CLOUDS OF GLASS:
FRAGILITY OF VIRTUAL MACHINE DISK IMAGES

BY

PUSKAR NAHA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Associate Professor Steven S. Lumetta

ABSTRACT

With virtual machines becoming common, it becomes important for them to be robust to errors. However, the virtual hard disk images used by virtual machines are not as reliable as physical disks. Existing solutions such as RAID can be used to improve reliability, but only at the cost of disk space usage and inconvenience.

Using the virtual machine software QEMU, we show that its qcow2 virtual hard disk format is vulnerable to failures. We then introduce and analyze two alternative solutions to RAID1 (mirror) that use less disk space and are more convenient to use. As a baseline, the qcow2 format is the easiest to use, uses the least disk space, and has no runtime performance slowdown, but is vulnerable to corruption and block failures. Our first solution, qcow2r, protects qcow2 metadata from corruption and block failures, and it is designed to be only slightly more inconvenient to use. Also, qcow2r uses only 1.003 times the disk space and slows down run time performance by less than 1.5 times. Our second solution, qcow2c, protects all metadata and data in a qcow2 image from corruption and block failures. It is also designed to be as convenient to use as qcow2r. Using qcow2c requires 2.04 times the disk space and slows down run time performance by less than 2 times.

In comparison, RAID1 is more inconvenient than our solutions, because it occupies at least half the provided disk space. It protects against block failures, but not corruption; thus, RAID1 is between qcow2r and qcow2c in terms of vulnerability. Also, RAID1 slows down run time performance by less than 1.5 times, which is comparable to qcow2r, but better than qcow2c. While our solutions target average users, they may be useful to enterprise cloud system operators as well.

To my soulmate Lissa Orstrom for being with me, supporting me, and encouraging me throughout graduate school.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

B	Byte(s)
ECC	Error Correcting Codes
FS	File System
GB	Gigabyte(s) (1024 MB)
GHz	Gigahertz
HDD	Hard Drive Disk
kB	Kilobyte(s) (1024 B)
LVM	Logical Volume Manager
MB	Megabyte(s) (1024 kB)
OS	Operating System
RAID	Redundant Arrays of Inexpensive Disks
Refblock	Reference count block
Refcount	Reference count
Reftable	Reference count table
SSD	Solid State Drive
Symlink	Symbolic link
USB	Universal Serial Bus
VHD	Virtual Hard Disk
VM	Virtual Machine

CHAPTER 1

INTRODUCTION

Virtual machines (VMs) are being used more frequently in both home and enterprise systems. Most home users implicitly assume that VMs are no more vulnerable than physical machines. However, this thesis shows how the virtual hard disk (VHD) image formats used by VMs generally are more vulnerable to block failures and corruption. While there are many existing methods of ensuring file integrity, ensuring that VHDs are not vulnerable to faults has impacts on both the performance of the VM and on the space usage of the VHD image.

We explore this vulnerability through the use of two fault models: corruption and block failures. Corruption is a silent undetected error where the data has been unintentionally altered. A block failure, on the other hand, is when an error is detected and a return value is used to indicate the error. While there are many ways for faults to occur in a hard drive disk (HDD), we limit our fault models to corruption [1] and block failures [2], because users and developers both implicitly assume these models. We limit the number of block failures to one block for simplicity and we limit the amount of corruption to a single byte to demonstrate how fragile VHD formats really are.

In most widely used VHD formats, should the metadata for a VHD become corrupt, the VM considers the entire VHD corrupt and makes no attempt at recovery. Almost no major VM provider's VHD formats provide redundancy for metadata. For example, QEMU's qcow2 format does not [3] and neither does virtualbox's VDI format [4] nor VMware's VMDK format [5]. Only Microsoft's VHD and VHDX formats provide two copies of metadata, each protected by a checksum [6, 7]. In this thesis, QEMU 2.0.0 and its qcow2 format are used as the example to demonstrate the risk of corruption and as a base for suggested improvements. QEMU was chosen due to its ease of use and because it is open source and cross-platform. For the remainder of this thesis, we use the terms VM and QEMU interchangeably.

In order to compare our solutions with existing ones, we use four metrics: convenience to users, vulnerability (or lack of reliability or robustness), disk space usage of the VHD, and run time performance. Run time performance is estimated by measuring the execution time of applications within the VM, which is running on top of the VHD. We show that our solutions are more convenient, have comparable disk space usage, and have comparable run time performance.

We consider redundant arrays of inexpensive disks (RAID) [8] as an existing solution for decreasing the vulnerability of HDDs. However, the average VM user may not be able or willing to set up and maintain a RAID environment themselves. Thus, we present RAID as an existing, but inconvenient, method of decreasing the vulnerability of VHDs stored on HDDs. RAID protects against block failures [9] by keeping copies of data. Inconveniently though, all levels of RAID require multiple HDDs. Even the simplest form of RAID requires two HDDs. As we are focusing on solutions for average users, we only consider RAID1, because it only requires two HDDs [8]. However, RAID1 does not protect against corruption [8,10]. We refer to this two HDD RAID1 as raid.

Considering how common laptops, netbooks, and tablets are, a two HDD solution is potentially not possible either. Instead, as an alternative that is more convenient, we also consider a single HDD partitioned in half with a RAID1 setup across the partitions. We refer to this single-partitioned HDD RAID1 as raidp. Figure 1.1 shows that raid and raidp are roughly equivalent in terms of vulnerability, but they are fairly inconvenient due to disk usage and setup overhead.

This thesis introduces two alternative solutions based on adding checksums, timestamps, and duplication to the qcow2 format, which are less vulnerable to corruption and block failures. Our first solution, qcow2r, is a qcow2 VHD with its metadata duplicated and protected by checksums. It has negligible impact on disk space usage and on run time. Our second solution, qcow2c, duplicates an entire qcow2 image and protects every part of the image with checksums. Qcow2c is even less vulnerable to corruption and block failures, but at the cost of higher disk space usage and slower run time performance. Both of our solutions are preferable to using RAID in terms of disk space usage and convenience. Figure 1.1 compares qcow2 against our two solutions and the two RAID solutions. Since qcow2r only protects qcow2 metadata, it is more vulnerable than raid or raidp. On the other hand, since qcow2c protects every block from corruption and from block failure, it is less vulnerable than raid or raidp. Since raid and raidp protect against block failures, but not corruption, they are between qcow2r and qcow2c in terms of vulnerability. The raid solution is the most inconvenient since it requires two whole HDDs. In contrast, raidp is less inconvenient than raid since it only requires one HDD, but typically consumes half of that HDD, which is less convenient than qcow2r or qcow2c.

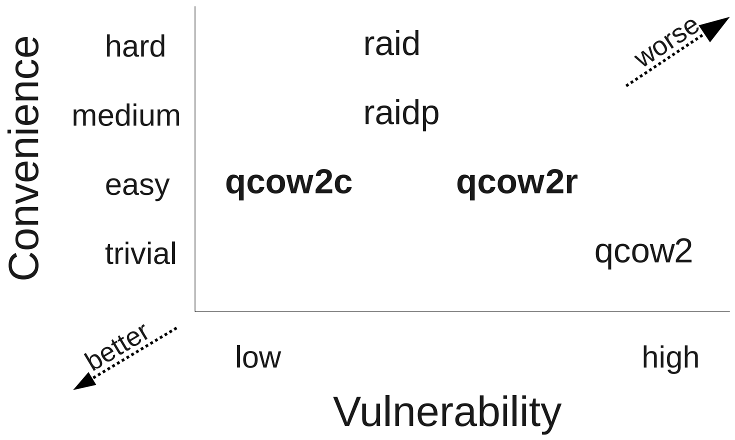


Figure 1.1: Comparison of solutions' vulnerability/convenience with our solutions in bold.

The rest of this thesis is organized as follows: First, Chapter 2 gives background information about VMs and the storage layers within and underneath VMs and illustrates how the VM abstraction creates additional vulnerability. It also gives detailed information about the qcow2 VHD image format used by QEMU. Chapter 3 explains the methodology used in our experiments. It includes the fault model used, how fault injection experiments were run, and the methodology used in measuring performance. Next, Chapter 4 provides the experimental results of fault injections into a qcow2 image. Chapter 5 describes each of the proposed solutions to make qcow2 images less vulnerable to errors. Chapter 6 compares them against the qcow2 image as well as the qcow2 image on RAID. Finally, Chapter 7 concludes this thesis and outlines future work along with directions in which our work can continue.

CHAPTER 2

BACKGROUND

To understand how errors within VHDs are caused and how those errors can cause issues in a VM, we review the necessary background information. First, we describe the VM environment that we are considering at a high level in Section 2.1. Next, we describe each of the layers of the storage system used by VMs to highlight which parts are vulnerable and how faults at lower layers can manifest within the VM. In Section 2.3 and Section 2.4, we go into further detail about the qcow2 format for the VHD layer and ext4 for the FS layers, because we focus on those layers in this thesis.

2.1 High-level Overview

Consider an operating system (OS) running inside a VM, which in turn is running on top of another OS, which is running natively on conventional hardware. The OS running underneath the VM is the host OS, while the OS inside the VM is referred to as the guest OS. The host OS runs the VM like any other application, and the guest OS can then run any application. The VM software we are using is QEMU. The guest and host OSs are both identical versions of Debian Linux for simplicity. Further details of the hardware and software used can be found in Section 3.4.

VMs are becoming more common due to their many advantages. The guest OS does not require information of whether or not it is running inside a VM. The VM emulates a physical machine, so from the guest OS's point of view, it is running on a physical machine. Thus, without any kind of software modification or porting, VMs can be used to run software on different OSs, test software on different architectures, and ensure software compatibility across heterogeneous physical machines. Furthermore, VMs are easy to use, even for average users.

Unfortunately, the VHDs used by VMs are more vulnerable than HDDs, because a single corruption or block failure can render the entire VHD unusable by the VM. As previously explained, most major VHD formats do not protect their VHD metadata from faults. However, VHD metadata is also invisible to higher layers, so higher layers cannot prevent faults in VHD metadata.

2.2 Detailed Overview

While Section 2.1 offers a simplistic view of the overall hierarchy involved with VMs, the storage system used by VMs has more layers and each layer imposes its own set of abstractions on top of the layer below. These abstractions prevent higher layers from having visibility into metadata of lower layers. However, this lack of visibility also prevents higher layers from being robust to faults in lower layers' metadata. Figure 2.1 shows a more detailed view of these layers that are subsequently described.

