim is the corresponding author. This work was supported Information & communications Technology Promotion grant funded by the Korea government (No.2018-0-00532).

REFERENCES

[1] Lawrence Abrams. 2021. Amazon AWS Outage Shows Data in the Cloud is Not Always Safe. https://www.bleepingcomputer.com/news/technology/amazon-aws-

outage-shows-data-in-the-cloud-is-not-always-safe/ (last accessed: 03/18/2021).
[2] Muhammad Ejaz Ahmed, Hyoungshick Kim, Seyit Camtepe, and Surya Nepal. 2021. Peeler: Profiling Kernel-Level Events to Detect Ransomware. *CoRR* abs/2101.12434 (2021).
[3] Jinwoo Ahn, Junghee Lee, Yungwoo Ko, Donghyun Min, Jiyun Park, Sungyong Park, and Youngjae Kim. 2020. DISKSHIELD: A Data Tamper-Resistant Storage for Intel SGX. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS)*.
[4] Bander Ali Saleh Al-rimy, Mohd Aizaini Maarof, and Syed Zainuddin Mohd Shaid. 2018. A 0-Day Aware Crypto-Ransomware Early Behavioral Detection Framework. In *Proceedings of the 2nd International Conference of Reliable Information and Communication Technology (IRICT)*.
[5] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable Data Possession at Untrusted Stores. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*.
[6] Alysson Neves Bessani, Miguel P. Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2011. DepSky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of the 6th European Conference on Computer systems (EuroSys)*.
[7] Kevin D. Bowers, Ari Juels, and Alina Oprea. 2009. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*.
[8] Kevin R.B. Butler, Stephen McLaughlin, and Patrick D. McDaniel. 2008. Rootkit-Resistant Disks. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*.
[9] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*.
[10] 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 Conference on Computer Security Applications (ACSAC)*.
[11] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. 2010. SPORC: Group Collaboration using Untrusted Cloud Resources. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[12] Scott Ferguson and Doug Olenick. 2021. FBI: DarkSide Ransomware Used in Colonial Pipeline Attack. https://www.bankinfosecurity.com/fbi-darkside-ransomware-used-in-colonial-pipeline-attack-a-16555 (last accessed: 06/18/2021).
[13] A. Gheorghe, C. Crecana, C. Negru, F. Pop, and C. Dobre. 2019. Decentralized Storage System for Edge Computing. In *Proceedings of the 18th International Symposium on Parallel and Distributed Computing (ISPDC)*.
[14] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. 2003. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the 10th Symposium on Network and Distributed System Security (NDSS)*.
[15] José Antonio Gómez-Hernández, L. Álvarez-González, and Pedro García-Teodoro. 2018. R-Locker: Thwarting ransomware action through a honeyfile-based approach. *Computer Security* 73 (2018), 389–398.
[16] Jonathan Greig. 2020. Ukrainian police partner with US, South Korea for raid on Clop ransomware members. https://www.zdnet.com/article/ukranian-police-partner-with-us-south-korea-for-raid-on-clop-ransomware-members/ (last accessed: 06/18/2021).
[17] Harshit Gupta and Umakishore Ramachandran. 2018. FogStore: A Geo-Distributed Key-Value Store Guaranteeing Low Latency for Strongly Consistent Access. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems (DEBS)*.
[18] Manabu Hirano and Ryotaro Kobayashi. 2019. Machine Learning Based Ransomware Detection Using Storage Access Patterns Obtained From Live-forensic Hypervisor. In *Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*.
[19] 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 (CCS)*.
[20] Ari Juels and Burton S. Kaliski Jr. 2007. PORs: Proofs of Retrievability for Large Files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*.
[21] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. 2003. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*.
[22] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. 2016. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC)*.
[23] Amin Kharraz and E. Kirda. 2017. Redemption: Real-Time Protection Against Ransomware at End-Hosts. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*.
[24] 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 (ASIACCS)*.
[25] Michael Kozuch and M. Satyanarayanan. 2002. Internet Suspend/Resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*.
[26] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the 13th European conference on Computer systems (EuroSys)*.
[27] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[28] Prince Mahajan, Srinath T. V. Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Michael Dahlin, and Michael Walfish. 2010. Depot: Cloud Storage with Minimal Trust. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[29] Fior Markets. 2021. Global Virtual Desktop Infrastructure Market Is Expected to Reach USD 38.41 billion by 2027 : Fior Markets. https://www.globenewswire.com/news-release/2021/02/05/2170486/0/en/Global-Virtual-Desktop-Infrastructure-Market-Is-Expected-to-Reach-USD-38-41-billion-by-2027-Fior-Markets.html (last accessed: 06/18/2021).
[30] Shagufta Mehnaz, Anand Mudgerikar, and Elisa Bertino. 2018. RWGuard: A Real-Time Detection System Against Cryptographic Ransomware. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*.
[31] Daniel Morato, Eduardo Berrueta, Eduardo Magaña, and Mikel Izal. 2018. Ransomware early detection by the analysis of file sharing traffic. *Journal of Network and Computer Applications* 124 (2018), 14–32.
[32] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. 2003. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of the 16th Large Installation Systems Administration Conference (LISA)*.
[33] Mahadev Satyanarayanan, Thomas Eiszler, Jan Harkes, Haithem Turki, and Ziqiang Feng. 2020. Edge Computing for Legacy Applications. *IEEE Pervasive Computing* 19, 4 (2020), 19–28.
[34] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin R. B. Butler. 2016. CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data. In *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS)*.
[35] Help Net Security. 2020. The ransomware plague cost the world over $1 billion. https://www.helpnetsecurity.com/2020/11/26/ransomware-cost-2020/ (last accessed: 06/18/2021).
[36] Daniele Sgandurra, Luis Muñoz-González, Rabih Mohsen, and Emil C. Lupu. 2016. Automated Dynamic Analysis of Ransomware: Benefits, Limitations and use for Detection. *CoRR* abs/1609.03020 (2016).
[37] SNIR BEN SHIMOL. 2021. How Dropbox sacrifices user privacy for cost savings. https://www.varonis.com/blog/darkside-ransomware/ (last accessed: 06/18/2021).
[38] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. 2010. Venus: Verification for Untrusted Cloud Storage. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop (CCSW)*.
[39] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. 2000. Self-Securing Storage: Protecting Data in Compromised System. In *Proceedings of the 4th Conference on Symposium on Operating System Design and Implementation (OSDI)*.
[40] Pedro Tavares. 2020. Ragnar locker malware: what it is, how it works and how to prevent it | Malware spotlight. https://resources.infosecinstitute.com/topic/ragnar-locker-malware-what-it-is-how-it-works-and-how-to-prevent-it-malware-spotlight/ (last accessed: 06/18/2021).
[41] Animesh Trivedi, Lin Wang, Henri E. Bal, and Alexandru Iosup. 2020. Sharing and Caring of Data at the Edge. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*.
[42] Marten van Dijk, Ari Juels, Alina Oprea, Ronald L. Rivest, Emil Stefanov, and Nikos Triandopoulos. 2012. Hourglass Schemes: How to Prove that Cloud Files Are Encrypted. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*.
[43] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. 2013. Robustness in the Salus Scalable Block Store. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI)*.

