



POLITECNICO DI TORINO

Master of Science in Software Engineering

Master Degree Thesis

Development of fuzzing methodologies for testing the resilience of the SATA protocol

Supervisor

prof. Paolo Bernardi

Sebastian Florin SANISLAV

MARCH 2020

Abstract

Communication between computing systems and storage device units, generally called Hard Drives, is one of the most important interfaces for in the IT world, and guaranteeing it's security has high priority. The main protocol in use today is known as the "ATA standard" and the focus of our work during this internship is to test its resilience through Fuzzing, an automated software testing technique that consists in providing unexpected, invalid, random data as input to the target system, and monitor its behaviour in search for anomalies such as bugs, crashes or potential security flaws.

In our work we propose different types of Fuzzing schemes, mainly based on a client-server architecture using the Fuzzing framework "Boofuzz" together with software developed by us. The systems we aim to target are not only physical hard disks but also virtual hard drives, tested under different virtualization softwares such as QEMU, VirtualBox and VMWare.

This work is the continuation of what has been started during a previous semester project, developed by the students Mirabella Angelo and Sanislav Sebastian (author of this work) and supervised by Balzarotti Davide (also supervisor for this internship). Even though during the semester project some results were achieved, their understanding was not clear. In this internship we try to explore more these outcomes and to continue developing the designed methodologies.

Contents

List of Figures	v
List of Tables	vi
1 Introduction	1
2 Background	3
2.1 Hard drive communication standards	3
2.1.1 The ATA protocol	4
2.1.2 ATA Application Layer	6
2.1.3 ATA IDENTIFY command	8
2.1.4 The SCSI protocol	9
2.1.5 ATA_Pass_Through	10
2.2 Virtualization	12
2.2.1 QEMU	14
2.2.2 VirtualBox	14
2.2.3 VMWare	15
3 The fuzzing system	16
3.1 Client-Server fuzzing model	16
3.2 The fuzzing data Generator: Boofuzz	17
3.3 The Fuzzer	18
3.3.1 The Python Fuzzer	19
3.3.2 The C Fuzzer	21
4 Fuzzing Virtual and Real Hard Drives	25
4.1 Qemu experiments	25
4.1.1 Command Tracing Mechanism	25
4.1.2 Code Coverage	26

4.1.3	First set of experiments	28
4.1.4	Investigating the Read-Only problem	29
4.1.5	The ATA SLEEP command	31
4.1.6	Second set of experiments	32
4.2	VirtualBox experiments	35
4.2.1	Experiments on VirtualBox	35
4.2.2	Incompatibilities with ATA_Pass_Through	37
4.3	VMWare experiments	38
4.4	Real Hard Drives	39
4.4.1	Malformed Sectors	39
5	Conclusions	41
	Bibliography	43

List of Figures

2.1	Difference between SATA and SAS drive connectors	4
2.2	Visual representation of the division in layers of the SATA protocol.	5
2.3	Model of the packages transported by SATA Link Layer.	6
2.4	Layout of a Host-to-Device type of Frame Information Structure managed by the SATA Transport Layer.	6
2.5	Fourth layer of the ATA architecture, representation of the registers that compose an ATA command.	7
2.6	Graphical representation of the role of a hypervisor in the virtualization mechanism.	13
3.1	Graphical depiction of the Fuzzing model	16
3.2	Example of the Boofuzz command line interface.	17
4.1	Occurrences of ATA commands in a regular Virtual Machine	26
4.2	Example of the graphical representation of the coverage report . . .	27
4.3	Kernel log of a VM with its file-system set to read-only	29
4.4	Linux kernel code snippet containing the leaf directory entry checks.	30
4.5	Memory leak in a part of the virtual hard drive.	31
4.6	Raw bytes leaked by the IDENTIFY command output.	32
4.7	Screenshot of the "top" process running in a flooded VM.	34
4.8	Data returned by the ATA READ_PIO command using the utility tool "hdparm".	38

List of Tables

2.1	SCSI Generic Structure (sg_io)	10
2.2	ATA_Pass_Through CDB 12 byte version (to the left) and 16 byte version (to the right)	11
2.3	Setup Bytes 1 and 2	11
3.1	List of the 45 ATA commands implemented by the C Fuzzer	23
4.1	Traced ATA commands	26
4.2	Structure of an EXT4 directory entry (ext4_dir_entry)	29
4.3	Structure of a leaf directory entry (ext4_dir_entry_tail)	30

Chapter 1

Introduction

The main idea behind our work is the intention of testing the resilience and security of the Hard Drive communication protocols in use today. This seems to be a novelty in literature, since we were unable to find any other similar works. Another motivation for this work comes from the importance that the interface with the storage devices has. It fulfills a role that is present and fundamental in any computing system and has been in use for almost half a century. The security of these type of interfaces is of extreme importance and securing it through extensive testing is a must.

This internship is aiming at continuing the work done during the semester project developed by the students Mirabella Angelo and Sanislav Sebastian (author of this work) and supervised by Balzarotti Davide (also supervisor for this internship). The main objective is to test the resilience of the ATA standard to fuzzing techniques and try to find bugs or vulnerabilities. During the semester project, we developed a first version of a fuzzing system that we used to perform testing of real and virtual hard drives.

The most accessible way of sending ATA commands that we found during the semester project, was by using the SCSI protocol, and in particular one specific SCSI command named "ATA_PASS_THROUGH". Practically, we are encapsulating an ATA command inside a SCSI command that will get processed and transported as a SCSI command until it gets to the bridge component, in charge of translating the command to an ATA command. This was necessary because the SATA devices are seen, by the Linux kernel, as SCSI devices and, in order to communicate with them from the user level, it is mandatory to use the SCSI protocol. For this reason in our work we don't only study the ATA protocol, but also the SCSI standard.

During the semester project we discovered different anomalous behaviours, both from the physical and from the virtual Hard Drives. Regarding the physical Hard Drives, first of all we found out that connecting it to the PC through a SATA to USB adaptor was not a resilient method, since it happened very often that after sending a few fuzzed commands it would lose connection. This did not happen when connected directly to the motherboard through a SATA cable. Moreover we were able to change the size of one SSD device, bringing it down from 250GB to 2.4GB. This looked like a permanent change since the new memory size was

detected the same by many different PCs. Regarding the virtual Hard Drives, the main anomaly that we found was the fact that, after fuzzing for different amounts of time, the file-system of the virtual machine would become read-only causing the Fuzzer to crash. The reasons behind this behaviour were still unclear.

Our objective for this internship was to continue the development of the techniques used during the semester project and to investigate more the results obtained, trying to give more explanations and to understand if there are potential security flaws that could be lurking under the observed anomalies.

Chapter 2

Background

2.1 Hard drive communication standards

Establishing communication with Storage devices has been one of the main and oldest challenges for designers of computing systems. Even though, in the recent history, the industry of Hard Drive Standards has gone through a chaotic and tumultuous evolution, its culmination in the present age can be summarized by two main defined and standardized protocols: the ATA (Advanced Technology Attachment) protocol and the SCSI (Small Computer System Interface) protocol. The main difference between these two standards is that ATA was first designed as a protocol whose purpose was to enable communication with hard disk drives only, while SCSI was designed as capable of controlling different device types, like mass storage devices, tape drives, removable optical media drives (CD, DVD, Blu-Ray...), scanners and many others.

The variants of these protocols that are in use today are the serial versions, respectively SATA (Serial-ATA) and SAS (Serial Attached SCSI). While SATA drives are mainly used for multipurpose, big scale production for the general public, SAS drives are faster and more reliable, but much more expensive, reason for which they tend to be used for Enterprise Computing such as banking transactions and E-commerce, where high speed and high availability are crucial and essential features.

It is important to note that, even though SATA and SAS are two different protocols defined by two different technologies, they are similar under some aspects. For example, they share the same drive connectors, and they are electrically compatible. Thanks to this, the ATA standard was later extended to provide compatibility with a broader range of devices by carrying SCSI commands and responses through the ATA interface, defining this way the ATAPI (ATA Packet Interface) protocol, mainly used to allow communication with removable optical media drives, like CD-ROMs and DVD-ROMs.

There is also a certain level of interoperability between the serial variants of the two standards in use today. In figure 2.1 we can observe the main difference between a SAS connector and a SATA connector, in particular the fact that the power supply is separated from the data connector in the SATA drive, while they are unified in the SAS drive. Because of this reason, and because SATA connector

signals are a subset of SAS signals, it is possible to connect a SATA drive to a SAS adapter, but SAS drives will not operate on a SATA adapter. Their different design purpose is indeed to prevent any chance of plugging them in incorrectly. In other words, we can state that a SAS controller is capable of "talking" both to SAS and SATA devices, while a SATA-only controller is not able to communicate with a SAS drive.

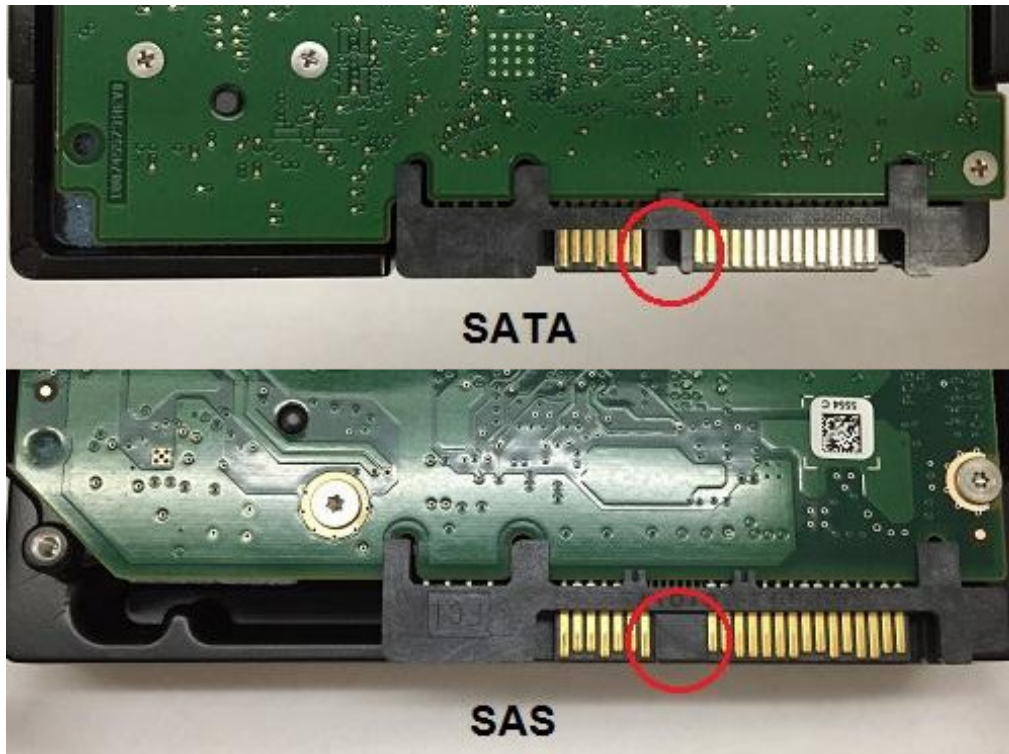


Figure 2.1: Difference between SATA and SAS drive connectors

2.1.1 The ATA protocol

The structure of the ATA protocol can be summarized by a multiple layer model, where each layer has a specific task, operates on the data received by the layer above and passes the processed information to the layer underneath. More specifically it is divided in four layers: physical layer, link layer, transport layer and application layer.