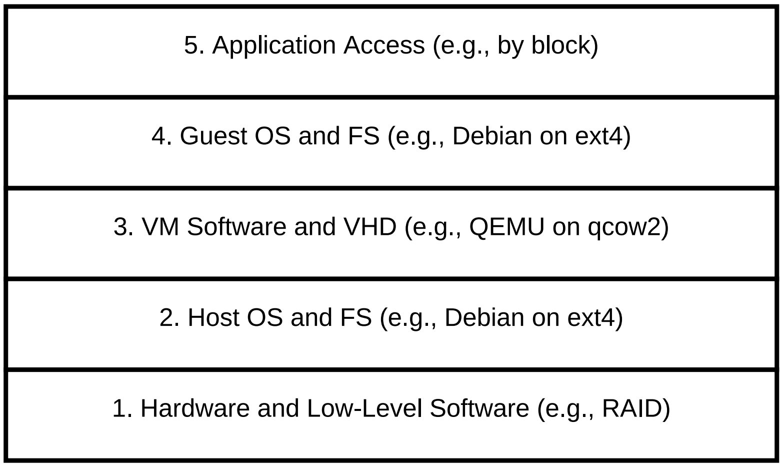

5. Application Access (e.g., by block)
4. Guest OS and FS (e.g., Debian on ext4)
3. VM Software and VHD (e.g., QEMU on qcow2)
2. Host OS and FS (e.g., Debian on ext4)
1. Hardware and Low-Level Software (e.g., RAID)

Figure 2.1: Storage layers.

1. The hardware and low-level software layer includes the physical disk drive, the drive's firmware and controller, and volume management. We are not investigating these components, so we do not consider them independent layers. The physical media component is the actual disk drive such as magnetic HDDs and solid-state drives (SSDs). Varying implementation details in disks may be interesting (re-location on

SSDs or heavier usage of inner tracks on HDDs); however, the causes and mitigations of errors on disks are understood fairly well [2,8,11–14]. The firmware component includes extensions such as RAID, which have been researched extensively, and error correction codes (ECC), which are added to mitigate errors that occur in the physical media component [11]. For example, since the flash memory used in SSDs has a higher error rate than HDDs, SSD controllers use ECC extensively to decrease the vulnerability of SSDs [12, 13]. Volume management is implemented as the Logical Volume Manager (LVM) in Linux. Implemented in software, volume management primarily deals with and manages partitions.

2. The number of choices of an OS and of a file system (FS) for the host are large due to the number of OSs and FSs and versions among them. This layer may introduce encryption, compression, relocation or several other optimizations or complications. FSs are broken down into blocks which are then broken down into sectors. These tend to match the block/sector size of the underlying disk. Most FSs are organized with a boot block first, then a super block or metadata block that gives information about the structure and/or organization of the FS. The super block is usually followed by index nodes (inodes) that represent files or directories which are followed by data blocks that store actual file data. Inodes that represent directories hold file names and those files' inode numbers whereas inodes that represent files point to data blocks for that file. The FS may also organize the layout into groups of blocks to increase spatial locality. Generally, this layer protects its metadata from faults, because upper layers can do nothing in the case that FS metadata is corrupted or suffers from block failure.

3. The VM and VHD image layer is fairly complex, since the VM acts both as a machine for the guest OS and as an application for the host OS, while the VHD acts both as the physical drive for the guest FS and as a file on the host FS. This duality is one of the reasons this thesis focuses on this layer. Also, this layer can possibly add encryption or compression and possibly undermine redundancy in upper layers by storing redundant blocks in a single block in the VHD image. VHD images generally reduce the overall size of the image by only storing

blocks that are in use rather than preallocating unused blocks. Tracking which blocks are or are not allocated is one of the pieces of metadata the VHD image stores. Like the FS layers, VHD images store many other pieces of metadata that give information about the structure and/or organization of the VHD image. Upper layers also can do nothing in the case that the metadata of the VHD is corrupted or suffers from a block failure. Thus, VHD images should include mechanisms to reduce their vulnerability, but most VHD formats do not.

4. The guest FS, like the host FS, has a large number of possible implementation. In this thesis, we select ext4 for both, but they do not have to be identical. All of the concerns and complexities for the host FS can be repeated here. Additionally, the guest FS is fully stored within the VHD in the layer below. Faults in metadata at the VHD layer, can render the entire VHD invalid, which renders any robustness in the guest FS useless.

5. The application layer pertains to how the user or other programs interact with and use files. The way files are accessed can increase or decrease the number of faults. For instance, most files are accessed serially, byte-by-byte or block-by-block. In this case, it is possible that any block that is corrupted would cause the application to be unable to use the remainder of the file. However, it is also possible for applications to access data non-serially or even recover from simple faults by reconstructing or inferring information from other blocks. For example, many files have a magic constant in the first few bytes that dictate the file type. If those bytes were corrupted, an application could use the file extension, file name or the data itself to prevent a fault. As mentioned before though, upper layers cannot recover from metadata faults in lower layers. It is possible for VMs to be run efficiently from inside another VM [15]; however, nesting VMs is outside the scope of this thesis.

Many of these layers may introduce encryption, de-duplication, compression or partitions. In general, any of these features can increase the risk of corruption; however, for the purposes of this thesis, we are not considering any of these features.

2.3 Qcow2

The qcow2 format [3] is a relatively simple format and has only a few types of metadata. A summary of the metadata organization is shown in Figure 2.2. Like most VHDs, FSs, and HDDs, the qcow2 format is organized into a series of fixed-size blocks, called clusters in qcow2, but denoted blocks in this thesis for consistency with common terminology. The size of a block can vary from 512 B to 2 MB and each block is made up of sectors of 512 B.

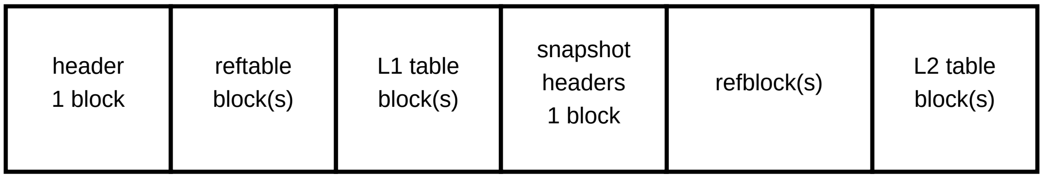


Figure 2.2: Qcow2 metadata. Aside from the header, other types can be located anywhere in the qcow2 image. The first four types shown are each contiguous.

2.3.1 Qcow2 Header

The first block in a qcow2 image, the header, contains information about the image. Other portions of a qcow2 image have variable length, so their lengths and starting locations are stored in the header. The header contains a pointer to the first block of a contiguous series of blocks that make up the first level (L1) table. The header also contains a pointer to the first block of a contiguous series of blocks that make up the reference count table (reftable). The lengths of both tables are also stored in the header. Table 2.1 lists the fields in the qcow2 header and specifies which ones are in use in our qcow2 images. Several pieces of metadata are not in use in our images, because we are not using a backing file, snapshots, encryption, or extensions. We show that faults in the header can render the entire qcow2 image useless to QEMU.

2.3.2 L1/L2 Tables

The L1 table is an array of 8 B pointers to second level (L2) tables and L2 tables are each an array of 8 B pointers to actual data blocks. L2 tables are

Table 2.1: Qcow2 header fields.

Field Name (byte #s)	Field Size	In Use?
Magic Number (0-3)	4 bytes	Y
Version Number (4-7)	4 bytes	Y
Backing File Offset (8-15)	8 bytes	N
Backing File Size (16-19)	4 bytes	N
Block Size (20-23)	4 bytes	Y
Qcow2 Size (24-31)	8 bytes	Y
Encryption Method (32-35)	4 bytes	N
L1 Table Size (36-39)	4 bytes	Y
L1 Table Offset (40-47)	8 bytes	Y
Reftable Offset (48-55)	8 bytes	Y
# of Reftable Blocks (56-59)	4 bytes	Y
# of Snapshots (60-63)	4 bytes	N
Snapshot Offset (64-71)	8 bytes	N
Header Extensions	8 bytes (each)	N

each the size of a single block, while the length of the L1 table is a variable number of blocks determined by the size of the image. The L1 table must be at least one block in size. These tables are used to convert from VHD block addresses (addresses the guest FS uses as physical block addresses) to offsets into the qcow2 file. Faults in the L1/L2 tables cause access to qcow2 data blocks to be lost. However, ext4's metadata blocks are just data blocks at this level. Thus, a single fault to the L1 or L2 tables can cause the guest FS to lose all its data.

2.3.3 Reference Counts

Reference counts (refcounts) are the mechanism qcow2 images use to keep track of which blocks are in use and which are not. All metadata blocks, including reftables and reference count blocks (refblocks), have refcounts as well. The reftable is an array of 8 B pointers to refblocks, just as the L1 table is an array of pointers to L2 tables. Each refblock, like L2 tables, is the size of a single block. Each refblock holds 2 B refcounts. For our images, each refcount is zero when the block it references is unallocated, and one when it is allocated. Faults in the refcounts prevent proper allocation and deallocation of blocks within the range of the fault. If refcounts are corrupted to lower values, there can be data loss since a block in use can be reallocated. On the other hand, if refcounts are corrupted to higher values, blocks are permanently allocated. Another instance when blocks can be permanently allocated is when a refblock experiences a block failure. QEMU must conservatively assume all refcounts in that failed refblock are allocated. Thus, refcounts are less vulernable to data loss in the event of block failures.

Refcounts can even be regenerated by traversing the L1/L2 tables. This is handled by the utility `qemu-img`. This utility comes with QEMU and manages the creation and modification of VHD images outside of QEMU. However, there is also no automatic protection, because users are required to run `qemu-img` themselves.

2.3.4 Other Metadata

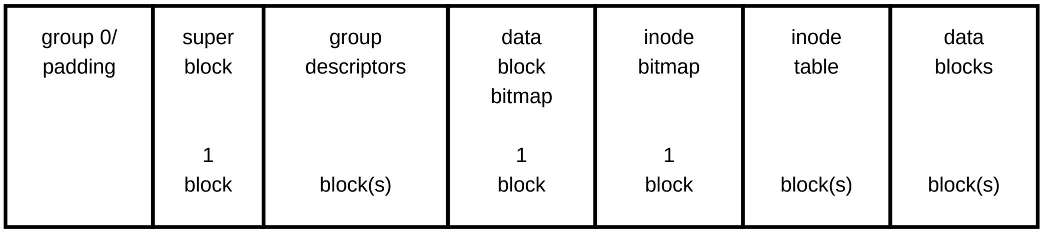
Snapshots and backing files are a level of indirection that are used to store the differences between qcow2 images. Since snapshots are just like regular qcow2 images with some extra metadata (snapshot headers) and backing files are just read-only qcow2 images, we did not investigate either.

2.3.5 Vulnerability

There is no protection of any of the metadata mentioned, so any corruption within the header, L1 table or reftable can corrupt the whole qcow2 image. For example, if the version number (which is stored in the header) is altered to an invalid value, the qcow2 image is unusable by QEMU. As of QEMU 2.0.0, QEMU makes no attempt to correct corrupted fields. It could be possible to recover data from a corrupted qcow2 image, if only the header is corrupted. However, the average user would not want to write their own recovery utilities. A potentially worse issue is that there are no checksums for any metadata. Thus a byzantine error [16] in the header can go unnoticed and cause blocks to be misinterpreted and corrupted, preventing any chance of data recovery.