[44] A. Young and Moti Yung. 1996. Cryptovirology: extortion-based security threats and countermeasures. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*.
[45] Lianying Zhao and Mohammad Mannan. 2019. TEE-aided Write Protection Against Privileged Data Tampering. *CoRR* abs/1905.10723 (2019).

A CORRECTNESS PROOF

We define a block device to be *coherent* if and only if there exists a consistent totally-ordered write sequence transitioned the state of the block device.

Theorem 1. Under normal circumstances with no failure, Rocky maintains a coherent block device.

Proof. By contradiction, suppose a Rocky block device, B, is not coherent after applying a state mutation M_i where $0 < i < n$. Because B is not coherent after applying M_i, M_i should contain a write sequence W_i that is deviated from a W_i' contained in M_i' such that $W_1, \ldots, W_{i-1}, W_i'$ is the consistent totally-ordered write sequence. Step 2 of PMSUP has o_i upload M_i' containing W_i' to its EBSS and CBSS. There are two possibilities for B to apply mutation snapshots. First, at Step 4 of PPP, B downloads M_i' and apply W_i' to its RBS. Second, while B is the owner, B accordingly fetches blocks in M_i' from CBSS on demand. By construction, for e_i, B always applies blocks written by W_i' but never by W_i, which contradicts our assumption. Thus, there is no Rocky block device that is not coherent. □

Theorem 2. Assuming an anti-malware correctly pinpoints and notifies with the epoch during which the tampering attack has first begun, Rocky can recover from tampering attacks by restoring an coherent block device which is not tampered.

Proof. Let us call a Rocky block device as B which has the initial state S_0. Suppose B's state changes from S_{i-1} to S_i after applying the mutation snapshot M_i containing writes tampering user data occurred during the epoch e_i. Then, the anti-malware notifies Rocky endpoints with the epoch e_i to indicate when tampering attacks began. Rocky runs the tampering attack recovery procedure and rolls back B's state to become S_{i-1}. By contradiction, suppose S_{i-1} is tampered. That is, there must exist at least one mutation snapshot M_t that tampered the block device state from S_{t-1} to S_t for $0 \leq t \leq i-1$. It means that there was an epoch e_t earlier than e_i that the anti-malware notifies as the first epoch when tampering attacks began. Therefore, it contradicts to our assumption on the anti-malware. □

Below, we prove that Rocky can successfully recover from the cloud storage failures while there is no tampering attack.

Theorem 3. Rocky can recover from failures with a coherent block device.

Proof. By contradiction, suppose a Rocky block device, B, becomes not coherent during the failure recovery right after applying a mutation snapshot M_i. That is, $W_1, \ldots, W_{i-1}, W_i$ applied to B is not a consistent totally-ordered write sequence. At Step 2 of IRP, l_2 uploads EBSS that contains a consistent totally-ordered write sequence upto e_2 as a new CBSS to the cloud storage. Moreover, at Step 3 of FRP, endpoints upload mutation snapshots which collectively contain a consistent totally-ordered write sequence from e_2+1 to an arbitrary e_x. B downloads the uploaded mutation snapshots so applies a consistent totally-ordered write sequence $W_1, \ldots, W_x$ where $x \geq i$. Hence, $W_1, \ldots, W_{i-1}, W_i$ must be a prefix of $W_1, \ldots, W_x$. However, because $W_1, \ldots, W_x$ is a consistent totally-ordered write sequence, it contradicts our assumption. □

Also, we prove that Rocky can recover the block device that is coherent with minimal data loss even if some endpoints failed along with the cloud storage.

Theorem 4. A recovered Rocky block device is the latest coherent block device recoverable with the given recovery committee.

Proof. By contradiction, suppose the e_x cannot be the maximum epoch in the longest prefix of epochs starting at e_2. However, Step 3 of IRP ensures that every contiguous subsequent epoch starting at e_2 are in L, contradicting our assumption. □