The physical layer specifies the means of transmitting raw bits through the electrical channel. It defines the connectors and cabling used to carry data and signaling information. In particular it defines "out of band" (OOB) signals, which are low-speed signal patterns recognized by the physical layer, that do not appear in normal data streams. They are made of defined amounts of idle time followed by repeated sequences (bursts) of a dword primitive (called ALIGN). The only difference between the OOB signals is the length of the idle time between bursts of ALIGNs. When the SATA device is powered on, the communication is established through OOB signaling followed by the speed negotiations. These are done by having the target device send ALIGN primitives at the fastest supported rate; if

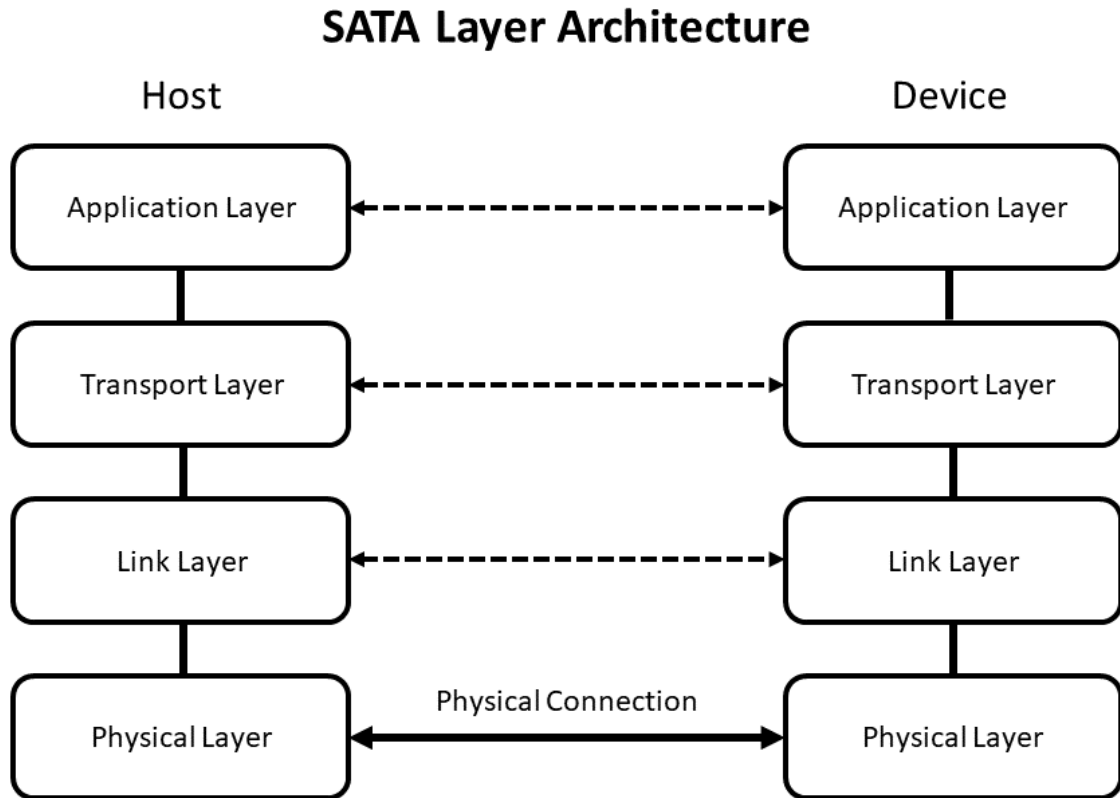


Figure 2.2: Visual representation of the division in layers of the SATA protocol.

the host doesn't reply with ALIGNs then it does not support this speed, so the device steps down to the next lower speed and tries again, until a reply is received.

The link layer has the main task to scramble and descramble data and control words from and to the physical layer and to convert data from the transport layer into structures called "frames". It manages the encoding process, which is based on a 8b/10b encoding scheme. This means that all 32 bits of data are encoded to 40 bits of transmission, mainly to provide a sufficient amount of transition density (guarantees the presence of transitions, 0-1 or 1-0, even when the bytes to be transmitted are 0x00 or 0xff) and to facilitate the detection of transmission errors. The link layer is indeed in charge of building and checking the CRC (an error detecting code) of the transmitted frames. The general structure of a frame is shown in figure 2.3. We can see that the data received from the transport layer (payload) and the CRC are enclosed between two SATA primitives: SOF (start of frame) and EOF (end of frame). These primitives have the fundamental role of delimiting the frame.

The transport layer is responsible for the management of data structures called Frame Information Structures, or FIS. Mainly its task is to build the FIS with the data received from the application layer, initiate the transmission request and pass it to the link layer together with the content of the FIS, receive the transmission status from the link layer and communicate it to the application layer. The FIS content varies based on the operation to be executed. In figure 2.4 an example of a

Sata Link Layer



Figure 2.3: Model of the packages transported by SATA Link Layer.

Host to Device FIS is shown. We can observe that it is composed of different fields, the majority of which are device registers whose values are set by the application layer. Other fields are mostly configuration flags that communicate transmission details to the target device. Apart from the *Host to Device* FIS there are seven more types of FIS defined by the standard.

	byte n+3	byte n+2	byte n+1				byte n
Data dword 0	Features 7:0	Command	c	r	r	r	PM Port
Data dword 1	Device [Dev/Head]	LBA 23:16 [Cyl High]	LBA 15:8 [Cyl Low]				LBA 7:0 [Sector Number]
Data dword 2	Features 15:8	LBA 47:40 [Cyl High (exp)]	LBA 39:32 [Cyl Low (exp)]				LBA 31:24 [Sector # (exp)]
Data dword 3	Control	<i>ICC</i>	Count 15:8 [Sector Cnt (exp)]				Count 7:0 [Sector Count]
Data dword 4	Reserved	Reserved	Reserved				Reserved

Figure 2.4: Layout of a Host-to-Device type of Frame Information Structure managed by the SATA Transport Layer.

At last we have the application layer. This is the most important layer for the purposes of our fuzzing scheme. The ACS (Ata Command Set) is defined at this layer: this is the collection of commands defined by the T13 association that makes it possible for applications to communicate and use the SATA drive functionalities. Every few years this document is revised and renewed, in order to be able to keep up with the technological evolution. At the moment of this writing, the most recent revision is the ACS-8. The commands defined change between revisions, with the possibility of having new commands introduced in the set, or some already existing commands become obsolete or retired.

2.1.2 ATA Application Layer

Before illustrating the ATA commands structure it is necessary to understand the way the data addressing mechanism worked in the past and today. To address data on a spinning disk device it is required to correctly position the mechanical components. In order to access a specific data block, the host needed to specify the cylinder, head and sector position (CHS), that, when written into the device

registers, would move the mechanical components to the specified position. CHS were in reality a logical mapping, allowing large capacity drives to be supported and giving the drive the possibility of performing its own defective sector management. This addressing scheme was the one used by the Original ATA command sets. The evolution of CHS addressing is called Logical Block Addressing (LBA) and it virtualizes the physical geometry of the device by providing a simple method to access fixed sized blocks of data (512 bytes, 4k bytes, etc.). More specifically all data blocks are mapped to a contiguous LBA range, where LBA 0 corresponds to the first block, LBA 1 corresponds to the second block, and so on. In this way only the device knows the CHS corresponding to a specific LBA and is able to translate it, making it significantly easier for the host to address a specific block.

In the original ATA implementations the translation between LBA and CHS was only performed on a 28-bit LBA. This implied that, with a block size of 512 bytes, the maximum addressable drive capacity was 128GB. To solve this problem, ACS-6 introduced the extended 48-bit sector addressing mechanism, creating also a new set of commands that would support this new addressing feature. Today, in the ACS-8, there are several commands with the "EXT" label that are able to address 48-bit LBA, bringing the theoretical addressable capacity up to 144 petabytes.

ATA Registers
Features/Error
Sector Count
LBA Low (Sector Number)
LBA Mid (Cylinder Low)
LBA High (Cylinder High)
Device/Head
Command/Status

Figure 2.5: Fourth layer of the ATA architecture, representation of the registers that compose an ATA command.

The way the application layer SATA commands interact with the hard drive is by setting the values of its internal registers, called ATA or shadow registers, of the SATA device. As shown in figure 2.5, there are mainly seven registers with different functionalities. Generally, the functionality of each register depends on the ATA command to be executed. However, for the majority of cases the purpose of these registers can be summarized as:

- **Features (Error):** This register has different functionalities that depend on the command to be executed. Its standard size is 8-bit but on 48-bit

commands it becomes a 16-bit register. After the completion of the command, this register contains the Error data, where each bit of the register has a specific meaning.

- **Sector Count:** This register indicates the number of blocks to read/write by the ATA command. Its size is 8-bit for the 28-bit commands and 16-bit for the 48-bit.
- **LBA Low/Sector Number:** This register was previously known as "Sector Number" because it used to indicate the sector of the disk. It represents now the low part of the LBA and its size is 8-bit for the 28-bit commands and 16-bit for the 48-bit commands.
- **LBA Mid/Cylinder Low:** This register was previously known as "Cylinder Low" because it used to indicate the low part of the cylinder of the disk. It represents now the low part of the LBA and its size is 8-bit for the 28-bit commands and 16-bit for the 48-bit commands.
- **LBA High/Cylinder High:** This register was previously known as "Cylinder High" because it used to indicate the high part of the cylinder of the disk. It represents now the low part of the LBA and its size is 8-bit for the 28-bit commands and 16-bit for the 48-bit commands.
- **Device/Head:** Before the serial version of the ATA protocol, this register had the important task of indicating which one of the two devices on the ATA bus was the target of the command. Furthermore, the four LSB of the register were used as the four MSB bits of the LBA address. However, with the introduction of SATA and with ATA-8, this functionalities have become obsolete, and many of the bits composing this register are reserved or obsolete. Its size is 8-bit with both 28-bit commands and 48-bit commands.
- **Command (Status):** This register contains the code of the ATA command to be executed, as specified in the ACS. It is a 8-bit register for both 28-bit commands and 48-bit commands. This is the first value checked by the device and depending on its value, the content of the other registers are interpreted accordingly. After the completion of the command, this register contains the Status data, and each of its bits describes the current status of the device.

2.1.3 ATA IDENTIFY command

One command that deserves more insight is the ATA IDENTIFY (or ATA IDENTIFY DEVICE) command. The host can, thanks to this command, request to receive parameter information from the device. Generally, in the ACS, there are commands that can be implemented or not by a device, depending on the manufacturer preferences. However, the ATA IDENTIFY command must be implemented on every ATA hard drive. This command OpCode is 0xEC and it needs to be placed in the Command register of the Device. The value of the other registers is N/A (Not Applicable).

The identification data returned by this command is arranged in a 256 words (512 bytes) structure whose composition is illustrated in the ACS description of the ATA IDENTIFY command. For our purposes, three of the fields in this structure are particularly relevant. These are the *Serial Number*, found in the words 10-19 and made of 20 ASCII characters, the *Firmware Revision*, found in the words 23-26 and made of 8 ASCII characters, and the *Model Number*, found in the words 27-46 and made of 40 ASCII characters. The usage of this data for our purposes will be illustrated in later chapters. These three pieces of information uniquely identify any real hard drive device.

2.1.4 The SCSI protocol

The SCSI protocol defines commands in a similar way to the ATA protocol. These can be found in the *SCSI Command Reference Manual*. The main structure that constitutes a SCSI command is called Command Descriptor Block (CDB). To issue a command to a SCSI device it is necessary to send this structure, containing proper values, which can be accompanied by additional information in the data buffer.

Sense data is the mechanism used by the SCSI protocol to provide feedback information after a command has been executed. In practice, it is a structure, called sense buffer, containing multiple fields that provide different type of information. Particularly, the first field of the sense buffer is the *response code* that can assume different values, depending on the type of sense data returned. For the descriptor format sense data the values can be 0x72 or 0x73, while for the fixed format sense data the values can be 0x70 or 0x71. Depending on the format, the whole sense buffer changes the structure of its fields. The three additional fields are useful for our purposes:

- **Sense Key:** This is a 4-bit field that can have up to 16 values. Each one of these values indicate generic information describing an error or exception condition.
- **Additional Sense Code (ASC):** The ASC field provides further information related to the error or exception condition that is reported in the Sense Key field. When no further information is provided by the device, this field shall be set to zero.
- **Additional Sense Code Qualifier (ASCQ):** The ASCQ field provides further information to the one provided by the ASC field. It is the last level of hierarchy of information and it shall be set to zero when no further information is provided.

The Linux driver that allows to send SCSI commands is called "SCSI Generic Driver" (sg) and it is the most accessible way for a host to communicate with a hard drive. Table 2.1 illustrates the `sg_io` structure, the main structure of the driver. Some of these fields are set by the host before issuing the command, while others are set by the device upon command completion.

Below is the description of the fields of the structure that are significant to the purposes of our work:

Byte/Byte	0	1	2	3	4	5	6	7
0	interface_id				dxfer_direction			
1	cmd_len	mx_sb_len	iovec_count		dxfer_len			
2	dxferp				cmdp			
3	sbp				timeout			
4	flags				pack_id			
5	usr_ptr				status	masked_status	msg_status	sb_len_wr
6	host_status		driver_status		resid			
7	duration				info			

Table 2.1: SCSI Generic Structure (sg_io)

- **dxfer_direction**: This field indicates the direction of the data transfer associated with the command. It can assume five values: none, to the device, from the device, to and from the device, unknown. It cannot be uninitialized or initialized to zero.
- **cmdp**: This field contains the pointer to the CDB structure of the SCSI command to be executed.
- **cmd_len**: Contains the length in bytes of the SCSI command that cmdp points to.
- **sbp**: This is the pointer to the Sense Buffer where the sense data will be written by the device.
- **mx_sb_len**: The maximum size in bytes that can be written in the array pointed by sbp.
- **sb_len_wr**: The actual number of bytes that have been written in the array pointed by sbp.
- **dxferp**: For commands that require data transfer, this field points to the user memory where the data will be transferred to or from.
- **dxfer_len**: This field indicates the length of memory pointed by dxferp.