2.4 Ext4

The FS we focus on in this thesis is ext4[1], which is currently the default Linux FS. Its format and layout are similar to its predecessors, ext2 and ext3. Aside from its being more current, we chose ext4 because its design and implementation cause its performance to be better than ext2 or ext3. Ext4 is a journaled FS like ext3. The journal is used as a temporary record for in-flight writes. For example, in the event of a system crash or loss of power after a write begins, but before a write is committed, the journal has a record that the FS can use to recover. Journals are one of the many mechanisms used outside the VHD layer to decrease vulnerability. Ext4, like its predecessors, is organized into block groups. See Figure 2.3 for the general layout of each block group. Several pieces of metadata in ext4 have checksums for detection of corruption, but not recovery from corruption or block failures.



group 0/ padding	super block	group descriptors	data block bitmap	inode bitmap	inode table	data blocks
	1 block	block(s)	1 block	1 block	block(s)	block(s)

Figure 2.3: Ext4 layout.

2.4.1 Super Block

The first block in ext4 is the super block, which contains FS-level metadata such as the number of blocks, the number of inodes, various flags, etc. It also contains a checksum of its contents. The super block is always replicated. Copies of the super block appear at the start of block group 0 and every block group whose block group number is an integral power of 3, 5, and 7. If the `sparse_super` flag is unset, all block groups begin with a copy of the super block. This replication and inclusion of a checksum protects the super block against corruption and block failure.

[1] https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout

2.4.2 Block Group Descriptors

The super block, if it is present in a given block group, is followed by a block of group descriptors. This block contains similar information to the super block. However the flags and counts stored here pertain to this block group rather than to the entire FS.

Unlike predecessors, ext4 can organize block groups together for the purposes of spatial locality, which improves performance. As this is an optional performance feature, which neither adds redundancy nor increases the chance of data being compromised, we did not consider this feature.

2.4.3 Bitmaps

The next pieces of a block group are the data block and inode bitmaps. Each takes up a single block and is a bitmap that specifies which inodes or data blocks are currently in use. These blocks also have checksums for corruption detection.

2.4.4 Inode Table

The inode table follows and is composed of potentially several blocks depending on the number of files. The inode table is broken down into entries (inodes) which store metadata about files. The inode table is effectively an array that stores a mapping from inode number to the actual inode. Every time a file or directory needs to be accessed, the associated inode is looked up in the inode table. The inode table also includes a checksum for corruption detection.

2.4.5 Directories

Directories are special files called directory entries (dentry) which store a mapping from file name to inode number. Originally, this was stored in a linear array, where each array took up an entire block. Ext3 introduced directories which can contain hash trees to map from file name to inode. The root of this tree occupies an entire block regardless of the number of files in the directory. Dentries do not have a checksum, so corruption of dentries is undetectable by ext4.

2.4.6 Files

The inode for a file contains either pointers to data blocks or extents, a performance optimization introduced in ext4. When storing pointers, the first 12 are direct pointers to the data blocks, the 13th is an indirect pointer, the 14th is a doubly-indirect pointer, and the 15th is a triply-indirect pointer. When storing extents, the inode contains the starting block number and the number of contiguous data blocks used by the file, instead of storing each data block number the file uses. Extents themselves can take up multiple blocks and form a tree structure so that multiple contiguous regions of data blocks are possible. Extents do include a checksum, so extents decrease the vulnerability of ext4 when compared with predecessors. However, indirect blocks do not include a checksum, so corruption of those blocks is undetectable.

2.5 Accessing a File

To understand how the layers interact and how errors can percolate upwards from one layer to the next, we walk through an example of an application inside a VM accessing a file inside a VHD. This example is summarized in Figure 2.4. As shown in Figure 2.4, accessing a file in a VM requires meta-information on the guest FS level, the VHD level, and the host FS levels. There is also the potential for multiple directory entries and symbolic links (symlinks) on the guest and/or host FS level. While blocks are shown in a logical order on each level in Figure 2.4, there is no guarantee they are stored in that order. Furthermore, while blocks may be stored in a certain order within one level, there is no guarantee that they are on a subsequent level. The reftable and refblocks are missing from the qcow2 layer because they are not accessed on reads; QEMU only uses them during operations that cause blocks to be allocated or deallocated. Super blocks are shown in gray, because in ext4, they hold a checksum and are replicated to be less vulnerable to errors. Other blocks in ext4 are not shown in gray, because while many of them do have checksums, they are not replicated. Blocks are shown to be the same size, because block size usually correlates between levels.

To access a file in a VM, an application executes an open and a read system call to a file in the VHD. In order for the application to read a single block from that file, there are many pieces of metadata that need to be accessed first. Given a cold cache, the FS code first locates and loads two blocks, the super block and group descriptors block, in order to know the organization of the FS. It then finds the inode for the root ('/') in the inode table. Since the root is a directory, the FS code looks through the root's dentry to find the file or subdirectory in question. File names are stored in the dentries rather than with the file or the file's inode. The dentry points to the inode for the subdirectory of the file in question. Given a subdirectory, this access pattern continues until the inode for the file is found. When the file is a symlink, it stores the path to another file, so the directory access pattern begins again with the new path. This pattern of accessing symlinks continues until the final non-symlink file's inode number is found. That inode is again looked up in the inode table. The inode contains (either indirectly or directly) pointers to data blocks which are read and returned.

The number of subdirectories and the number of symlinks that need to be accessed is variable. We use the term *directory depth* to represent the variable number of directories and subdirectories being accessed to reach a file. For example, given a file whose path is /top/middle/last/file, the *directory depth* is four (/, top, middle, last) and each of those directories require their inodes and data blocks to be accessed. Furthermore, the number of symlinks accessed to reach a file is also variable and each symlink has its own variable *directory depth*, so we use the term *access depth* to represent the number of directories and symlinks accessed before getting to the final file's inode. For example, given a series of symlinks, /a/symlink1 → /b/symlink2 → /c/file, the *access depth* is eight (/, a, symlink1, /, b, symlink2, /, c). To keep our analysis simple, we assume that each symlink and directory is the size of a single block, though in reality any of them can be larger. Thus, the *access depth* for a file is at most the sum of the *directory depths* of each symlink, the *directory depth* of the final file, and the number of symlinks.

Thus, the total number of block accesses at the physical disk level is $M + N + G + H + 12$. The *access depth* of the qcow2 file determines N and the *access depth* of the guest file (the file inside the qcow2 image) likewise determines M. Since each directory and file has an inode that needs to be found in the inode tables, the number of inode table blocks that need to be accessed depends on the layout of inode numbers being accessed. G and H are at most $M + 1$ and $N + 1$ (when each inode is in a different block) and they are at least one (when all the inodes being accessed are within one block).

While the metadata expansion shown in Figure 2.4 is not required on every single read thanks to caching at multiple levels, it illustrates that higher layers have no access or insight into lower layers' metadata. Furthermore, it shows how corruption of metadata or inability to access metadata can percolate upward.

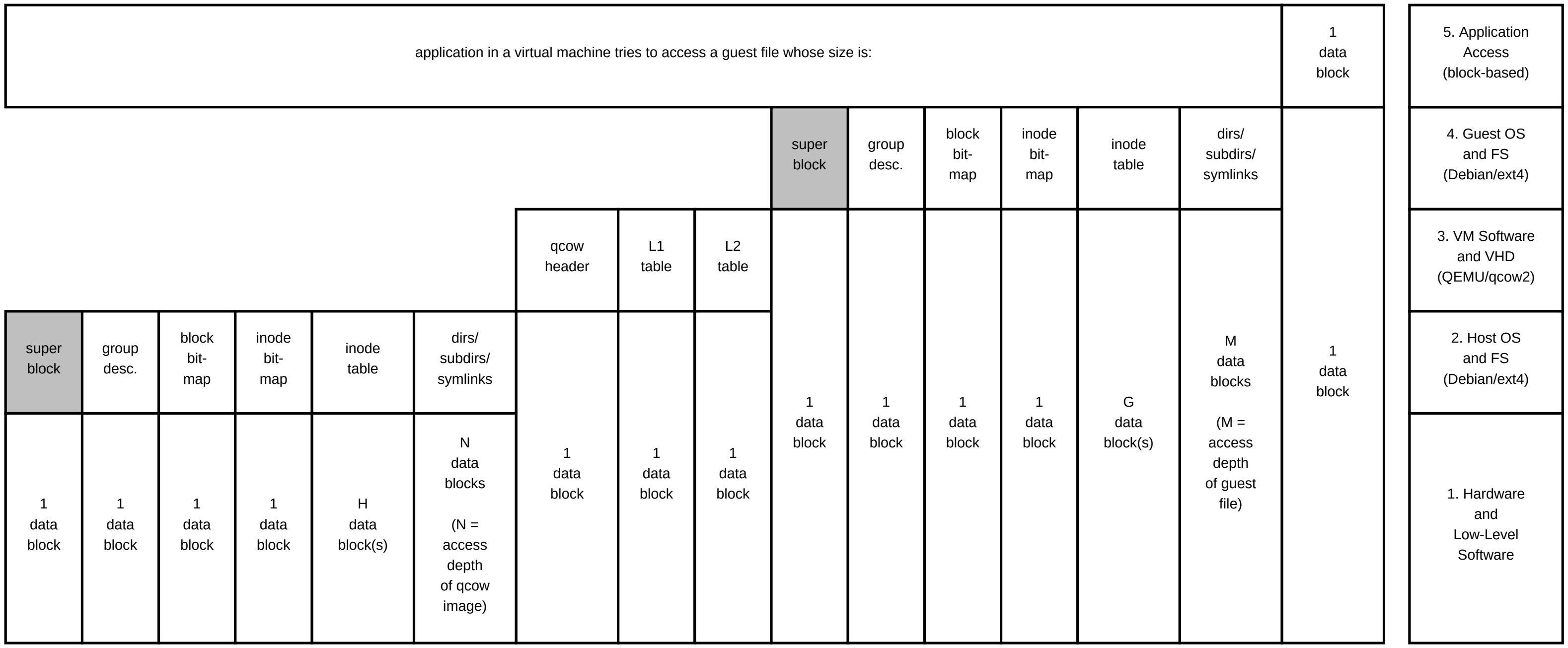


Figure 2.4: All metadata accesses necessary to access a single file in a VM. The total number is $M + N + G + H + 12$. Blocks in gray are less vulnerable.

CHAPTER 3

METHODOLOGY

In order to demonstrate the vulnerabilities of qcow2 images and our solutions' ability to reduce those vulnerabilities, we apply fault injection experiments. We also measure performance, which we use in comparisons among solutions. As stated before, our fault models are corruption and block failures, described in detail in Section 3.1. Our fault injection experiments are described in Section 3.2, and our performance experiments are described in Section 3.3. Our physical setup is described in Section 3.4.

3.1 Fault Model

We focus on errors that manifest within the VM and thus occur at or below the VM layer. Our fault models are based on two different ways that HDD errors can manifest, as corruption and as block failures. We also focus on read failures rather than write failures, since they are more common [10].

Our fault models are not meant to model reality perfectly, but to imitate faults that the average user should be wary of. While enterprise cloud providers have many spare HDDs available when HDDs exhibit any faults, the average user cannot use that strategy. Thus, our simple fault models encompass issues that concern the average user, demonstrate fragility in VHD formats, and encourage a VHD-level solution without having many extra HDDs.