2.1.5 ATA_Pass_Through

Even though the SCSI protocol is not directly correlated with the objectives of our work, it granted us a simple and accessible solution for the non trivial task of sending ATA commands from a Linux host. For practical purposes, the SATA disks under Linux use the SCSI subsystem as interface, which means that a SATA device is recognized as a SCSI device. To be able to communicate with it it is possible to use a particular SCSI command called ATA_Pass_Through. This is implemented by a translation bridge mechanism that extract the encapsulated ATA command transported through the SCSI protocol.

There are two versions of ATA_Pass_Through: 12 byte and 16 byte. These correspond to the two types of available ATA command: 28-bit and 48-bit. Table 2.2 illustrates the structure of the two SCSI CDBs where it is possible to see that,

Byte	Description
0	OpCode (0xA1)
1	Setup_Byte_1
2	Setup_Byte_2
3	Features
4	Sector_Count
5	LBA_Low
6	LBA_Mid
7	LBA_High
8	Device
9	Command
10	Reserved
11	Control

Byte	Description
0	OpCode (0x85)
1	Setup_Byte_1
2	Setup_Byte_1
3	Features (15:8)
4	Features (7:0)
5	Sector_Count (15:8)
6	Sector_Count (7:0)
7	LBA_Low (15:8)
8	LBA_Low (7:0)
9	LBA_Mid (15:8)
10	LBA_Mid (7:0)
11	LBA_High (15:8)
12	LBA_High (7:0)
13	Device
14	Command
15	Control

Table 2.2: ATA_Pass_Through CDB 12 byte version (to the left) and 16 byte version (to the right)

from bytes 3 to 9, for the 12 byte CDB, and from bytes 3 to 14, for the 16 byte CDB, the content corresponds to the ATA registers, as they are defined in the ACS.

The OpCode field has the purpose of indicating which SCSI command is contained in the CDB. Its value is 0xA1 for the ATA_Pass_Through 12 byte CDB and 0x85 for the ATA_Pass_Through 16 byte CDB. The last byte of both structures, the Control byte, is very poorly documented by the T10 documentation and has always been assigned value 0 in all our tests. Setup Bytes 1 and 2 carry information about the command to be executed, that is used by the translation mechanism.

Byte/Bit	7	6	5	4	3	2	1	0
1	Multiple_Count			Protocol				Reserved
2	Off_Line	CK_Cond	Reserved	T_Dir	Byt_Blok	T_Length		

Table 2.3: Setup Bytes 1 and 2

- **Protocol:** It represents the type of operation that the command requires to execute. Examples are: Non-data, PIO Data-In, DMA, etc., up to 16 different options.
- **Byt_Blok:** Indicates whether T_Length specifies the number of bytes to transfer or the number of blocks to transfer (0-bytes, 1-blocks).
- **CK_Cond:** If set it requests the translation bridge to return the ATA register information in the sense data, even if no error or exception occurs. In this case the Sense Key will contain value 0.

- **T_Dir:** Specifies the direction of the data transfer. If 0 the transfer is from host to device, if 1 it is from device to host. Its content is ignored if T_Length is 0. If T_Dir and Protocol fields carry conflicting values, then an ILLEGAL REQUEST value will be returned in the Sense Buffer.
- **T_Length:** Used to communicate to the translation bridge the transfer length. In particular 0 corresponds to no data transfer, 1 indicates that the transfer length is specified in the Features field of the CDB, 2 indicates that the transfer length is specified in the Sector_Count field of the CDB.

2.2 Virtualization

The term *Virtualization* refers to the process of running a virtual instance of a computer system in a layer abstracted from the actual hardware. In other words virtualization software allows the user to run multiple operating systems using only one physical computer. More generally, virtualization may refer to any instance of one IT resource hosting several other IT resources, including programs, databases, users, or networks. Commonly, the physical machine is called "host" system, while the virtual environments are called "guest" systems. An ideally perfect virtualization technique would make it impossible for any operating system to distinguish between an underlying real or a virtual environment. Today's technologies are getting increasingly closer to this level of isolation.

Although virtualization technology can be traced back to the 1960s, until the early 2000s it has not been widely adopted. Technologies that allowed virtualization (such as hypervisors) were created decades ago to provide simultaneous access to computers conducting batch processing for multiple users. This was a popular business computing style that consisted in performing routine tasks very quickly in one large group instead of individually. However, other approaches to the "many users/one machine" problem have gained popularity over the next few decades, while virtualization has not. One of those alternatives was time-sharing, which separated users within operating systems, leading to the birth of UNIX, that eventually gave way to Linux. In the meanwhile, virtualization remained a niche technology, adopted only in small scale.

Skip forward to the nineties, most businesses were equipped with physical servers and software components from a single vendor, that generally did not allow older, existing applications to run on hardware from another vendor. When businesses upgraded their equipment with cheaper and more convenient servers, operating systems and applications from a variety of vendors, they were obliged to run underused physical hardware, since each server was only able to run one vendor-specific task. This created a strong appeal for the virtualization techniques, which were able to provide two main benefits. Firstly, it was now possible to partition the servers and run legacy applications on different operating systems types and versions. Secondly, the server hardware could now be used more efficiently, helping reducing the cost of purchase, set up, cooling and maintenance. Virtualization made it harder for vendors to "lock" their customers into utilizing only their products and became the foundation of cloud computing.

The main component behind virtualization technologies is called *hypervisor*. This software is responsible for the separation between the physical resources and the virtual environments. They can either be installed on top of an operating system (usually the case for private, personal computer users) or directly onto hardware (very common for server environments), which is the technique mainly used by enterprises.

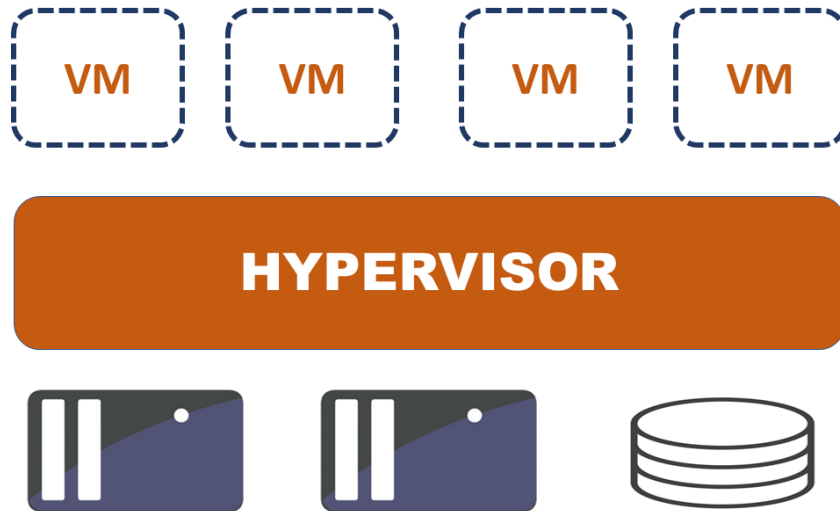


Figure 2.6: Graphical representation of the role of a hypervisor in the virtualization mechanism.

The job of a hypervisor is to partition physical resources in such a way that fulfills the needs of the virtual environments. Users can run computations and perform tasks in the virtual environment, usually called *virtual machine*. Effectively, a virtual machine is nothing else than a single data file, and as such it can be moved, copied and reused with no drawbacks. While running a virtual machine, it is possible that the user issues an instruction whose requirements exceed the current amount of resources allocated from the physical environment. In this case the hypervisor communicates with the physical system and requests the additional resources.

As any resource in a virtual machine, the hard drives are also virtualized. There are several types of disk image formats that are used by different virtualization softwares. The most common ones are QCOW2 (QEMU copy-on-write), VDI (VirtualBox Disk Image) and VHD (Virtual Hard Disk). All of them have different perks and features and are generally connected to one specific virtualization software. They can be contained in a single file or multiple files, and they can have a fixed size (entirely allocated when created) or a dynamic size (that increases when the virtual machine is used).

One convenient feature of virtual environments is the possibility of taking snapshots. A snapshot allows the user to save the exact state of a virtual machine at a given point, that can be restored at a later time, after changes have been applied

to the virtual machine. Generally snapshots are taken in an offline state, which means that the virtual machine has to be powered off in order to save a snapshot. However, some virtualization software offer the possibility of taking live snapshots, saving this way the entire state of the machine (RAM, etc.) and not only the current disk state.

Today, there are many different virtualization software solutions, designed in different ways by different parties. The three main implementations that we used in our work are QEMU, VirtualBox and VMWare.

2.2.1 QEMU

QEMU (Quick EMUlator) is one of the oldest solutions for virtualization. It is open-source and completely free. It was born as a system emulator, capable of running Operating Systems and programs that were made for one specific machine (for example an ARM board) on a different physical machine (for example a PC). In later years, thanks to a Linux kernel module called Kernel-based Virtual Machine (KVM), it became able to execute guest code directly on the host CPU. This allows it to obtain significantly better performances, making it possible to achieve near native speed.

Many APIs have been created to aid with the usage of this virtualizer, that was originally born as a command line tool equipped with a vast amount of options and modes of usage. Main examples of softwares like these are *libvirt* and *vagrant* that, even though are not specifically designed for QEMU since they are also capable of facilitate the usage of other virtualization softwares as well (like VirtualBox or VMWare), they provide options that help take advantage of all the options offered by QEMU.

It is possible to take snapshots of a QEMU virtual machine, however taking and managing live snapshots can be troublesome and complicated, reason for which it is not advised to use QEMU for necessities like this. Offline snapshots are very efficient and easily scriptable.

2.2.2 VirtualBox

VirtualBox is another free and open-source virtualization tool, owned and developed by Oracle, capable of performing x86 virtualization. Differently from QEMU, this solution is much more user friendly, and is generally the solution preferred by mass consumers that are approaching the virtualization world. It is possible to install it on almost all Operating Systems and it is equipped with a very intuitive graphical user interface. VirtualBox is capable of running both as a hardware based virtualization tool, when hardware assisting solutions exist for the host machine, or as a software based virtualization tool, simply taking advantage of the operating system user space.

There are also command line solutions for managing VirtualBox (the main one is called "vboxmanage") that allows scripting and automating virtual machines creation and usage. Snapshot in VirtualBox are extremely accessible and easy to use, both the live version or the offline version.

2.2.3 VMWare

Differently from the previously described virtualization solutions, VMWare is a commercial solution, implying that the source code is not available, and in order to be able to use all the functionalities it is necessary to purchase a license. There is however a free version that includes basic functionalities that are sufficient for our purposes. In particular it is possible to create one or multiple virtual machines and to attach multiple virtual hard drives. Snapshots however are not included in the basic version of the software, making it harder to reproduce the same experiments.

Chapter 3

The fuzzing system

3.1 Client-Server fuzzing model

Generally, in a fuzzing system, there are two main roles that come into play: generate the fuzzing data and performing the actual fuzzing. The model we used to implement this system is a Client-Server model. The Client takes the role of the data generator: its purpose is to create random bytes and send them to the Server, which will be listening on a specified port. The Server's task is to properly build the data structures where the bytes received from the Client will be encapsulated. The Server sends these structures to the target Hard Drive, then collects the feedback data it returns. It also has other features, like logging the commands before sending them, logging the results after the commands have been executed and checking for any unexpected behaviour in the fuzzing target.

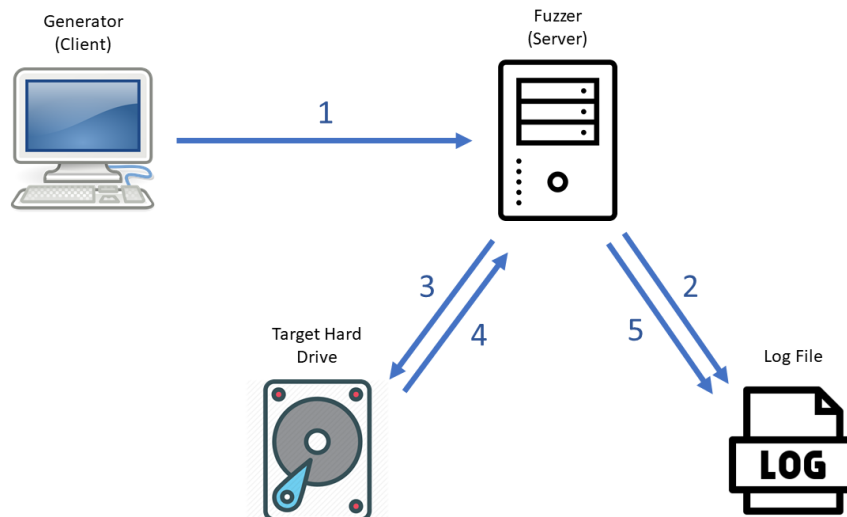


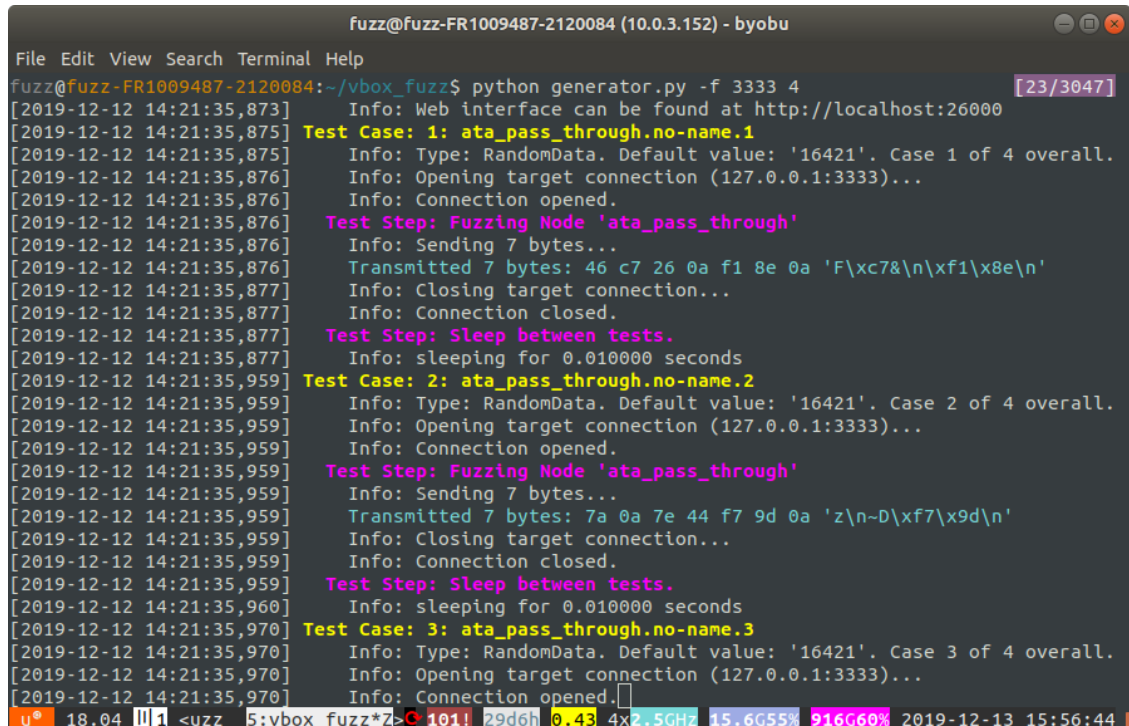
Figure 3.1: Graphical depiction of the Fuzzing model. 1) Generate the fuzzed bytes; 2) Log the command; 3) Issue the command; 4) Wait for execution and fetch results; 5) Log the results;

One of the main benefits of having this kind of structure becomes evident when dealing with Virtual Environments. By separating the command generating component from the command executing component we have the possibility of splitting the tasks between the Host machine and the Guest machine. It is indeed possible to have the Client component running outside the Virtual Machine, while the Server component runs inside it, with no major changes in the way the model functions. Another perk of this model is the capability of modifying one component at a time, without the need to change the other component.

The implementation of the Client was done by using a third party fuzzing framework called "Boofuzz", while the Server was coded in two main versions, whose features will be highlighted in the following paragraphs, using the Python and C programming languages. We will mainly refer, for the rest of this document, to the Client as the "Generator" and to the Server as the "Fuzzer".

3.2 The fuzzing data Generator: Boofuzz

Boofuzz is a fuzzing framework written in python 2.7 with a big variety of features. For instance it is capable of recording test data, detect failures, restart the target process and more. However, in our specific case, many of these features cannot be taken advantage of, since what we are trying to fuzz are not processes but communication interfaces with hard drives. Consequently, we used it mainly for data generation, printing of fuzzing information and, in some cases, for logging purposes.



```

fuzz@fuzz-FR1009487-2120084 (10.0.3.152) - byobu
File Edit View Search Terminal Help
fuzz@fuzz-FR1009487-2120084:~/vbox_fuzz$ python generator.py -f 3333 4 [23/3047]
[2019-12-12 14:21:35,873] Info: Web interface can be found at http://localhost:26000
[2019-12-12 14:21:35,875] Test Case: 1: ata_pass_through.no-name.1
[2019-12-12 14:21:35,875] Info: Type: RandomData. Default value: '16421'. Case 1 of 4 overall.
[2019-12-12 14:21:35,876] Info: Opening target connection (127.0.0.1:3333)...
[2019-12-12 14:21:35,876] Info: Connection opened.
[2019-12-12 14:21:35,876] Test Step: Fuzzing Node 'ata_pass_through'
[2019-12-12 14:21:35,876] Info: Sending 7 bytes...
[2019-12-12 14:21:35,876] Transmitted 7 bytes: 46 c7 26 0a f1 8e 0a 'F\xc7&\n\xf1\x8e\n'
[2019-12-12 14:21:35,877] Info: Closing target connection...
[2019-12-12 14:21:35,877] Info: Connection closed.
[2019-12-12 14:21:35,877] Test Step: Sleep between tests.
[2019-12-12 14:21:35,877] Info: sleeping for 0.010000 seconds
[2019-12-12 14:21:35,959] Test Case: 2: ata_pass_through.no-name.2
[2019-12-12 14:21:35,959] Info: Type: RandomData. Default value: '16421'. Case 2 of 4 overall.
[2019-12-12 14:21:35,959] Info: Opening target connection (127.0.0.1:3333)...
[2019-12-12 14:21:35,959] Info: Connection opened.
[2019-12-12 14:21:35,959] Test Step: Fuzzing Node 'ata_pass_through'
[2019-12-12 14:21:35,959] Info: Sending 7 bytes...
[2019-12-12 14:21:35,959] Transmitted 7 bytes: 7a 0a 7e 44 f7 9d 0a 'z\n-D\xf7\x9d\n'
[2019-12-12 14:21:35,959] Info: Closing target connection...
[2019-12-12 14:21:35,959] Info: Connection closed.
[2019-12-12 14:21:35,959] Test Step: Sleep between tests.
[2019-12-12 14:21:35,959] Info: sleeping for 0.010000 seconds
[2019-12-12 14:21:35,970] Test Case: 3: ata_pass_through.no-name.3
[2019-12-12 14:21:35,970] Info: Type: RandomData. Default value: '16421'. Case 3 of 4 overall.
[2019-12-12 14:21:35,970] Info: Opening target connection (127.0.0.1:3333)...
[2019-12-12 14:21:35,970] Info: Connection opened.
u 18.04 111 <uzz 5:vbox_fuzz*Z> 101! 29d6h 0.43 4x2.5GHz 15.6G55% 916G60% 2019-12-13 15:56:44

```

Figure 3.2: Example of the Boofuzz command line interface.

The Boofuzz framework is designed to work within a client-server architecture,

with the fuzzing data generator taking the role of the client. In our model, when launched, Boofuzz will try to connect to the server, which is the fuzzer designated to send the generated bytes to the target hard drive. For each one of the multiple, consecutive generated bytes, the generator client creates a new connection with the fuzzer, and closes it after the command is sent.

3.3 The Fuzzer

As mentioned previously, the main task that the Fuzzer component of the system has to execute is taking the bytes transmitted by the Generator and sending them to the target Hard Drive. This is achievable through the usage of a particular system call, the *ioctl()* system call, which is nothing more than an interface that allows to execute device specific input/output operations. In our case, the specific device we want to access is the target Hard Drive, usually mapped in Linux under the path `/dev/sd?`, where the “?” indicates one letter in alphabetical order (a, b, c, etc.), depending on the number of hard drives connected to the machine at a given time. For each Hard Drive, the partitions contained in it are also mapped under a similar path. For instance, if we consider the `/dev/sda` drive, each of its partitions will be mapped under `/dev/sdaN`, where “N” is an increasing number representing the partition.

In order to issue an ATA command, incapsulated in a SCSI command, the *ioctl()* system call requires three parameters. The first parameter is the file pointer to the device we want to communicate with, that will be mapped under a specific `/dev/sd?` path. The second argument is a request code that depends on the type of device you are trying to interact with. In our specific case this request code is defined in the *scsi/sg.h* kernel source code, under the name *SG_IO*, and it corresponds to the hexadecimal value `0x2285`. By passing this value to the system call, we communicate to the kernel our intention of communicating with a SCSI drive, since, as mentioned previously, in Linux both SCSI and SATA drives are seen by the kernel as SCSI devices. The third parameter required by the *ioctl()* system call is a generic untyped pointer to a memory location. In this location the kernel expects to receive data in a certain format, that depends on the type of request passed in the second argument. In our specific case, it expects to find the pointer to the beginning of the area of memory containing the *SCSI Generic Structure*, which, in turn, will contain the pointer to the *ATA_Pass_Through* structure. This is the main and most important task that the Fuzzer has to execute, to properly build and load the received bytes in the *ATA_Pass_Through* and the *SCSI Generic Structure*, to then call the *ioctl()* system call which will execute the transmitted command.

The second task executed by the Fuzzer part of the system is the monitoring and collection of the data returned by the device. As mentioned previously, after the issue of each *ATA_Pass_Through* command, feedback data containing information about the outcome of the command execution is returned in the SCSI Sense Buffer. The Fuzzer, first of all, check the return status of the *ioctl()* function, to ensure that the command execution has been transmitted properly. Then it checks the byte 0 of the sense buffer and, depending on its content, it identifies the right location in

the structure that will allow it to find the Sense Key value, the ASC value and the ASCQ value.

After few experiments with the fuzzing system, we introduced another check to be performed by the Fuzzer: a liveness check. It would often happen, mainly with real Hard Drives, that after sending fuzzed commands, the device would either get into an anomalous state, or it would be disconnected from the machine by the kernel. To recognize when these events happened, we introduced what we called a "liveness" check, which simply consists in sending an ATA IDENTIFY command after the device executes a fuzzed command, and checking that the contents of the identify data returned are the expected ones (meaning the ones that the device would return in a normal state). The downside of this approach is the mild slowdown introduced by the execution of this extra command after every fuzzed command. However, in most cases, this was not relevant since the time required by the IDENTIFY command to be executed is several orders of magnitude smaller than the time required by other commands, especially the ones inducing a timeout.

Logging is another crucial feature we needed to implement in our fuzzing system. The most trivial way of doing so would have been by using the logging mechanism provided by the Boofuzz framework. Unfortunately this option would have been insufficient, since the only data Boofuzz is capable of logging, are the fuzzing bytes created by the Generator. This required us to implement the logging mechanism in the Fuzzer component. The way it works is by, first of all, logging the command bytes prior to sending the command, then logging both the sense data returned by the device and the outcome of the liveness check. The timestamp is also recorded together with the other data.

One curious peculiarity of the interaction with the Hard Drives is present in the ATA IDENTIFY command. The 512 byte structure containing the device information and returned by this command appears to be byteswapped, even on little-endian architectures. By analyzing the code of other open-source disk utilities it seems that they also take this problem in consideration and advise to byteswap the returned data before utilizing it, on all platform, not just big-endian ones. We couldn't find any specific reason for which this happens, but our Fuzzer programs had all to keep this problem in consideration.

3.3.1 The Python Fuzzer

The first version of the fuzzer, whose development begun during the semester project, uses Python 2.7 as programming language. This is mainly due to the ease of use of the language and to the fact that Boofuzz, the backbone of our Generator component, is a Python framework, which makes it straightforward to use the same language also for the Fuzzer. Python also provides an easily implementable interface with the C libraries, in particular to the *ioctl()* system call, essential for sending commands to the hard drive.

The way we designed this component was by splitting it into two modules that we called *fuzzer.py* and *ata.py*:

Module *ata.py*

Its task is to mainly deal with the communication between the Fuzzer and the Hard Drive. It receives the `ATA_Pass_Through` structure, filled with the fuzzed bytes by the *fuzzer.py* module, it builds the *SCSI Generic Structure* and properly initializes its fields with appropriate values, then it makes the call to the `ioctl()` function. It is also appointed to collect the returned data, and to forward it to the *fuzzer.py* module. Some extra features implemented by this module are the ability to execute few different ATA commands for testing purposes, for example the ATA IDENTIFY command, or the READ/WRITE PIO commands. It is also the module in charge of taking care of byteswapping the identify data.