Our fault models do not include total HDD failure for two reasons. First, we are focusing on solutions for the average user, who may or may not have multiple HDDs. Trivially, it can be said that with only one HDD, there is vulnerability of total HDD failure. Furthermore, using or not using a VM does not affect the chances of total HDD failure. Second, existing literature already investigates total HDD failure and its potential causes. The failure, and fear of failure, of HDDs causes disks to be replaced more often than any other computer components [17]. Errors that are perceived as disk failures can be caused by software, firmware, or even the cables used to connect the disk [18]. HDD manufacturers claim the percentage of disks which fail per year to be less than 1% [17], while the average rate of failure found by independent studies is about 3% [17] to 4% [19]. Some studies have even reported rates up to 6% [18] and to 24% [17].

3.1.1 Corruption

Our corruption fault model assumes that any byte in a VHD image may be corrupted. Since the qcow2 format is highly vulnerable to corruption, we limit the amount of corruption in our fault model to a single byte. We limit the amount of corruption to such a small amount in order to have a more fine-grained understanding of how vulnerable parts of the qcow2 format are. Thus, we not only demonstrate that even a single byte's corruption can cause extreme effects on qcow2 images, but we also demonstrate that several bytes can invalidate the entire image.

The two forms of corruption we use experimentally are that one byte is either zeroed or has all of its bits flipped. Zeroing specific bytes is comparable to silent failures at lower layers, where an initial value (zero) is returned. Flipping all the bits in a byte is a generic way of corrupting a byte, without looking at its specific value or purpose. Again, these corruptions are injected with no indication that an error has occurred, by return value or otherwise.

3.1.2 Block Failures

Our block failure fault model assumes that an error is detected by the VM or that an underlying layer detects corruption and returns an error to the VM. Given a failed block, adjacent blocks have a higher rate of failure than other blocks [1, 2, 10]. Furthermore, a HDD with a block failure is more likely to have further block failures [2]. In other literature, this kind of fault is known as a latent sector error and 3.45% of HDDs develop these kinds of errors with over 19.7 errors per HDD on average [2]. For experimental feasibility, we limit the number of block failures in our fault model to one. Limiting the model to a single block failure is analogous to the block failure being the first one detected.

3.2 Fault Injection Methodology

We created a script to partially automate the process of fault injection. First, the script formats the HDD to hold the qcow2 image. The HDD is only formatted once at the start of the fault injection experiments. Afterward, in a loop, the script copies the qcow2 image onto the HDD, injects a fault, and attempts to boot the qcow2 image with QEMU. To prevent QEMU's default behavior of rebooting after a fatal error, QEMU is run with the options `-boot reboot-timeout=0 -no-reboot`. The qcow2 image being copied has its `.bashrc` file set up to automatically create a file with a unique string, then runs `sudo halt`. The last part of the script searches through the qcow2 image for the unique string to validate whether or not QEMU was able to successfully boot the qcow2 image. When the string is found, the VM is successful, i.e., unaffected by the fault injection experiment. When the string is not found, the VM is unsuccessful, i.e., affected by the fault injection experiment. Finally, the reason the script is not fully automated is that QEMU can reach an unrecoverable state, in which case it must be closed manually. This type of situation is not common. For example, QEMU may not detect the error, but the guest OS's FS code neither works properly nor forces the VM to shutdown.

3.2.1 Corruption

Since VHD images can be quite large and many bytes may not be used, we focus our corruption experiments on the qcow2 header. The size of a qcow2 header is 72 bytes when there are no extensions. Even though qcow2 VHDs do not have extensions by default, our experiments corrupt each of the first 100 bytes to test the first few extensions. We run two sets of corruption experiments. See Section 4.1 for the experiment where each byte is zeroed, and Section 4.2 for the experiment where each byte is XORed with 0xFF in order to flip each bit in the byte.

3.2.2 Block Failures

Even with a small qcow2 image, injecting faults into every single block is infeasible. However, the VHD layer is not increasing the risk of faults in data blocks. If data blocks are unprotected at the host FS layer, they remain unprotected at the VHD layer. Injecting faults into data blocks in the qcow2 would test the guest FS's vulnerability, but does not gain any insights into the qcow2 format. Instead, we target qcow2 metadata blocks: the header, the L1 table, the L2 tables, the reftable, and the refblocks. When these metadata blocks have faults, they risk the entire image becoming unusable. The base qcow2 image being used in each run is 1309 MB in size and has 885 metadata blocks that we are injecting. Block failures are simulated in QEMU by injecting faults in handle_aiocb_rw_linear() in block/raw-posix.c. Faults are injected by returning -1 when the block being requested is equal to the block we are trying to fail. To partially automate testing, we also added a command-line parameter to QEMU which determines which block is unreadable.

There are two different binary observances per block failure experiment run. First, whether the VM is successful or not, and, second, whether the block failure is triggered or not. If the block failure is triggered, QEMU tries to access the block, but cannot. If the block failure is not triggered, QEMU never tries to access the block. Of the four permutations of these observances, one is not possible. When the block failure is untriggered, the VM cannot fail. See Section 4.3 for the experimental results of injecting block failures for the other three permutations.

3.3 Performance Methodology

In order to compare our solutions with qcow2 images running with and without RAID, we have a series of performance experiments. Our performance experiments involve measuring the run times of a series of tests within one VM run. Before running these experiments, the physical machine is restarted to prevent any effects from uptime affecting our metric gathering. On every run, the HDDs that store the VHDs are reformatted, unlike our fault injection experiments. Also, a new copy of the qcow2 image is made to improve consistency between runs and to prevent caching or preallocation from interfering. The qcow2 image being copied is set up to automatically run the series of seven tests and `sudo halt`. The set of tests is run at least twenty times, and slow outliers are removed. The run times are measured with the Linux utility `time`. The average run time for each test across all runs is calculated with a 95% confidence interval. The confidence interval is calculated with the formula $1.96 * \frac{\sigma}{\sqrt{N-1}}$. σ is the standard deviation of the test's results and N is the number test results.

The performance tests we are running are grouped into three categories based on whether their time is spent mostly CPU-bound, read-bound, or write-bound. The tests are as follows:

- CPU-bound: run time is dominated by time spent in computation.

  tar: `time` to un-`tar` QEMU 2.0.0's source code

  configure: `time` to run `configure` on QEMU 2.0.0's source code

  make: `time` to run `make` on QEMU 2.0.0's source code

- Read-bound: few writes, with more time spent reading from disk than in computation.

  boot: VM boot time (measured from the time QEMU is run until QEMU can boot and record the final time)

  find: `time` to run `find / -type f`

- Write-bound: few reads, little computation, and spend most of their run time creating single files.

  create_1gb: `time` to create a 1 GB file

  create_5gb: `time` to create a 5 GB file

QEMU's source code is compressed as a bzip2 file, so the *tar* test does some I/O, but is CPU-bound, just like the *configure* and *make* tests. On the other hand, the *boot* and *find* tests are I/O-bound, doing mostly reads. Our heaviest I/O tests, *create_1gb* and *create_5gb* are I/O-intensive, but do mostly writes. Tests *tar*, *configure*, and *find* only take a couple seconds to run, while the *make* test takes several minutes to run. The *create_1gb* and *create_5gb* tests are commands that run a short C program, which is compiled with `gcc` without optimization enabled. This program writes 1 GB or 5 GB of ASCII ones to a newly created file. It writes these ones 4 kB at a time in order to match the block size used throughout our setup. These last two tests are entirely I/O-bound and expose the overhead of using RAID, qcow2r, or qcow2c versus qcow2. The reason we have two of these tests is, in our VHD, the 1 GB file does not cause any qcow2 metadata tables to grow while the 5 GB file does require the reftable to grow at least once.

3.4 Testbed

The physical machine being used to run these tests is an HP Z220 Workstation with a quad-core 3.2 GHz Intel®Core™ i5-3470 CPU with 8 GB of memory. The host and guest OSs are both Debian amd 6.0.9 (squeeze) with Linux kernel version 2.6.32.

The version of QEMU being used is 2.0.0 and the architecture is x86_64. The block size being used throughout each layer is 4 kB. QEMU is run with a 9 GB qcow2 image, 2 GB of memory, and KVM enabled. On the physical machine, the QEMU executable is stored on the OS HDD, while the VHDs being operated on and being booted are stored on their own HDDs. The host OS's FS is XFS, while the HDDs that store the VHDs are ext4. The guest OS's FS is also ext4. Both the ext4 FSs have barriers disabled to enable faster write speeds. The OS HDD is a Western Digital Caviar with 500 GB, 7200 rpm, and SATA interface, while the first and second HDDs are Seagate Barracudas[1] with 250 GB, 7200 rpm, and SATA interface. For a summary of the HDDs being used and in which types of experiments they are used, see Table 3.1.

3.4.1 RAID1

The existing solution we are comparing qcow2r and qcow2c with is RAID1, since it does not require a minimum of three or more HDDs. RAID1 is merely a mirror where a minimum of two HDDs are kept in sync in an array. Remember, we refer to this traditional RAID1 setup across two similar HDDs as raid. The RAID1 used here is based on the `mdadm` Linux utility which implements RAID in software rather than hardware, which is the more common use case for the average user. The version of `mdadm` being used is

[1] model numbers ST3250310AS, ST3250318AS

3.1.4. With mdadm, writes to the array are duplicated onto each HDD and reads from the array are optimized to come from whichever HDD is faster or whichever HDD has not failed [9]. In the event of a HDD failure, the failed HDD is removed from the array and another fresh HDD needs to be rebuilt. While this rebuilding is pending or in progress, the array is effectively reduced to the vulnerability of a single HDD.

As mentioned before, the average user may not want to or be capable of having multiple HDDs or may not be willing to have yet another HDD available in the case of a failure. Thus, our raidp solution is more convenient by setting up a RAID1 across two partitions of a single HDD. An advantage of RAID1 that we are not considering is that raidp can tolerate a partition failure and raid can tolerate a HDD failure. While there are versions of RAID that use more than two HDDs and provide protection against corruption, that is outside the scope of this thesis.

3.4.2 External HDD

Our external HDD setup moves the first HDD from being connected internally through SATA to being connected through a SATA-to-USB 2.0 board. The board is a Cedar 3.5 USB Rev 1.4 which has 128 kB of flash memory. While an external HDD would never be used in enterprise clouds, it is a reasonable mechanism for an average user to have an additional HDD. Table 3.1 summarizes our HDD usage.

Table 3.1: HDD usage.

	non-RAID	raidp	raid
OS HDD: 500 GB	Host OS, QEMU	Host OS, QEMU	Host OS, QEMU
First HDD: 250 GB (internal & external)	VHD	RAID-protected VHD	RAID-protected VHD
Second HDD: 250 GB (internal only)	not used	not used	RAID-protected VHD

CHAPTER 4

FAULT-INJECTION EXPERIMENTS & RESULTS

This section reports and discusses the results of our fault-injection experiments on the qcow2 format. These experiments show how vulnerable the qcow2 format is. These experiments are not a comprehensive set of tests for the qcow2 format. Instead, they apply our simple fault models of corruption and block failure to demonstrate the qcow2 format's vulnerability. These experiments are partially automated (see Section 3.2). Our three fault-injection experiments are:

1. Zeroing a byte in the qcow2 image's header.
2. Flipping all the bits in a byte in the qcow2 image's header.
3. Failing a metadata block in the qcow2 image.