Module *fuzzer.py*

Main module designed to be the bridge between the Generator component and the *ata.py* module. It receives the fuzzed bytes, encapsulates them into the `ATA_Pass_Through` structure then forwards it. It is also the module in charge of printing and logging results, as well as providing several different fuzzing modalities that can be selected when launching the program from command line:

- *Fuzzing with identify check (-f)*: Standard fuzzing mode that sends the fuzzed bytes received from the Generator to the Hard Drive then performs a liveness check after each command.
- *Fuzzing without identify check (-fn)*: Very similar to the previous mode, except for the fact that the liveness check is not performed anymore. This brings a slight improvement in performance, since there are less commands to be executed, and also performs less log operations.
- *Offline fuzzing with identify check (-fo)*: This mode doesn't use the Generator component anymore as it takes the input bytes from a file passed on the command line. This mode is useful when anomalous behaviour happened with a certain batch of commands and we want to replicate them. Few modifications are required to the log files for them to be made compatible input files for this mode.
- *Offline fuzzing without identify check (-fon)*: Similar to the previous mode without the liveness check.
- *Execute an ATA IDENTIFY command (-id)*: This mode is for testing purposes. It executes an ATA IDENTIFY command and prints the Serial number, Firmware revision and Model number contained in the data returned by the Disk.
- *Execute an ATA READ PIO command (-r)*: This mode is for testing purposes. It executes an ATA READ PIO command and prints its content. The LBA number of the sector to be read must be provided.

- *Execute an ATA WRITE PIO command (-w)*: This mode is for testing purposes. It executes an ATA WRITE PIO command and prints its content. The LBA number of the sector to be written must be provided.

At start, the way the Fuzzer operated was by simply opening a connection with the Generator, receiving the fuzzed data and sending it to the Hard Drive, then closing the connection and opening a new one for the next command. In the general case this method has no evident flaws since the Generator waits until one connection is closed before opening a new one. However, in the particular case in which we are dealing with a Virtual Environment and we need to have the Generator component outside the Virtual Machine, while the Fuzzer component is inside, problems may arise mainly because the Generator and the Fuzzer are not synchronized anymore, because they are not communicating directly anymore, but through the Virtual Machine port forwarding mechanism. This means that the Generator will keep generating data, while the Fuzzer may be much slower to process these commands. This situation could quickly bring to the saturation of the buffer between the Virtual Machine and the Host Machine. To deal with this problem we implemented a queue of commands inside the Fuzzer. In particular we created a second thread whose task is to collect the commands sent by the Generator and insert them into a queue. The main thread, will then fetch them from the queue in a FIFO (first-in first-out) manner. This method has been working reliably in all the experiments performed, as the number of commands present in the queue at the same time never reaches levels that could be troublesome to deal with.

Another peculiarity of the Python Fuzzer that is worth mentioning is the fact that it is only capable of executing 12 bytes ATA_Pass_Through commands, with the first three bytes (the bridge setup bytes) fixed to values that are appropriate for a READ PIO command. Moreover, this Fuzzer is only capable of sending CDBs that are entirely randomly generated by the Generator component, including the OPCode of the command to be executed. This means that this fuzzer does not only fuzz ATA commands that are defined by the T13 ACS document, but it fuzzes all the possible 256 values contained in the OPCode byte of the CDB.

3.3.2 The C Fuzzer

As it will be illustrated later in the document, the fuzzing method described previously, that was implemented by the Python Fuzzer, had a certain degree of rigidity to it. This first Fuzzer was only able to send 12-byte versions of the ATA_Pass_Through command, while neglecting completely all the 48-bit ATA commands that require a 16-byte ATA_Pass_Through structure. In addition, the way the Python Fuzzer builds the ATA_Pass_Through structure is by maintaining the first three bytes fixed to the same values, whichever is the command that it executes. This caused many uncertainties due to the fact that it was not clear whether the commands would actually reach the SATA drive or they would just get discarded by the translation bridge, since the setup bytes are not appropriate to the command we are trying to execute. In other words it was not easily determined whether we were fuzzing the SCSI to ATA translation bridge or the actual ATA

device, as intended. Due to these factors, we decided to design a slightly different Fuzzer, that would be able to fuzz both the 28-bit and the 48-bit ATA commands, and that would be able to modify the setup bytes in accordance to the command that is to be executed. To be able to this, we had to limit the set of fuzzed OpCodes to only those that are actually defined by the ACS.

While the first intention was to just extend and modify the way the Python Fuzzer worked, some other factors came into play that skewed our plan towards using the C language instead of Python. First of all, the C language brings some intrinsic advantages and disadvantages: due to its low-level nature, it is much faster than Python, which takes significant computing time for the interpretation process. C code is, however, more difficult to write and maintain, and it requires extra attention in some particular cases, for example when dealing with dynamically allocated memory.

To understand the main reason that led to the choice of this programming language, we are now going to introduce the disk utility tool named *hdparm*. This is an open source command line program for Linux that allows the user to interact with an ATA hard drive by granting the possibility to view and change disk hardware parameters and to test performance. The way its code is implemented is by having few "handler" functions that are called with different parameters, depending on the ATA command the user is willing to execute. This paradigm is remarkably close to what we need to implement in our Fuzzer: a way of changing the setup bytes in the `ATA_Pass_Through` structure, depending on the ATA command we are trying to execute. For this reason we decided to reuse these snippets of code from *hdparm*, and to modify them to fit our purposes. We implemented the ATA commands individually, by testing their behaviour and the feedback data returned by the disk, reaching a total of 45 implemented commands (table 3.1).

One difference with the Python Fuzzer is that the C Fuzzer does not implement the multi-threaded queue of commands. This is mainly due to the fact that we did not have the necessity to use this Fuzzer in a Virtual Machine, while keeping the Generator outside, and consequently that feature was not required. A similar case can be made in regards to the offline fuzzing, another feature that has been omitted for lack of necessity.

The operating modes of this Fuzzer, with the corresponding command line flag, are listed below:

- *Fuzzing without identify check (-f)*: Standard fuzzing mode that sends the fuzzed bytes received from the Generator to the Hard Drive. No liveness check executed.
- *Fuzzing with identify check (-fid)*: Identical to the previous mode, with the liveness check included.
- *Fuzz with only ATA SMART command (-smart)*: This fuzzing mode keeps the OpCode of each sent command fixed and equal to the ATA SMART command. This is because this specific command has multiple subcommands, that depend on the value of the *features* field in the CDB. All these subcommands constitute an interesting fuzzing surface, which is the purpose of this mode.

COMMAND CODE	COMMAND NAME
0x03	CFA_REQ_EXT_ERROR_CODE
0x06	DSM
0x08	DEVICE_RESET
0x10	RECAL
0x20	READ_PIO
0x24	READ_PIO_EXT
0x25	READ_DMA_EXT
0x27	READ_NATIVE_MAX_EXT
0x29	READ_MULTIPLE_EXT
0x30	WRITE_PIO
0x34	WRITE_PIO_EXT
0x35	WRITE_DMA_EXT
0x39	WRITE_MULTIPLE_EXT
0x3d	WRITE_DMA_FUA_EXT
0x3f	READ_LOG_EXT
0x40	READ_VERIFY
0x41	READ_VERIFY_ONCE
0x42	READ_VERIFY_EXT
0x44	ZERO_EXT
0x57	READ_LOG_DMA_EXT
0x5b	TRUSTED_NON_DATA
0x5c	TRUSTED_RECEIVE
0x5d	TRUSTED_RECEIVE_DMA
0x5e	TRUSTED_SEND
0x60	READ_FPDMA
0x61	WRITE_FPDMA
0x70	SEEK
0x87	CFA_TRANSLATE_SECTOR
0x90	DIAGNOSE
0x98	CHECKPOWERMODE2
0xa0	PACKET
0xa1	PIDENTIFY
0xb0	SMART
0xb8	CFA_ACCESS_METADATA_STORAGE
0xc0	ERASE_SECTORS
0xc6	SET_MULTIPLE
0xc8	READ_DMA
0xca	WRITE_DMA
0xe7	FLUSHCACHE
0xe8	WRITE_BUFFER
0xea	FLUSHCACHE_EXT
0xec	IDENTIFY
0xef	SETFEATURES
0xf0	IBM_SENSE_CONDITION
0xf8	READ_NATIVE_MAX

Table 3.1: List of the 45 ATA commands implemented by the C Fuzzer

- *Fuzz with only ATA SETFEATURES command (-feat)*: Similar mode to the previous one. In this case the fixed command is the ATA SETFEATURES command, which is also composed of multiple subcommands.

Chapter 4

Fuzzing Virtual and Real Hard Drives

4.1 Qemu experiments

4.1.1 Command Tracing Mechanism

As previously explained, our fuzzing model was based on sending ATA commands through the SCSI translation mechanism. One of the downsides of this model was the uncertainty of how many of the fuzzed ATA commands we issued (including the ones with non-existing ATA command code) were actually able to reach the ATA device, main target of our system, and how many were stopped by the SCSI translation layer. In order to obtain more information about this issue, we took advantage of one of the many utilities made available by the QEMU configuration options: the tracing mechanism. This is a tool that allows the user to keep track of many different types of variables in the virtualization system. For our purposes, the virtual machine variables we were interested in were the ones corresponding to each ATA command sent by our fuzzing system.

To enable this option, it is enough to toggle some specific configuration variables during the compilation process of QEMU. At this point the Command Code of every ATA command that is executed inside the virtual machine will be logged by QEMU into a file. To be able to analyze these logs and extract the desired information it was enough to write a script in python whose main function was to count, rank and print the ATA commands. To test the correct operability of the tracing mechanism we tried to simply monitor a normal running virtual machine (no fuzzing involved). In figure 4.1 it is possible to see the output example of a Ubuntu 18.04 machine running for ten minutes.

Thanks to this new tool, we were able to test all the 256 possible ATA Command Codes and observe whether they were tracked or not by this mechanism. The results are displayed in table 4.1. All the other Command Codes that are not displayed in the table were processed and detected regularly by the virtual machine. It is also possible to see that even though some commands were detected as passed, it took the system an abnormal amount of time to process them (generally between


```

Statistics for tracefile trace.cmds
Total number of commands: 20515

Occurrences Command Code/Name

12923 (0xc8) READ DMA
3183 (0x20) READ SECTOR(S)
3075 (0xca) WRITE DMA
829 (0xa0) PACKET
313 (0xe7) FLUSH CACHE
136 (0x25) READ DMA EXT
16 (0x35) WRITE DMA EXT
13 (0xec) IDENTIFY DEVICE
11 (0xa1) IDENTIFY PACKET DEVICE
9 (0xb0) SMART (command family, see docs)
3 (0xe5) CHECK POWER MODE
2 (0xef) SET FEATURES
1 (0xe0) STANDBY IMMEDIATE
1 (0x30) WRITE SECTOR(S) / WRITE SECTOR(S) WITHOUT RETRIES

```

Figure 4.1: Occurrences of ATA commands in a regular Virtual Machine

30 seconds and 2 minutes). This is probably due to the fact that they toggled some timeout mechanism either in the SCSI translation bridge or in the ATA device. The commands that did not pass were simply not detected by the tracing mechanism and returned Sense Data that indicated an error in the SCSI CDB ("Illegal Request - Invalid field in CDB"). Thanks to these tests we concluded that there is a high probability that command codes that are not defined by the ATA standard should be capable of reaching the ATA device.

PASSED/NOT PASSED	COMMAND CODE	COMMAND NAME
Not Passed	0x22	READ LONG
Not Passed	0x23	READ LONG WITHOUT RETRIES
Passed (Slow)	0x30	WRITE SECTOR(S) (WITHOUT RETRIES)
Passed (Slow)	0x31	UNKNOWN (OBSOLETE COMMAND)
Not Passed	0x32	WRITE LONG
Not Passed	0x33	WRITE LONG WITHOUT RETRIES
Passed (Slow)	0x34	WRITE SECTOR(S) EXT
Passed (Slow)	0x39	WRITE MULTIPLE EXT
Passed (Slow)	0x3c	CFA WRITE VERIFY / WRITE VERIFY
Not Passed	0x5c	TRUSTED RECEIVE
Not Passed	0x5d	TRUSTED RECEIVE DMA
Not Passed	0x5e	TRUSTED SEND
Not Passed	0x5f	TRUSTED SEND DMA
Passed (Slow)	0xc5	WRITE MULTIPLE
Passed (Slow)	0xe6	SLEEP

Table 4.1: Traced ATA commands

4.1.2 Code Coverage

Another technique we used to evaluate the efficiency of our fuzzing system was measuring the QEMU source code coverage while issuing fuzzed commands. Code coverage is a measurement of how many lines and branches of code are executed

when running a program. This was again possible thanks to the options provided by QEMU, where, in order to enable coverage monitoring, it is enough to toggle few configuration variables in the compilation process. It is possible to view the coverage extract both in a graphical version, in HTML format, and in a text-based version. In the graphical version, the information provided are the percentage and number of code lines executed for each source file, percentage and number of branches executed for each source file, and the total percentage and line coverage for all the monitored files. In the text-based version all the previously mentioned information are available, plus the number of each line that has not been executed, allowing for a more in-depth analysis.