Remember that a run is considered successful when the VM is able to boot and write a unique string to a guest file inside the qcow2 image. When the string cannot be found, the run is considered unsuccessful. In each run, a fresh auto-`halt`ing qcow2 image is copied and booted.

4.1 Qcow2 Header: Zeroing Bytes

As mentioned before, the size of the qcow2 header is 72 bytes. Header extensions are already zero in our qcow2 image. There are twelve specific bytes, which, when zeroed, prevent the VM from running successfully. Table 4.1 below summarizes which fields are affected.

The qcow2 image being tested only has a single byte per field that is not set to zero usually, which is why only one byte out of four or eight for each field in Table 4.1 causes issues. Most of the values in the qcow2 header are smaller than 256 and thus fit within a single byte. Since there are a lot of zeros in the qcow2 header, our next experiment flips all the bits in one byte to expose more vulnerabilities.

Table 4.1: Results of corrupting single bytes in qcow2 header.

Field Name (byte #s)	Field Size (B)	bytes corruptible by	
		zeroing	flipping
Magic Number (0-3)	4	4	4
Version Number (4-7)	4	1	4
Backing File Offset (8-15)	8	0	7
Backing File Size (16-19)	4	0	0
Block Size (20-23)	4	1	4
Qcow2 Size (24-31)	8	2	8
Encryption Method (32-35)	4	0	4
L1 Table Size (36-39)	4	1	3
L1 Table Offset (40-47)	8	1	8
Reftable Offset (48-55)	8	1	8
# of Reftable Blocks (56-59)	4	1	3
# of Snapshots (60-63)	4	0	4
Snapshot Offset (64-71)	8	0	3
First Extension (72-79)	8	0	3
Other Extensions (80-99)	8 bytes (each)	0	3

4.2 Qcow2 Header: XORing Bytes

In this experiment, we XORed individual bytes in the header with 0xFF, causing every bit in the byte to be flipped. We then found that even more fields were susceptible to corruption that can prevent the VM from booting. Table 4.1 states which fields, when one of their bytes is XORed, prevents the VM from being able to boot.

In Table 4.1, the *Backing File Offset*, *L1 Table Size*, and *# of Reftable Blocks* fields were partially resilient to this type of an error, because corruption to the lowest byte did not significantly alter the overall value and is undetected.

The *Snapshot Offset* field is interesting. First, the highest byte prevents booting since the resulting value is too large for QEMU. Second, the lowest two bytes also prevent booting, because snapshots must be stored block-aligned. The middle bytes can be corrupted, which shows that this field is not used by QEMU to boot the OS, but is checked for validity on startup.

Furthermore, corruption of three of the bytes in the *First Extension* prevents booting. Extensions have an ID for the first 4 bytes and then a size for the second 4 bytes. QEMU only checks for extensions until the ID is zero, meaning QEMU read at most the first two extensions in this experiment. However, when we corrupt the upper bytes of the size of the first header, it is too large for QEMU and prevents booting. Corruption of the size of subsequent headers does not have an effect, because QEMU does not read past the first header extension with an ID of zero.

Several of these corruptions prevent booting; however, several can be avoided by improving input validation. For example, corruptions to header extensions should not prevent booting, since VMs that use header extensions need their own validation that the extensions are correct. Unnecessary header extension validation is more of a bug than a demonstration of the qcow2 image's vulnerability, so we suggest correcting it by having QEMU print a warning message and continue to boot. Likewise, corruption to the *# of Snapshots* should not matter when *Snapshot Offset* is zero and vice-versa. The same goes for *Backing File Offset* and *Backing File Size*. In each of these cases, QEMU can also simply print a warning message and continue to boot. It is worth noting that QEMU 2.0.0 already has code for managing qcow3. The qcow3 code has even more input validation of header extensions and thus the majority of our corruptions to header extensions would prevent QEMU from booting in the future.

It is also worth noting that in previous versions of QEMU (for example, 1.6.2 and earlier), several of these corruptions cause QEMU to crash or even segfault when starting up due to a lack of input validation by the qcow2 driver. However, error-checking and error-handling are significantly improved in subsequent versions. The lack of input validation in previous versions has the side effect of allowing certain corruptions to go by undetected, such as the *L1 Table Size* being too large. This corruption does not prevent QEMU from booting, because the *L1 Table Size* is shifted up and used as a number of bytes. Shifting overflows some of the bytes of *L1 Table Size* and thus the field is unintentionally padded in prior versions. The relatively recent addition of input validation to QEMU demonstrates that as QEMU's development continues, it continues to improve its handling of qcow2 image corruption, implying that qcow2 image vulnerability is a concern for the future.

4.3 Block Failures

In this experiment, we inject individual block failures into QEMU, to simulate errors percolating upward from layers below the VHD layer. We then boot the VM, which automatically writes a unique string to a new guest file inside the qcow2 image and halts. Remember that a run is considered successful when the unique string is found inside the qcow2 image. Otherwise, the run is considered unsuccessful. Recall that we only inject faults into qcow2 metadata blocks, because data blocks are unprotected in the host FS, regardless of whether or not it stores a VHD image. Figure 4.1 summarizes the results.

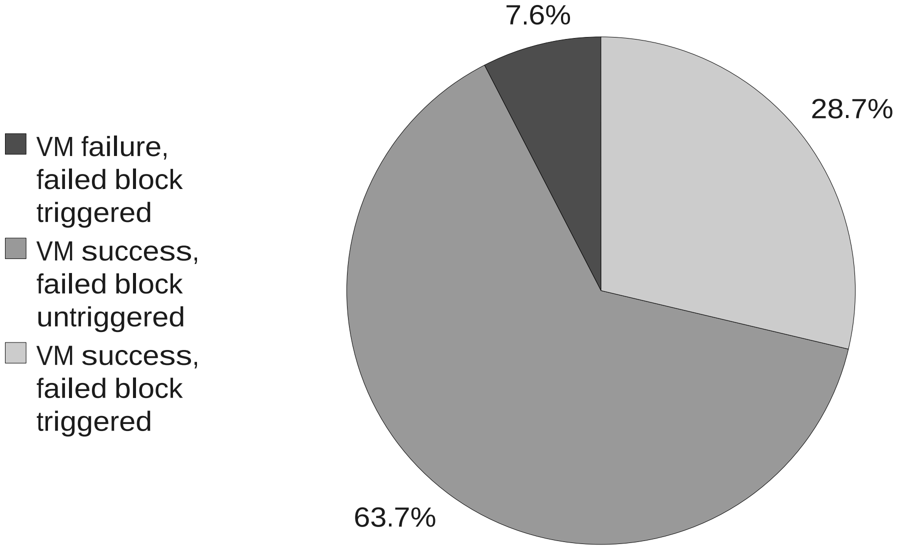


Figure 4.1: Results of injecting block failures.

In 7.6% of the runs, the block being failed is triggered and the VM is unsuccessful. These runs include every run that affects the L1 table, reftable, and header. They also include several of the L2 tables. These results corroborate our expectations.

In 63.7% of the runs, the block being failed is untriggered during the VM run and the VM is also successful. This is also expected behavior since any VHD or FS is tolerant to failures in blocks that are unused. Only L2 table blocks appear in this category.

Interestingly, in 28.7% of the runs, the block being failed is triggered, but the VM is unsuccessful. Expectedly, these runs include all refblock failure injections. Remember that when QEMU encounters a failed refblock, it conservatively treats all refcounts in that failed refblock as allocated. Thus, this experiment confirms that QEMU tolerates refblock failures. These runs also include some L2 tables, which imply that some non-critical files or utilities are read from the qcow2 image when the VM boots. It is worth noting that in several of these runs, there are many on-screen error messages generated by ext4. In our experiment, the VM only has to write to a file after booting to be considered successful. Since the error messages are ext4-specific, we do not investigate them in this thesis.

While the probability is low that one of the metadata blocks in use by QEMU fails, this experiment shows how devastating such a simple fault can be for the qcow2 format. Subsequently, we present our qcow2r solution to correct the faults seen in our fault injection experiments.

CHAPTER 5

PROPOSED SOLUTIONS

In this chapter, we introduce our qcow2r and qcow2c solutions. Our solutions' strategy is based on the well-known technique of adding checksums to protect data [1,2,10,11]. Our proposed solutions require adding checksums and last-updated timestamps to the qcow2 VHD image format as well as duplicating the checksum-protected data. The data is duplicated to create a backup in case an error occurs or corruption is detected. The timestamps are necessary in case one of the copies is updated, but the other copy is not, due to some kind of fault. In such a case, both copies have correct checksums, but only the more recent checksum is correct. Our implementations have 32-bit checksums and 32-bit timestamps. The checksum is calculated simply by interpreting data as a series of big-endian signed 32-bit integers, summing them together, and flipping the bits of the result. This simple checksum is used for its ease of implementation. The final sum is flipped as a cautionary guard against a series of zeros having a checksum of zero. Since our timestamps are only 32-bits in length, they will overflow in the year 2106. For our purposes, and the purposes of QEMU, this is a minor limitation.

When QEMU needs to read any of the protected data, it loads both the first copy and the duplicate copy and checks their checksums. The copy with the newest timestamp whose checksum is also correct is the one which should be read. When QEMU needs to write any of the protected data, it calculates the checksum and writes back the first copy of data and the first checksum and next it writes back the second copy of data and the second checksum. Again, this ordering is necessary in case a fault occurs in the middle of the write back.

One of the design choices of QEMU for their in-development qcow3 format is to make sure the qcow3 format is similar enough to the qcow2 format, so that the qcow2 driver can be extended to support qcow3[1]. We also keep that

[1] http://wiki.qemu.org/Features/Qcow3#Requirements

design choice in mind and ensure that our solutions are backwards compatible with the qcow2 format.

5.1 Qcow2r Format

Our first proposed solution is qcow2r: a qcow2 image with robust metadata. With qcow2r, all metadata is protected via the checksum/timestamp/-duplication strategy. This metadata is composed of the header, L1 table, reftable, and each L2 table and refblock. Since our fault model assumes up to a single block may fail or be corrupted, the duplicate of each block or table of metadata is stored immediately after its respective original. Storing the duplicates with their originals ensures that qcow2r is backwards compatible with the qcow2 driver. Since the L1 table, L2 tables, reftable, and refblocks are stored via offsets, the qcow2 driver does not know the duplicates are there. As mentioned before, it is known that when a block fails, adjacent blocks have a higher rate of failure than other blocks [1, 10]. Thus, the best method of storing duplicate metadata blocks is to have one copy at the beginning and a second copy at the end of the file, which is the mechanism provided by Microsoft's VHD and VHDX [6, 7]. However, separate duplicates of metadata cannot work for qcow2 images, while maintaining backwards compatibility. The qcow2 block allocator would need to be modified to allocate disparate rather than contiguous blocks. Furthermore, since qcow2 images' metadata includes several offsets, each pointer would need to be duplicated. Thus, the qcow2 format would need to modified, preventing any backwards compatibility.

As with the other metadata, the duplicate header is stored immediately after the original header. However, the block size is stored within the header, so QEMU cannot reliably read the block size before it confirms that the header is not corrupt. Since QEMU cannot know the block size, the duplicate header must be at a fixed offset. For qcow2r, this offset is 4 kB since it matches our block size. In the future, to make this format usable with any block size, the duplicate header would need to be stored at a fixed offset of the maximum block size for QEMU (at the moment that is 64 kB). Furthermore, since the minimum size of the header is 512 bytes, the qcow2r header's checksum is stored at the end of the 512 bytes. While having the checksum at a fixed offset limits the number of header extensions, it ensures backwards compatibility and works with all block sizes.

The other metadata's checksums are not quite as simple to store, because all the other tables and blocks take up whole blocks. Thus, we are not able to store the checksums with their respective tables and blocks without breaking backwards compatibility between the qcow2r and qcow2 formats. To solve this problem, we add tables of checksums that also take up whole blocks. To store checksums for each L2 table, we add an L1-sized table, *L2 checksums*. For example, in order to access the L2 table corresponding to the third entry of the L1 table, the third entry of the *L2 checksums* table would need to be accessed to ensure the L2 table has not been corrupted. Since we are adding the *L2 checksums* table to the metadata, we need to make it less vulnerable by duplicating it too. Likewise, we need to add a reftable-sized table, *Refblock checksums*, which is also duplicated and stores the checksums for each refblock. To avoid adding more offsets to the qcow2 header, the *L2 checksums* table and its duplicate are stored immediately after the L1 table and its duplicate, and the *Refblock checksums* table and its duplicate are stored immediately after the reftable and its duplicate. For the checksums of the *L2 checksums* table, the *Refblock checksums* table, the L1 table, and the reftable, we add another block after the header blocks, a *Checksums* block. Finally, this block is also duplicated. Figure 5.1 shows the layout of our qcow2r format's metadata. Remember that the header must be the first two blocks and each table or block must immediately precede its duplicate. However, order between any pieces of metadata is not guaranteed.

To estimate the space overhead of qcow2r, we analyze each piece of metadata in a qcow2 image, because the space overhead of qcow2r is equivalent to the metadata overhead of a qcow2 image. For each block in a qcow2 image, there is an L2 table entry of 8 B, so the L2 tables take up approximately $\lceil\dfrac{8\ B}{4\ kB}\rceil \approx 0.002$ of the image. For each L2 table, there is an L1 table entry of 8 B, so the L1 table takes up approximately $\lceil\frac{8\ B*\frac{8\ B}{4\ kB}}{4\ kB}\rceil \approx 0.000004$. For every block in the qcow2 image, there is a refblock entry of 2 B, so the refblocks take up approximately $\lceil\dfrac{2\ B}{4\ kB}\rceil \approx 0.0005$. For each refblock, there is a reftable entry of 8 B, so the reftable takes up approximately $\lceil\frac{8\ B*\frac{2\ B}{4\ kB}}{4\ kB}\rceil \approx 0.000001$. The header only takes up a single block, so its effect on the metadata overhead is negligible. Like the header, the *Checksums* block, has negligible impact on

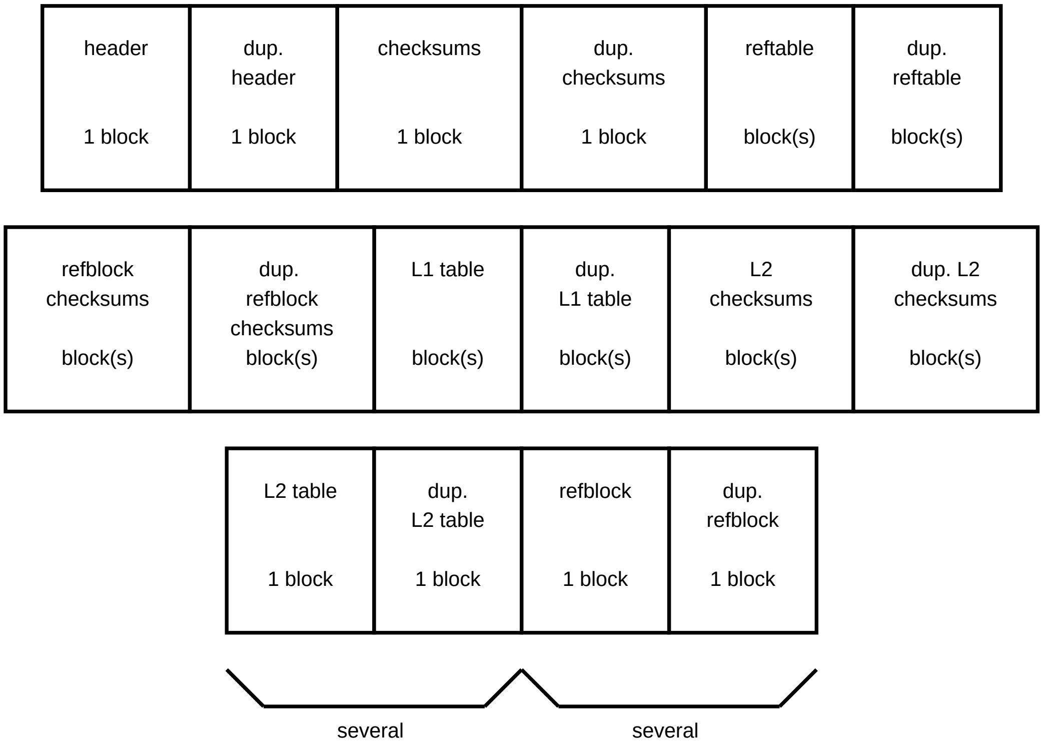


Figure 5.1: Qcow2r metadata.

the space overhead for qcow2r. Rounding up, the total estimated metadata overhead for a qcow2 image is $\lceil 0.002+0.000004+0.0005+0.000001 \rceil \approx 0.003$. 0.003 is also the space overhead for a qcow2r image, since it duplicates all the qcow2 metadata.

5.2 Qcow2c Format

In the interest of finding a solution that is both more convenient and less vulnerable than RAID (see Figure 1.1), we have also created a solution called qcow2c. With qcow2c, every sector is protected via the checksum/-timestamp/duplication strategy. Qcow2c keeps a copy of the qcow2 image and two copies of a file with checksums and timestamps for each sector in the qcow2 image. This solution is thus not a different file format *per se*, but several separate files being used in unison. A benefit of qcow2c's implementation is the qcow2 file, and its copy, are unmodified versions of qcow2 files, thus this implementation is fully backwards compatible.

Qcow2c offers significantly less vulnerability than qcow2r offers. Qcow2r only protects qcow2 metadata and relies on higher or lower layers to protect data blocks. However, as seen in Section 2.2, most of the ext4 FS is not resilient to block failures. Thus, to ensure qcow2c can handle block failures, there are two qcow2 image copies and each copy is kept in sync. Furthermore, there is a file of checksums and timestamps for each sector of the qcow2 image. Again, this file is kept in duplicate, so when either file becomes corrupt or suffers block failures, QEMU can recover. Once more, timestamps are included in case one copy is updated but the other is not. While this solution is actually simpler than qcow2r and easier to implement, it is more expensive in terms of performance. What is a single block update with qcow2, becomes a 4-way update with qcow2c (original qcow2 image, duplicate qcow2 image, qcow2 checksum file, duplicate qcow2 checksum file). Another penalty of this implementation choice is that these four reads or writes are done sequentially. Performance can be improved by reading and returning the first copy if no problem is detected with it. However, that approach misses errors in which only one copy is updated prior to a system crash. As a result, this optimization is not implemented. In Figure 1.1, this optimization would be in-between qcow2r and qcow2c in terms of vulnerability.

Our qcow2c solution has checksums per sector rather than per block due to QEMU's implementation. QEMU's lowest level read/write functions operate sector by sector, so it required the least number of changes to the source code to add checksums to each sector rather than each block. A benefit of this implementation choice is that it makes this solution potentially both backwards and forwards compatible with any format QEMU uses, since sector size is fixed at 512 B for QEMU while block size varies by VHD image. Our current implementation works for the qcow2 format and the in-development qcow3 format, but is untested and may require some additional changes for other formats.

Another benefit of our qcow2c implementation is that it stores its copies in separate files. By storing them separately, qcow2c prevents copies from being adjacent to each other. Thus if one block in one file fails, it is more likely that subsequent blocks in the same file will fail, rather than blocks in the duplicate file.

The space overhead of qcow2c is simple to calculate. First, each sector has a 4 B checksum and a 4 B timestamp. Next, there are two copies of both the qcow2 file and the checksum/timestamp file. Thus, the space overhead of qcow2c is $2 * (1 + \frac{8B}{512B}) = 2.03125$.

5.3 Implementation

Our implementation of qcow2r and qcow2c began by creating programs to convert a qcow2 image into a qcow2r or qcow2c image. Thus, using either of our solutions would be easy for any user, by using our conversion tools. Next, the converted images were run with an unmodified version of QEMU 2.0.0, to ensure they maintained backwards compatibility. After QEMU was modified, the images had corruption and block failures injected, to ensure they were less vulnerable than the qcow2 format. Further details can be found subsequently in Section 5.4.

First, qcow2r was implemented. Its implementation started with a qcow2-to-qcow2r generator. Each piece of metadata was replicated with a checksum. After each step of development of the generator, the in-progress qcow2r image was booted by an unmodified version of QEMU 2.0.0 to ensure backwards compatibility. Finally, QEMU 2.0.0 was modified to support qcow2r images instead of qcow2 images.

Second, qcow2c was implemented. Its implementation was much simpler and much quicker than qcow2r's implementation. Again, implementation started with a qcow2-to-qcow2c checksum generator. The generator utility creates two identical files with the checksums for every sector in the original qcow2 image. The generator utility also creates a copy of the original qcow2 image. Finally, QEMU 2.0.0 was modified to support qcow2c images instead of qcow2 images.

5.4 Validation

Our solutions were validated by injecting faults into both. For qcow2r, every duplicate metadata block can be corrupted, but the VM is successful. Likewise, the VM is successful when every original metadata block is corrupted. For qcow2c, either the entire duplicate qcow2 file can be corrupted or the entire original qcow2 file can be corrupted. In either case, the VM is successful.

Finally, individual block failures can be injected in several metadata blocks in qcow2r and in several blocks (metadata or data) in qcow2c. In either case, the VMs are successful.

CHAPTER 6

COMPARISON OF SOLUTIONS

In order to demonstrate how our new drivers/formats (qcow2r and qcow2c) perform, we compare them not only against the original qcow2 image, but with the original qcow2 image running on a RAID. Furthermore, we compare the qcow2 image running on two identical HDDs that are in a RAID together (raid) as well as on a single HDD with two partitions that are in a RAID together (raidp). Again, raidp is presented as a more convenient alternative to raid. See Section 3.4 for more details on the physical system.

6.1 Convenience Comparison

Our first comparison metric is convenience to the user. This comparison is made of two parts: ease of use and the minimum number of HDDs needed. Qcow2 is the easiest to use, since it requires no change by users or developers, while qcow2r and qcow2c are the next easiest to use since after making the decision to use them, they are as easy to use as the qcow2 format. Also, qcow2r and qcow2c do not require whole HDDs to be reformatted, and thus are more convenient than raid or raidp. Both raid and raidp are of comparable difficulty to use, since failure recovery with either requires the failed HDD or partition to be rebuilt. Finally, raidp is still more convenient than raid, since raid requires that a user purchase and utilize a second HDD.