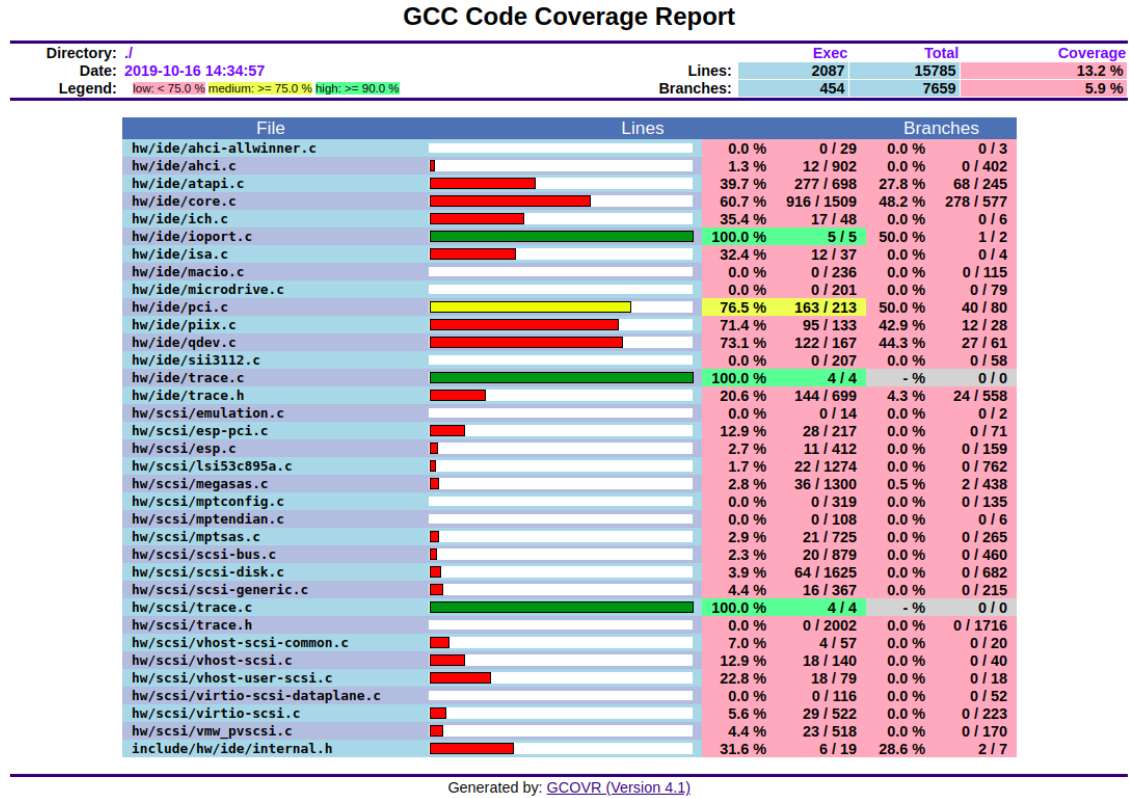


Figure 4.2: Example of the graphical representation of the coverage report

With this tool we analyzed the behaviour of both our Fuzzers. We focused our attention mainly on the source code files that were involved with the management of the ATA/SCSI commands, since these were the ones directly impacted by our fuzzer. Generally, the results we observed were that the larger part of the lines of code were covered during the first few hundreds of commands issued. Following, there would only be a minor increase in coverage, which tended to become smaller with the increasing number of commands sent. This is plausible since the more commands we send, the more probable it is for these new commands to execute lines of code that were already executed previously. However, since the increase in coverage was considerably small, it is also possible that our fuzzer could spend a remarkable amount of time sending commands that were not executing any new code. Nonetheless, we considered this to be acceptable, since, even though the increase in coverage was negligible, it would have been enough to find a single specific combination of input bytes that could bring the VM to an erroneous state.

4.1.3 First set of experiments

One of the results obtained during the semester project was that, after letting the Python Fuzzer run for a variable amount of time, the virtual machine file-system was set to read-only, causing the fuzzing system to crash since it became impossible for it to generate new commands. It is important to note that the virtual hard disk, target of our fuzzing system, was the main hard disk containing the virtual machine file-system.

In order to understand more about what was happening we ran several experiments. We extracted a batch of nineteen thousand (19k) ATA commands that caused, in a previously fuzzed virtual machine, its file-system to become read-only. By having this batch of commands, it is possible to try to recreate the same conditions by sending these commands to multiple, newly created, virtual machines. We used Vagrant to help us automatize the fuzzing process, by creating a new Vagrant box containing Ubuntu 18.04 with the Linux Kernel version 5.0.0 and with all the necessary software, fuzzing system included, already installed. By scripting the creation and the execution of the virtual machines, we were able to run tests multiple times in an efficient manner. However, since the results obtained from these tests were sometimes different from our expectations, we decided to also run some tests without using Vagrant, to ensure that by using multiple layers of software we were not bringing changes to the virtual machines configuration.

In particular, these were the different experiments we tried to run:

- First of all, we tried to create two new VMs (without using Vagrant) and fuzz them with the 19k commands batch. In one of them, the file-system became read-only, however it happened with a smaller number of commands, before all the 19k were processed. In the second VM the file-system was never set to read-only.
- By using Vagrant, we launched four fresh VMs, two of which ran with the 19k commands batch, one ran with the last 12k commands of the 19k batch and one ran with the first 10k commands of the 19k batch. All of them finished sending their sets of commands without the file-system becoming read-only.
- With the help of Vagrant, we tried to launch five newly setup VMs and fuzz them with 300k new randomly generated commands. After more or less one day, the file-system of three of them became read-only (after approximately 50k-60k commands), while the file-systems in the other two VMs never became read-only and completed all the 300k commands after approximately three to four days.
- From the three VMs whose file-system became read-only, we extracted three new batches of commands and we executed them on three freshly setup VMs. None of these VMs's file-system was set to read-only after these batches of commands.

The results of these experiments brought us to the main conclusion that the behaviour of the virtual machines to the issued commands were not deterministic.

It is clear that it is not possible to reliably replicate the reason for which the file-system is set to read-only. Instead there seems to be a certain degree of randomness.

4.1.4 Investigating the Read-Only problem

In order to understand more about why the file-system would sometimes be set to read-only, we decided to check the system, kernel and journal logs for error messages. This was possible to do after the read-only event since there were no write operations involved in the log reading process. It is displayed in figure 4.3 the output of the kernel log (in Linux, under the path `"/var/log/kern.log"`) of one of the virtual machines in read-only mode. From these messages it is inferable that some problems occurred in one of the directories inodes, and that the kernel intervened by setting the file-system to read-only, to avoid any additional damage. To better comprehend what happened we need to understand how the EXT4 file-system works under Linux, in particular how directories are managed.

```
Jul 26 11:52:22 fuzz1vbox-VirtualBox kernel: [ 1178.263144] atkbd serio0: Unknown key released (translated set 2, code 0x6a on isa0060/serio0).
Jul 26 11:52:22 fuzz1vbox-VirtualBox kernel: [ 1178.263149] atkbd serio0: Use 'setkeycodes 6a <keycode>' to make it known.
Jul 26 11:52:24 fuzz1vbox-VirtualBox kernel: [ 1179.479185] ata3.00: invalid multi_count 1 ignored
Jul 26 11:52:24 fuzz1vbox-VirtualBox kernel: [ 1179.479252] ata3.00: invalid multi_count 1 ignored
Jul 26 11:52:24 fuzz1vbox-VirtualBox kernel: [ 1180.022305] EXT4-fs warning (device sda1): ext4_diritem_csum_verify:355: inode #418705: comm updatedb.mlocat: No space for directory leaf checksum. Please run e2fsck -D.
Jul 26 11:52:24 fuzz1vbox-VirtualBox kernel: [ 1180.022308] EXT4-fs error (device sda1): htree_dirblock_to_tree:979: inode #418705: comm updatedb.mlocat: Directory block failed checksum
fuzz1vbox@fuzz1vbox-VirtualBox: /var/log$
```

Figure 4.3: Kernel log of a VM with its file-system set to read-only

In Linux, directories are considered files, and as such they have an inode associated to them. An inode is a metadata structure that contains information about a file, and in case of a normal file, it also contains the file data (or pointers to where the file data is stored). In case of a directory, there is no data to be stored. What is stored are instead the directory entries. These are nothing else than a list of mappings between the names of the files contained in the directory and their inodes.

Offset	Size	Name	Description
0x0	_le32	inode	Number of the inode that this directory entry points to
0x4	_le16	rec_len	Length of this directory entry
0x6	_u8	name_len	Length of the file name
0x7	_u8	file_type	File type code
0x8	char	name[EXT4_NAME_LEN]	File name

Table 4.2: Structure of an EXT4 directory entry (`ext4_dir_entry`)

By analyzing this structure more in depth, it is possible to notice that the first four fields have always the same size, since they contain numbers represented on a defined number of bytes. The file name, instead, has a variable length, that can go up to 255 characters (255 bytes). These structures are normally stored in a linear array that is contained in the previously mentioned directory data blocks. To make

these blocks more robust it is necessary to add checksums structures. These are nothing else but phony structures (identical to the `ext4_dir_entry` from a physical point of view) placed at the end of each directory data block.

Offset	Size	Name	Description
0x0	__le32	det_reserved_zero1	Inode number, which must be zero
0x4	__le16	det_rec_len	Length of this directory entry, which must be 12
0x6	__u8	det_reserved_zero2	Length of the file name, which must be zero
0x7	__u8	det_reserved_ft	File type, which must be 0xDE
0x8	__le32	det_checksum	Directory leaf block checksum

Table 4.3: Structure of a leaf directory entry (`ext4_dir_entry_tail`)

This special directory entry is always 12 bytes long. The inode number and `name_len` fields are set to zero to fool old software into ignoring an apparently empty directory entry. The checksum is stored in the place of the filename and in this particular case it has a fixed size of four bytes, since the size of the checksum value stored in it does not depend on the data.

Going back to our particular case, we decided to investigate the Linux kernel source code. We found that the "No space for directory leaf checksum" message was printed when the checks (figure 4.4) on the fixed values of the first four fields in the `ext4_dir_entry_tail` structure failed. Since they are fixed when the structure is created and they are not modified in any way by the Kernel code, it was clear that there must have been some kind of memory corruption that overwrote the values of the said structure. This could be seen as a normal consequence of our actions. Since we were fuzzing the virtual hard drive containing the OS file-system, it was definitely plausible that some random write commands could be issued to write to random parts of memory, overwriting the checksum directory entry and the fixed values fields.

However the observed behaviour was not as straightforward as expected. First of all we need to keep in mind that these tests were run with the Python Fuzzer, which kept the setup bytes of the `ATA_Pass_Through` fixed to values corresponding to a read operation. Several tests showed that in a real machine it was impossible to write on disk with such setup bytes in the structure. Moreover, the writes on

```
333
334     if (t->det_reserved_zero1 ||
335         le16_to_cpu(t->det_rec_len) != sizeof(struct
336             ext4_dir_entry_tail) ||
337         t->det_reserved_zero2 ||
338         t->det_reserved_ft != EXT4_FT_DIR_CSUM)
339         return NULL;
340     return t;
```

Figure 4.4: Linux kernel code snippet containing the leaf directory entry checks.