6.2 Vulnerability Comparison

Our second comparison metric is vulnerability, or the lack of reliability or robustness. As Chapter 4 shows, qcow2 images are highly vulnerable to corruption and block failures. Qcow2c has the least vulnerability with respect to corruption and block failures, while qcow2r has the highest vulnerability of the solutions being presented. Qcow2r is meant to be a step better than qcow2 in terms of vulnerability by protecting qcow2 metadata, but leaves protection of actual data blocks to higher or lower layers. Raidp and raid have medium vulnerability, because they are not vulnerable to block failures, but are vulnerable to corruption.

A comparison of the various solutions in terms of inconvenience and vulnerability is shown in Figure 6.1, which is a duplicate of Figure 1.1. The lower the value in the chart, the better the solution is for that metric. Overall, qcow2c has the lowest vulnerability and is easy to use, since it does not require a whole HDD. Raidp and qcow2r are the next best: raidp has less vulnerability at the cost of being less convenient, while qcow2r is more convenient at the cost of vulnerability. Qcow2 and raid are the worst, because raid inconveniently requires multiple HDDs and qcow2 is highly vulnerable.

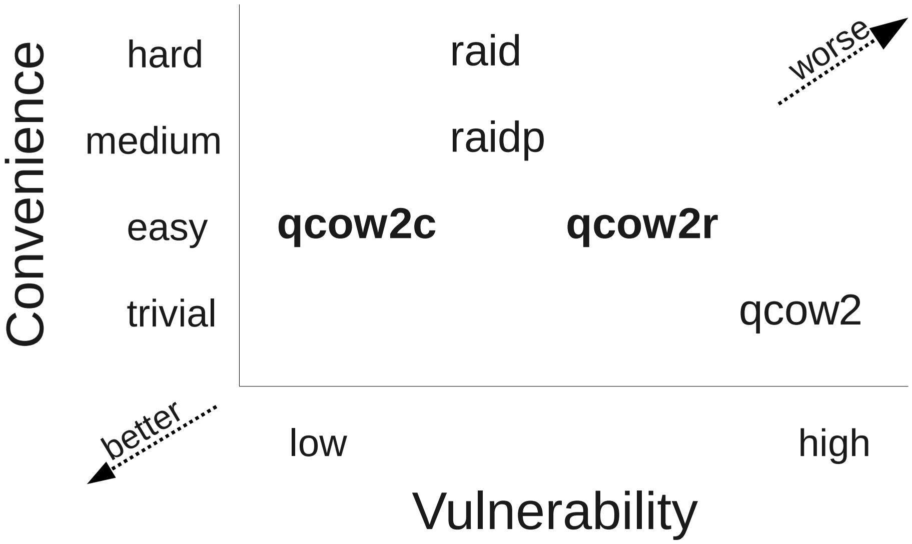


Figure 6.1: Comparison of solutions' vulnerability/convenience with our solutions in bold.

6.3 Performance Comparison

Our next metric is performance and we compare the runtime performance on internal HDDs of qcow2r, qcow2c, and qcow2 image on RAID (raid and raidp) relative to the same qcow2 image on non-RAID. Refer to Section 3.3 for the detailed description of the tests and how they are run. A comparison of the solutions' execution time in various tests is shown in the chart in Figure 6.2. Since the tests vary in runtime from a couple seconds to several minutes, the chart is normalized based on qcow2's performance. The actual run times can be found in Table 6.1. As expected, qcow2r generally performs better than qcow2c, and raid generally outperforms raidp. Our results are slower, but similar when the host and guest FSs are both ext3 instead of ext4.

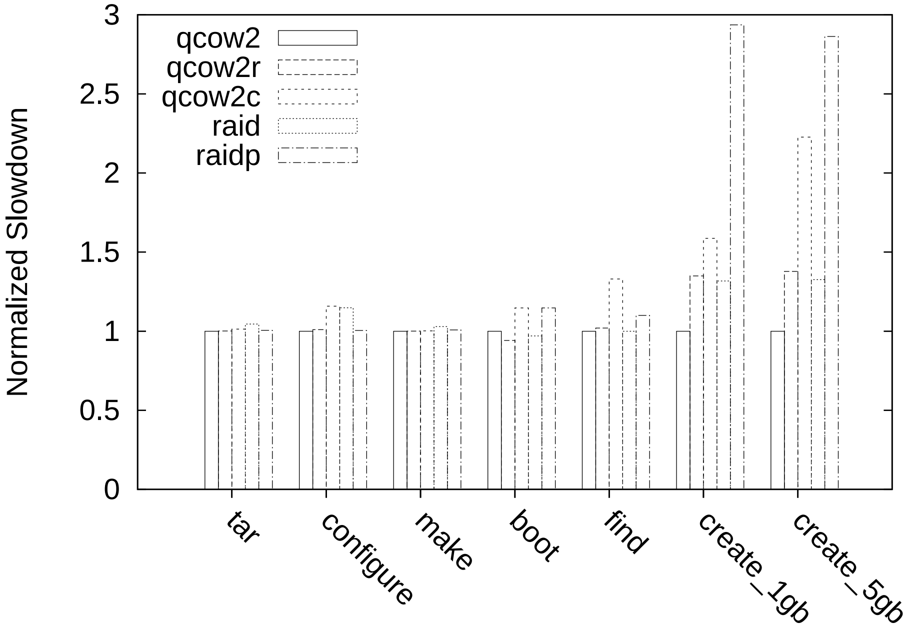


Figure 6.2: Comparison of solutions' execution time normalized with qcow2's results as a baseline.

In the CPU-bound and read-bound categories, raidp is comparable to raid and sometimes even faster than raid. The reason for this unexpected performance boost is probably due to how we partitioned the HDD. Since the HDD is partitioned in half, it is probable that two different heads were able to read from two different partitions simultaneously. Furthermore, both raidp and raid use RAID1, which, on reads, only reads from a single copy. While raidp sometimes being faster than raid is an interesting outcome, it is a RAID-specific outcome and thus is not investigated further in this thesis. For the write-bound tests, raidp is over twice as slow as raid, which is also reasonable

Table 6.1: Comparison of solutions' execution time in seconds with confidence intervals on internal HDDs.

Category	Test	qcow2	qcow2r	qcow2c	raid	raidp
CPU-bound	tar	2.009 ± 0.004	2.012 ± 0.003	2.035 ± 0.005	2.10 ± 0.02	2.02 ± 0.006
	configure	2.09 ± 0.009	2.11 ± 0.02	2.42 ± 0.02	2.4 ± 0.4	2.10 ± 0.01
	make	1234.8 ± 0.5	1235 ± 0.8	1238 ± 0.7	1271 ± 2	1245 ± 2
Read-bound	boot	6.8 ± 0.3	6.4 ± 0.4	7.8 ± 0.2	6.6 ± 0.4	7.8 ± 0.5
	find	1.0 ± 0.08	1.02 ± 0.04	1.33 ± 0.02	1.0 ± 0.06	1.1 ± 0.07
Write-bound	create_1gb	12.6 ± 0.2	17.0 ± 0.2	20 ± 0.8	16.6 ± 0.2	37.0 ± 0.3
	create_5gb	94.3 ± 0.4	130 ± 2	210 ± 5	125 ± 0.6	270 ± 1

since both partitions need to be written to and both partitions were on the same HDD.

Overall, our tests show that qcow2r's performance is comparable to qcow2 and raid across all tests and is comparable to raidp for the CPU-bound category. In fact, the write-bound category is the only category with enough variation to be of interest. Thus, for the average user, qcow2r is a good alternative to protect qcow2 metadata, especially if the user only has access to a single HDD. Even if the user has multiple HDDs, qcow2r is still a reasonable choice since qcow2r is only slightly slower than raid in the write-bound tests.

Overall, our tests also show that qcow2c's performance can be compared with raid and raidp. Generally, in the CPU-bound and read-bound categories, qcow2c is comparable to raidp and worse than raid. In write-bound tests, however, qcow2c also performs worse than raid, but performs better than raidp. Thus, when choosing between qcow2c and raid, there is a performance/vulnerability trade off since qcow2c has lower vulnerability and raid has higher performance.

It is worth noting that, originally, barriers were turned on in ext4 and most results were negligibly slower. However, the write-bound tests for qcow2r were approximately twice as slow as qcow2r on ext3 or ext4 without barriers.

When compared to qcow2, raid, and raidp, these write-bound tests imply that checksum updates in qcow2r cause write barriers to be invoked frequently. When compared to qcow2c, qcow2r's frequent writes to a single file, rather than qcow2c's writes to four files, caused ext4 to use more write barriers.

Our final performance tests were run with the first HDD connected through USB. The exact same tests as above were run, and Table 6.2 provides the run times. Interestingly, the results are comparable to the internal HDD results in all the CPU-bound and read-bound performance tests. The exception is the performance of write-bound tests. The external HDD causes create_1gb and create_5gb to take about twice as long to run as they did on the internal HDD.

Table 6.2: Comparison of solutions' execution time in seconds with confidence intervals on external HDDs.

Category	Test	qcow2	qcow2r	qcow2c	raid	raidp
CPU-bound	tar	2.04 ± 0.008	2.06 ± 0.007	2.04 ± 0.02	2.09 ± 0.02	2.07 ± 0.009
	configure	2.12 ± 0.02	2.15 ± 0.02	2.40 ± 0.02	2 ± 0.6	2.13 ± 0.02
	make	1239 ± 4	1241 ± 2	1246 ± 2	1269 ± 2	1273 ± 4
Read-bound	boot	7.2 ± 0.3	7.4 ± 0.2	7.8 ± 0.2	7.5 ± 0.2	7.2 ± 0.2
	find	1.0 ± 0.06	1.1 ± 0.08	1.39 ± 0.05	1.1 ± 0.08	1.1 ± 0.08
Write-bound	create_1gb	27.7 ± 0.08	35.6 ± 0.2	52 ± 2	30.3 ± 0.2	60.5 ± 0.5
	create_5gb	200.4 ± 0.2	260 ± 0.7	421 ± 0.7	231.1 ± 0.5	476 ± 3

6.4 Disk Usage Comparison

Our last metric is the disk usage of the VHD in each solution. Table 6.3 shows the size of the VHD before running the tests in Section 6.3 and the size of the VHD after. The RAID entries are marked with * since their overhead applies to the entire disk or disks, not just to the VHD.

As expected, qcow2r has negligible space overhead when compared to a

Table 6.3: Comparison of solutions' disk usage.

Metric	qcow2	qcow2r	qcow2c	raidp	raid
Initial Size (MB)	1308	1312	2658	1308	1308
Initial Size (normalized)	1	1.003	2.04	2*	2*
Final Size (MB)	8152	8175	16561	8152	8152
Final Size (normalized)	1	1.003	2.04	2*	2*

qcow2 image, while qcow2c has over double the space overhead since it stores a replica and checksum for every sector. Both raid and raidp are estimated to have twice the overhead since we are using a RAID1 mirror. The fact that raid and raidp's overhead actually applies to entire HDDs is encapsulated in our convenience metric.