the disk were not consistent: by running tests restoring multiple times the same initial snapshot and running the exact same batch of commands, the corruption of the disk did not happen every time in the same manner. Even if the address at which the disk was overwritten were very similar (when running multiple tests with the same batch of commands) the amount of memory overwritten was practically always different both in size. Moreover, the data that was written in the corrupted sectors also looked pretty arbitrary: at times address values would get printed, some other times the return data of the ATA IDENTIFY command (which probably derived from the successive IDENTIFY command that was sent after each fuzzing command) was printed, in other cases data from other files or directories would get leaked. This kind of volatile behaviour brought us to the conclusion that there must be a memory leak involved, since it is the main memory (and not the disk) whose content tends to change over time.

```

00000850 9e a7 0f 00 b6 31 04 00 c9 50 02 00 cb e7 03 00 |.....1...P.....|
00000860 96 a7 0f 00 00 00 00 00 b7 31 04 00 16 12 02 00 |.....1.....|
00000870 cc e7 03 00 98 a7 0f 00 a0 a7 0f 00 b8 31 04 00 |.....1.....|
00000880 c4 50 02 00 cc e7 03 00 99 a7 0f 00 a1 a7 0f 00 |.P.....|
00000890 b9 31 04 00 22 8a 13 00 cc e7 03 00 77 a7 0f 00 |.1...".....w...|
000008a0 00 00 00 00 ba 31 04 00 fb e5 00 00 cd e7 03 00 |.....1.....|
000008b0 a0 a7 0f 00 a3 a7 0f 00 bb 31 04 00 65 8a 13 00 |.....1..e...|
000008c0 cd e7 03 00 33 d8 06 00 a4 a7 0f 00 bc 31 04 00 |....3.....1..|
000008d0 dd 7b 00 00 cd e7 03 00 8b a7 0f 00 a5 a7 0f 00 |.{.....|
000008e0 bd 31 04 00 66 8a 13 00 cd e7 03 00 a7 c4 28 00 |.1..f.....(|
000008f0 a6 a7 0f 00 be 31 04 00 ee 7b 00 00 cd e7 03 00 |.....1...{.....|
00000900 8c a7 0f 00 a7 a7 0f 00 bf 31 04 00 ef 7b 00 00 |.....1...{...|
00000910 cd e7 03 00 8d a7 0f 00 a8 a7 0f 00 c0 31 04 00 |.....1..|
00000920 b8 11 02 00 cd e7 03 00 8e a7 0f 00 a9 a7 0f 00 |.....|
00000930 c1 31 04 00 b6 11 02 00 cd e7 03 00 8f a7 0f 00 |.1.....|
00000940 aa a7 0f 00 c2 31 04 00 67 8a 13 00 cd e7 03 00 |.....1..g.....|
00000950 00 00 00 00 41 31 1e 00 c3 31 04 00 fb e5 00 00 |....A1...1.....|
00000960 ce e7 03 00 a2 a7 0f 00 ac a7 0f 00 c4 31 04 00 |.....1..|
00000970 ca 50 02 00 ce e7 03 00 9a a7 0f 00 ad a7 0f 00 |.P.....|
00000980 c5 31 04 00 c6 50 02 00 ce e7 03 00 9b a7 0f 00 |.1...P.....|
00000990 ae a7 0f 00 c6 31 04 00 c7 50 02 00 ce e7 03 00 |.....1...P.....|
000009a0 9c a7 0f 00 af a7 0f 00 c7 31 04 00 c8 50 02 00 |.....1...P..|
000009b0 ce e7 03 00 9d a7 0f 00 b0 a7 0f 00 c8 31 04 00 |.....1.....|
000009c0 c9 50 02 00 ce e7 03 00 9e a7 0f 00 b1 a7 0f 00 |.P.....|
000009d0 c9 31 04 00 68 8a 13 00 ce e7 03 00 33 6d 28 00 |.1..h.....3m(.|
000009e0 b2 a7 0f 00 ca 31 04 00 dd 7b 00 00 ce e7 03 00 |.....1...{.....|
000009f0 a4 a7 0f 00 b3 a7 0f 00 cb 31 04 00 ee 7b 00 00 |.....1...{...|
00000a00
fuzz1vbox@fuzz1vbox-VirtualBox: /usr/src/linux-headers-4.18.0-25/drivers/infiniband/hw$

```

Figure 4.5: Memory leak in a part of the virtual hard drive.

4.1.5 The ATA SLEEP command

While running the previously mentioned fuzzing experiments we stumbled upon one particular anomalous behaviour. In some cases, the liveness check (which consists of the IDENTIFY command after each fuzzing command) would fail, as it did not return the expected IDENTIFY data it usually returns during normal conditions (see 2.1.3). Instead, the 512 bytes structure generally contained only null bytes. At times, however, leaked bytes would be present in this structure. The leaked data looked very similar to the leaks described in the previous section (corrupted writes), meaning that sometimes addresses would get leaked. In figure 4.6 it is possible to observe the raw bytes, in hexadecimal form, contained in the IDENTIFY command output. By paying particular attention, it is noticeable that these values

able to fuzz command codes that were actually implemented by the ATA protocol. Moreover, instead of keeping the setup bytes in the `ATA_Pass_Through` structure fixed, it was capable of changing the bytes to better fit the ATA command to be issued.

In this case we ran in parallel 16 virtual machines in one host. They were all setup with one core and 2GB of RAM. This is how we divided the 16 virtual machines:

- Three virtual machines were fuzzing with the old Python Fuzzer. The main reason behind this choice was trying to see if there could be possible interactions between virtual machine (which of course would be a very anomalous behaviour since VMs are designed to be isolated systems)
- Seven virtual machines were fuzzing with the C Fuzzer, without performing the liveness check. By not executing an ATA IDENTIFY after every command it was possible issue more commands.
- Four virtual machines were fuzzing with the C Fuzzer, executing the liveness check after each fuzzing command.
- One virtual machine was only fuzzing the ATA SET FEATURES command. We decided to isolate this command and fuzz it because of its particular structure. This command is made of multiple subcommands, that can be chosen by the value in the Features register. Our intention was to explore all the possible subcommands.
- One virtual machine was only fuzzing the ATA SMART command. Similar to SET FEATURES, this is another command made of multiple subcommands.

The general difference with the previous fuzzing mechanism was mainly the fact that it was possible to send command much faster, since there were considerably less timeouts involved. This was probably due to the fact that many timeouts are triggered when executing Command codes that are not expected by the SCSI to ATA translation mechanism. For this reason, it was possible to fuzz with considerable amounts of commands (millions) in acceptable periods of time (few days). This brought us to observe a new anomaly.

More precisely, a general RAM flooding problem arised. This is to be expected when processing huge amounts of data, as millions of commands. There was, however, a problem when fuzzing with the C fuzzer, with all the 46 ATA commands that it implemented. In particular, here is what happened when we tried to run the different tests:

- When fuzzing with the C fuzzer with all 46 commands, after sending an amount of commands ,generally included between 1 and 1.2 million, the vm RAM got flooded and the VM slowed down until the Boofuzz or the Fuzzer process got killed. However, even after both these processes are shut down, the RAM is not freed. The VM is still extremely slow and, by analyzing the memory status with different tools, it is detected as full, without identifying

any process that is occupying the memory. By letting different VMs in this condition run, they all crashed after a couple of days the flooding occurred.

- We tried running the Fuzzer without actually sending the commands, by commenting the line of code containing the `ioctl()`. The reason behind this was to try to isolate the problem and identify it more easily. In particular we thought that Boofuzz could be causing the flooding problem, so by eliminating the Fuzzer component from the scheme we would be able to see how the system would behave. The result was that, after Boofuzz generated 18.6 million commands, it got killed by the Kernel for its memory usage. However, after killing it, the RAM got freed and the VM went back to a normal status.
- The VM running the C fuzzer with only the ATA SETFEATURES command behaved in a similar way to the case with the commented `ioctl()`. After 18.8 million commands the RAM got flooded and the processes were killed. The RAM was freed and the VM went back to a normal state.
- Regarding the Python Fuzzer, it runs slower than the C Fuzzer, so it only got to 7 million commands after 6 days of fuzzing. It didn't create any issues, and the feeling is that it would also behave like the two previous cases, where Boofuzz would probably fill the memory up but the VM would be able to recover and return to a normal state.

Summarizing, the general trend seems to be that Boofuzz floods the RAM memory when it generates considerable amounts of data. but when no substantial fuzzing is happening, the kernel is able to manage it normally: it detects the use of too much RAM, it kills the process that is using it and returns to a regular state. However, when paired with the Fuzzer sending real ATA commands in rapid succession, the memory gets flooded much faster and it doesn't seem like the Kernel is able to restore the machine to a normal state, even after killing the processes. In figure 4.7 it is possible to observe the output of the "top" command in a VM in these conditions, after the processes have been killed.

```
top - 11:01:04 up 15:20, 5 users, load average: 4.45, 5.59, 6.25
Tasks: 138 total, 3 running, 78 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 7.8 sy, 0.0 ni, 0.0 id, 67.0 wa, 0.0 hi, 25.3 si, 0.0 st
KiB Mem : 2041316 total, 52824 free, 1971340 used, 17152 buff/cache
KiB Swap: 1459804 total, 1402972 free, 56832 used, 5054 avail Mem
```

PID	USER	PR	NI	VRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1289	root	20	0	129196	1404	1376	R	14.9	0.1	24:52.53	apt-get
1	root	20	0	77444	552	552	S	7.8	0.0	16:16.33	systemd
864	fuzz	20	0	44156	900	724	R	6.0	0.0	19:55.06	top
785	fuzz	20	0	32180	0	0	S	3.2	0.0	42:21.09	tmux: server
541	root	20	0	31320	0	0	S	1.7	0.0	0:48.43	cron
34	root	20	0	0	0	0	D	1.5	0.0	5:12.19	kswapd0
237	root	0	-20	0	0	0	R	0.9	0.0	7:35.64	kworker/0:1H
7	root	20	0	0	0	0	S	0.6	0.0	6:47.29	ksoftirqd/0
752	fuzz	20	0	107984	0	0	S	0.6	0.0	38:31.40	sshd
550	root	20	0	287548	0	0	S	0.4	0.0	3:00.55	accounts-daemon
1329	root	20	0	0	0	0	I	0.4	0.0	0:08.83	kworker/0:1
1274	man	20	0	17896	0	0	D	0.2	0.0	0:04.45	mandb
2	root	20	0	0	0	0	S	0.0	0.0	0:00.03	kthreadd
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker/0:0H
6	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_percpu_wq
8	root	20	0	0	0	0	I	0.0	0.0	0:59.83	rcu_sched
9	root	20	0	0	0	0	I	0.0	0.0	0:00.00	rcu_bh
10	root	rt	0	0	0	0	S	0.0	0.0	0:00.00	migration/0
11	root	rt	0	0	0	0	S	0.0	0.0	0:01.00	watchdog/0

Figure 4.7: Screenshot of the "top" process running in a flooded VM.

It is evident that the RAM is almost all used but no active process can be identified as the responsible. As previously mentioned, all the VMs in these conditions crash after one or two days.

4.2 VirtualBox experiments

The second virtualization software we tested was VirtualBox. This tool is more modern than QEMU and its improved usability makes it a more appealing choice for the general consumer. The advanced functionalities offered by this software helped us to facilitate the tests we ran. In particular the very intuitive snapshot interface made it trivial to run several times the same tests in order to understand more about the very unpredictable behaviour of the memory corruption anomaly. The online snapshot functionality was what allowed the virtual machine to return to the exact state every time and produce similar results, without having to reboot it which, even slightly, changed the internal configuration.

Regarding the results obtained while using this virtualization software, they were in general very similar to the ones obtained with QEMU. The memory corruption happened also in VirtualBox virtual machines, as the ATA SLEEP leak. For the RAM flooding anomaly, it manifested in the same way in the VirtualBox VMs as it did in the QEMU ones. The only slight difference was related to the command line interface, known as the "vboxmanage" tool. Usually this tool is able to list both all the created VMs and the running VMs. However, after the memory flood event happened, the tool was not able to list the running VMs anymore, working as if no machine was running anymore. Although, it was still able to recognize that the machines were still existing. We were not able to find any plausible explanation for this event, however it could be correlated to just some internal bug in the VirtualBox code when dealing with memory flood virtual machines, as we were not able to extensively test if this behaviour happened only when fuzzing, or if it happened when the RAM got flooded.

Thanks to VirtualBox, we were able to improve the runnability of tests, which made us understand better the events that were happening. These details are described in the next paragraphs.

4.2.1 Experiments on VirtualBox

When changing the virtual environment from QEMU to VirtualBox, two main differences in the behaviour of the virtual machines to fuzzing were quickly noticeable:

- After running the fuzzing system for a while, in general some hundreds of commands but it would change a lot depending on the commands sent, because of some fuzzed commands that were sent to the virtual hard disk, the VM would pause with a popup error message about the I/O cache. However it could be resumed manually without any problems, and it would continue to operate normally. The same behaviour happened both when using the software through the GUI and through the command line interface. The advantage brought in this case by the command line interface was the possibility

of scripting the unpausing command, in order to automatize this behaviour. Our script works by sending an unpausing request every second to the virtual machine. In this way there is a good compromise between the computational overhead caused by this additional running process and the downtime during which the virtual machine is paused and not able to send fuzzing commands.

- In VirtualBox, the commands are processed much faster than in QEMU, which is probably due to the way the two different softwares manage the interface with the virtual hard drives. More specifically, we were able to process the batch of 19k commands in more or less thirty minutes with VirtualBox, while in QEMU it normally takes a couple of hours. This is mainly due to the fact that in QEMU there are more timeouts that last for a longer period of time, while VBox doesn't seem to have the same problem.

As previously mentioned, VirtualBox's snapshot facility allowed us to run more tests on the memory corruption issue. In particular our strategy in this case was to save, in an online snapshot, the state of a virtual machine right before fuzzing it, then run the fuzzer until it corrupted the memory, restore the snapshot and try again. The result was that the file-system would become read-only after a similar number of commands every time. Here is what happened in the five tests we ran:

1. read-only after 1852 cmds
2. read-only after 1833 cmds
3. read-only after 1870 cmds
4. read-only after 1822 cmds
5. read-only after 1827 cmds

This indicates that there must be a deterministic element that is making the corruption happen every time. The small difference in number of necessary commands is probably due to the time it takes the guest kernel to identify the corruption, and to set the file-system to read-only. Also what we found was that this behaviour was strictly correlated to the current state of the machine, since by running the same batch of commands on a different virtual machine, or even on the same virtual machine but after rebooting it, the memory would not get corrupted with the same deterministic level.