Figure 6.3, Figure 6.4, and Figure 6.5 show a comparison of performance and disk space overhead for our CPU-bound, read-bound, and write-bound tests respectively. The space overhead for all three charts is from Table 6.3 and the performance overhead for the charts is calculated as the arithmetic mean of the normalized values from Figure 6.2. While there is no clear 'winner,' it is clear that when disk space is a priority and performance is not, qcow2 and qcow2r are the best options with qcow2r being better than qcow2 in terms of vulnerability. If performance is a priority and disk space is not, any option is good except for qcow2c. In that case, raid or raidp is better in terms of vulnerability while qcow2r is better in terms of convenience. If better performance, lower disk space usage, less vulnerability, and convenience are needed, qcow2r is the best alternative to qcow2 since qcow2r has low performance slowdown, negligible disk space overhead, and provides less vulnerability. Alternately, if better performance, less vulnerability, and I/O-throughput are needed, raid is the best option.

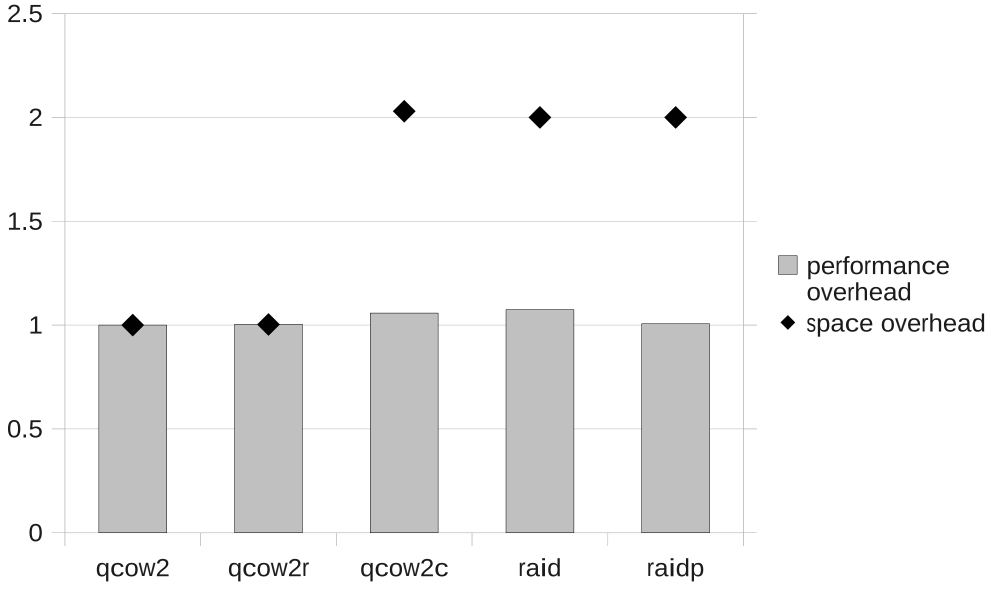


Figure 6.3: Comparison of normalized time/space overhead for CPU-bound tests.

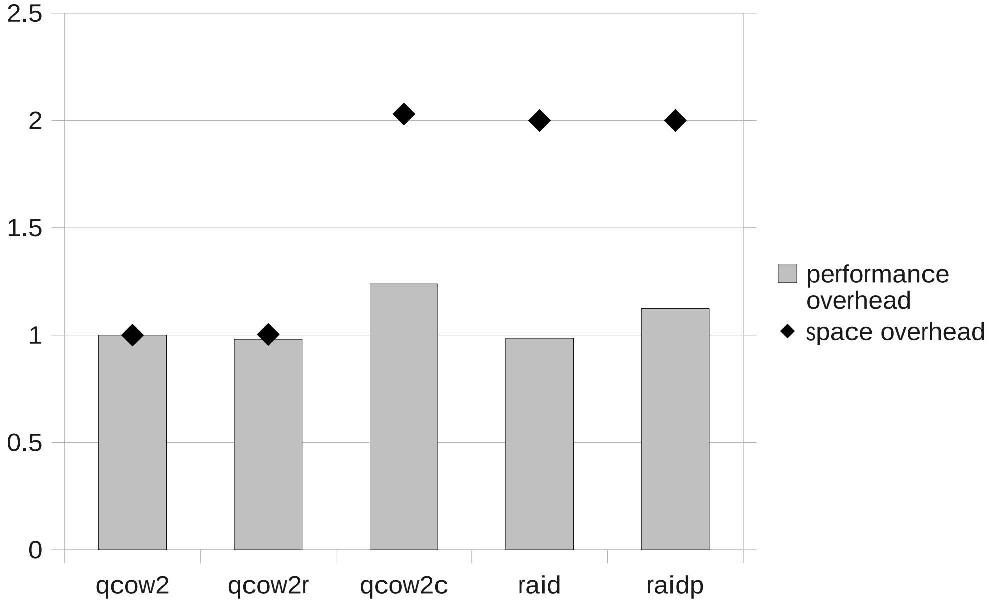


Figure 6.4: Comparison of normalized time/space overhead for read-bound tests.

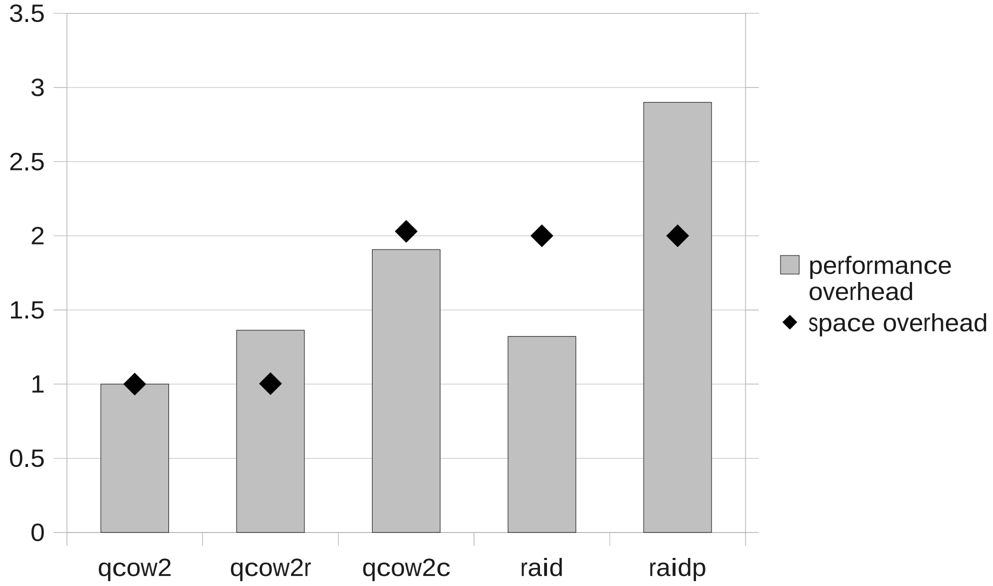


Figure 6.5: Comparison of normalized time/space overhead for write-bound tests.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In conclusion, our checksum-based solutions, qcow2r and qcow2c, are good alternatives to RAID for the average user in terms of performance, vulnerability, space overhead, and convenience. Our corruption experiments show that corruption to individual bytes can render QEMU's qcow2 VHD format unusable, which shows how fragile qcow2 images are. While there are many ways to make files less vulnerable to corruption and block failures, avoiding impact to the VM's performance and the VHD's disk space usage is more difficult. In order to minimize that impact, we suggest qcow2r, which is approximately as robust as a physical HDD. While still vulnerable to data block failures, qcow2r has negligible performance and disk space overhead In order to minimize vulnerability, we suggest qcow2c, which protects every block in the image at the cost of performance during heavy writes and just over double the disk space usage of qcow2. However, qcow2c provides lower vulnerability than RAID can and, overall, uses less disk space than any implementation of RAID. Finally, both qcow2r and qcow2c are more convenient than RAID for the average user.

Future work includes demonstrating similar fragility in the other major VM's VHD formats such as virtualbox's VDI, VMware's VMDK, and Microsoft's VHD/VHDX. Furthermore, it is possible that the same critical values that need to be protected in the VHD formats also need to be protected once read into memory from memory corruption. It would also be interesting to determine whether enabling encryption or compression in a qcow2 image increases the risk of corruption, and if so, by how much. Also, since QEMU supports several VHD formats other than the qcow2 format, it is compelling to extend our qcow2c solution to include them and find out how the performance impact for those formats compares with the qcow2 format.

REFERENCES

[1] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *Transactions on Storage*, vol. 4, no. 3, pp. 8:1–8:28, Nov. 2008. [Online]. Available: http://doi.acm.org/10.1145/1416944.1416947

[2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1254882.1254917 pp. 289–300.

[3] M. McLoughlin, "The QCOW2 image format," Sep. 2008. [Online]. Available: https://people.gnome.org/~markmc/qcow-image-format.html

[4] TerryE (user name), "All about VDIs," July 2008. [Online]. Available: https://forums.virtualbox.org/viewtopic.php?t=8046

[5] "VMware virtual disks: Virtual disk format 1.1," Nov. 2007. [Online]. Available: http://www.vmware.com/app/vmdk/?src=vmdk

[6] "Virtual hard disk image format specification," Oct. 2006. [Online]. Available: http://go.microsoft.com/fwlink/p/?linkid=137171

[7] "VHDX format specification," Aug. 2012. [Online]. Available: www.microsoft.com/en-us/download/details.aspx?id=34750

[8] D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz, "Introduction to redundant arrays of inexpensive disks (RAID)," in *Proceedings of the IEEE COMPCON Spring'89*. IEEE, 1989, pp. 112–117.

[9] T. Niedermeier, "Mdadm checkarray," May 2014. [Online]. Available: http://www.thomas-krenn.com/en/wiki/Mdadm_checkarray

[10] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "IRON file systems," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095830 pp. 206–220.

[11] T. Zhang, G. Mathew, H. Zhong, and R. Micheloni, "Modern hard disk drive systems: Fundamentals and future trends," in *Memory Mass Storage*. Springer, 2011, pp. 169–212.

[12] M. Cornwell, "Anatomy of a solid-state drive," *Communications of the ACM*, vol. 55, no. 12, pp. 59–63, Dec. 2012. [Online]. Available: http://doi.acm.org/10.1145/2380656.2380672

[13] W. Hutsell, J. Bowen, and N. Ekker, "Flash solid-state disk reliability," *Texas Memory Systems White Paper*, 2008. [Online]. Available: http://www.dynamicsolutions.com/assets/pdfs/whitepapers/flash_solid_state_disk_reliablity.pdf

[14] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. Ng, "Disk scrubbing in large archival storage systems," in *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on.* IEEE, 2004, pp. 409–418.

[15] Z. Pan, Q. He, W. Jiang, Y. Chen, and Y. Dong, "NestCloud: Towards practical nested virtualization," in *Cloud and Service Computing (CSC), 2011 International Conference on.* IEEE, 2011, pp. 321–329.

[16] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, July 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[17] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" 2007, pp. 1–16.

[18] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population." 2007, pp. 17–23.

[19] J. Gray and C. Van Ingen, "Empirical measurements of disk failure rates and error rates," *arXiv preprint cs/0701166*, 2007.