In order to dig deeper into why the memory corruption was happening, we tried to isolate one or few commands that caused this to happen. Our strategy was to make the corruption happen with a certain batch of commands from 0 to x , restore the previous snapshot and try running the two half batches of commands of the previous test, in particular from 0 to $x/2$ and from $x/2$ to x . In this way we could monitor which of the two new sets of commands caused the memory leak, and repeat the process until possible. This is what is called a dicotomic search, and it is theoretically the fastest way of isolating the target command without knowing anything about which one it is.

What we found was that there was an ATA WRITE DMA EXT command (0x35 is the command code) in the set of command and it was most likely the one responsible for the writes on the disk. Even though it did not write anything when sent individually, it always worked when sent in pair with another command or bunch of commands. Moreover, we were able to find a correlation between the LBA written in the registers of the command and the start of the corrupted memory area. First of all we started running these tests on a different virtual hard drive, which was empty, in order to avoid corrupting the file-system (and to avoid restoring the snapshot after every test). This also made it trivial for us to locate the memory area that was corrupted. We found that after every test, the start of the corrupted memory area would be the same, even though the amount of memory written would generally be different. By checking what the LBA written in the ATA command register were, we found out that it corresponded with the start of the corrupted area. This means that the leaked write was indeed caused by an ATA command, even though the setup bytes of the ATA_Pass_Through CDB were set to only enable read operations.

Since we were only able to reproduce this behaviour in a virtualized environment, we can conclude that with high probability there are some bugs present in the SCSI to ATA translation mechanism. This not only allows operations that are not permitted to be executed, but it also leaks parts of the guest virtual machine kernel memory.

4.2.2 Incompatibilities with ATA_Pass_Through

Aside from the previously mentioned behaviours, VirtualBox revealed also some kind of incompatibilities problems with the SCSI to ATA translation mechanism. While running the tests, one thing we stumbled upon was an anomalous output when sending read/write commands to the virtual disk in VirtualBox. By using the Python Fuzzer, we tried sending well formed ATA READ_PIO and WRITE_PIO commands through an ATA_Pass_Through CDB. These two commands worked as intended both on a QEMU virtual machine and on a real Hard Disk. However, in the case of the READ_PIO command, when ran in a VirtualBox virtual machine, the returned data was most of the time NULL bytes, instead of the real data that was written on the disk. Sporadically, it also happened that some leaked bytes would get returned, such as addresses or files' data. When testing the WRITE_PIO command, nothing would get written on the virtual disk, while the same command worked normally on QEMU and real hard drive.

To ensure that no bug was present in our code, we also tried running the same tests with the `hdparm` utility. This is a widely used and tested tool so its reliability is far superior to our Fuzzer code. By checking its source code, what we found was that it was using the same method for sending ATA commands, which was through the ATA_Pass_Through mechanism, making it an ideal option for double checking our findings. The results of these tests were exactly the same as the previous ones, meaning that no read or write command worked in a VirtualBox VM.

Our conclusions were that VirtualBox did probably not implement some parts of the SCSI to ATA translation layer, which caused the error Sense data to be returned

```

fuzz1vbox@fuzz1vbox-VirtualBox:/media/sf_shared$ sudo hdparm --yes-i-know-what-i-am-doing --read-sector 1 /dev/sdb
/dev/sdb:
reading sector 1: SG_IO: bad/missing sense data, sb[]: 70 00 05 00 00 00 00 0a 04 41 e0 01 21 04 00
00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00
succeeded
0015 a692 399f ffff 00d0 a2b1 ffff ffff
0100 0000 0000 0000 4097 c8c1 21e3 ffff
401b 6dc3 21e3 ffff c0b5 cbc2 21e3 ffff
c07b 3fc2 21e3 ffff 80af 55c0 21e3 ffff
8089 14c2 21e3 ffff 00c4 19c3 21e3 ffff
406d 63c3 21e3 ffff 406a 7ac2 21e3 ffff
4076 63c3 21e3 ffff 8055 4ec3 21e3 ffff
4036 eec1 21e3 ffff 801c 6dc3 21e3 ffff
c0e5 6dc3 21e3 ffff 4001 0ec2 21e3 ffff
40e8 3ac2 21e3 ffff 4004 d5c0 21e3 ffff
4019 e6c1 21e3 ffff 0023 12c2 21e3 ffff
c008 e6c1 21e3 ffff c085 14c2 21e3 ffff
00aa d5c0 21e3 ffff 0007 63c3 21e3 ffff
c016 19c3 21e3 ffff 8084 cdc1 21e3 ffff
0007 0ec2 21e3 ffff 8075 19c3 21e3 ffff
809b b9c2 21e3 ffff 8089 d5c0 21e3 ffff
8023 0ec2 21e3 ffff 400c 8bc2 21e3 ffff
8005 e6c1 21e3 ffff 400e eac1 21e3 ffff
40aa 65c0 21e3 ffff 40a0 8fc2 21e3 ffff
4098 fcc1 21e3 ffff 40f8 d2c1 21e3 ffff
c076 cdc1 21e3 ffff 405c d9c1 21e3 ffff
c0b7 b9c2 21e3 ffff 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
fuzz1vbox@fuzz1vbox-VirtualBox:/media/sf_shared$

```

Figure 4.8: Data returned by the ATA READ_PIO command using the utility tool "hdparm".

when executing a non implemented command. We knew however that some other parts of the protocol were implemented, since it was, for example, possible to run an ATA IDENTIFY command through the ATA_Pass_Through mechanism and receive the correct data from the virtual hard disk.

4.3 VMWare experiments

In order to better understand the results of the experiments, we decided to run them in a third virtualization software, in particular VMWare. This is again a very widespread software solution, even though it is not free. However, because of VMWare limited functionalities, we were not able to run extensive testing on VMWare.

First of all we tried to recreate the memory leak behaviour in the same way we did in QEMU and VirtualBox. Even though it took a big number of trials to be able to bring the VM to the "leaky" state, we were able to recreate the memory leak in the VMWare environment as well. The results were very similar to QEMU and VirtualBox, as the leaked memory would be made of addresses, data from files or other hard to identify bytes. This enforces our belief that the problem is connected to the linux kernel code, and the way it interacts with virtual machines.

Instead, we were able to find, by accident, a similar behaviour to the one found in VirtualBox. When first installing VMWare and creating a virtual machine, the default type of the virtual bus is set to SCSI instead of SATA. Since at first we were unaware of this, when we tried to send ATA commands with `ATA_Pass_Through` to the virtual hard drive, we obtained a similar behaviour to the one in VirtualBox. In particular, by sending read and write operations with `hdparm`, we would get Sense data that indicated an error (even though it was a different error than the one in VirtualBox), and also memory leaks in the returned data, looking the same as the ones in VirtualBox. Our interpretation of this is that there is no implementation of an `ATA_Pass_Through` command for a SCSI device. This is similar to what we think was happening in VirtualBox, where probably there is no implementation for these commands (even though the device is set to ATA in VirtualBox).

4.4 Real Hard Drives

As a continuation of the semester project, we tried to understand more about physical hard drives as well. In particular we were interested in the reduction of the size of the SSD device that had undergone extensive fuzzing sessions. As mentioned in the introduction, this had gone from being a 250GB to 2.4GB in size, after running the Python fuzzer for several days. By investigating more, we discovered that one of the ATA commands defined in the ACS, can be used to set the HPA (Host Protected Area). This is an area of the disk that is not accessible to the user and can only be unlocked through specific ATA commands. By using the same command, we were able to set the dimension of the SSD back to its original value, indicating that, when executing randomly fuzzed commands on the disk, we set its size to a smaller number. We concluded that this was a completely normal behaviour, as no anomaly or bug was found.

4.4.1 Malformed Sectors

While running additional tests on one of the real hard drives we fuzzed, we stumbled upon an anomalous behaviour in one of them. In particular some of its sectors were not accessible anymore. Normally, our default way of checking the memory content of a disk, both virtual and physical, was through the Linux command line utility named `"dd"`. This allows to perform input and output operations on the content of files and devices. When trying to print the entire content of the SSD device mentioned above, the command would give an error message when reaching a certain part of the device. Unfortunately, the error message was not too descriptive, as the only thing indicated was `"dd: error reading '/dev/sdc': Input/output error"`.

In order to investigate more, we decided to try to use the `ATA_Pass_Through` way of reading and writing a disk, by using both our program and the `hdparm` utility. What we found was that there were some areas of the disk, made by several sectors, which were not accessible by the `ATA_Pass_Through` commands. However, the sectors at the beginning and at the end of these areas, that were not accessible by the `dd` utility, were instead readable by the `ATA_Pass_Through`

commands. They were containing random looking bytes, instead of null bytes as all the other sectors in the device. We did not spot any recognizable data content, as for example addresses or file content, so we were not able to relate this behaviour to the one observed in the virtual machines. Everything returned to normal when we used the `dd` utility to write null bytes in those sectors. In this case the tool did not report any errors and proceeded with no interruptions. Afterwards, it was possible to normally read these sectors without any problem.

What we can conclude from these behaviours is that probably some fuzzed commands wrote some data on the disk that caused problem to the reading tools, like `dd` or the `ATA_Pass_Through` programs. When these bytes were overwritten with null bytes, everything returned to the standard behaviour.

Chapter 5

Conclusions

We found different anomalous behaviours through our testing schemes and tried to investigate them. First of all we continued the investigation of the memory corruption problem in the virtual machines, already discovered during the semester project, and that was causing the file-system to be set to read-only. This anomaly is most likely due to an incorrect management of the ATA protocol and device by the virtual environments. The memory corruption cases only happened in virtual machines, and they never happened with real hard disks, reason for which we think that the problem is due to the interface between the virtualization software and the Linux kernel. Another anomalous problem was the unpredictability with which these events happened, and that made it non trivial to test and recreate. More specifically, the memory leak did not happen in a reliable way, but it seemed more as it only happened when the virtual machine was in a particular state. We ran several tests to try to reliably bring the virtual machines to this state, but no solid pattern was found.

Even though a memory leak could be an interesting flaw from the point of view of a security tester, the data leaked in our case seems to belong to the guest kernel inside the virtual machine. This makes it less interesting from a security point of view, since in order to execute our fuzzing system, and to cause the memory leak to happen, we already need administrator permissions. With these kind of permissions it is already possible to inspect the memory, even if it is kernel memory.

The second anomaly we found related to virtual machines was the RAM flooding problem. In general, any process could allocate as much memory as possible, ending up using all the available RAM. This is what our Boofuzz component did, when generating a significant amount of commands. The normal operating system behaviour should be to eliminate the process that causes the RAM to flood, in order to restore the operability of the system. This happened in most cases, except when fuzzing with our C Fuzzer, which was designed to fuzz only with ATA commands defined by the ATA standard. In this case the RAM got flooded and remained allocated even after the processes were killed. This is probably due to an internal bug linked to the management of the ATA commands, where there may be present some kind of memory leak. This behaviour was present both on QEMU and VirtualBox. We were not able to test these conditions on a real machine since real devices are much slower than virtual devices, making it very hard to reach the

necessary number of commands for the memory flooding to manifest.

Regarding the real devices, we also performed tests to try to understand more about the size change in the device. By looking deeper into it we found out that there was nothing anomalous involved in this. All we did was changing the HPA (Host Protected Area) of the disk through a specific ATA command. This is an area of the disk that is hidden from the user, but it is still available. By randomly fuzzing we probably changed its size in the process. To restore the default size it was enough to send the same command again.

The general conclusion of our work is that hard drive communication systems have a good level of reliability, and we can state that, by fuzzing the ATA protocol, the probability of finding an important and crucial security flaw is very low. Different bugs were found, as described in this document, but none of them could be exploited to facilitate or create security attacks.

Bibliography

- [1] Mirabella Angelo, Sanislav Sebastian, Balzarotti Davide, Pagani Fabio, Marius Münch “Fuzzing Hard Drives”, Semester Project, Eurecom, 2019
- [2] Project T13/BSR INCITS 529, “Information technology - ATA Command Set - 4 (ACS-4)”, Working Draft American National Standard, 2016
- [3] David A. Deming, “The Essential Guide to Serial ATA and SATA Express”, CRC Press, 2015, ISBN: 978-1-4822-4333-8
- [4] Don Anderson, “SATA Storage Technology”, Mindshare Inc., 2007, ISBN: 978-0-9770878-1-5