SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Automated Test Case Generation for Emulators Using Symbolic Execution

Alp Berkman

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Automated Test Case Generation for Emulators Using Symbolic Execution

Automatische Testgenerierung für Emulatoren mit Hilfe von Symbolic Execution

Author:	Alp Berkman
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Sebastian Reimers, M.Sc. & Theofilos Augoustis, M.Sc.
Submission Date:	14th March 2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 14th March 2024 Alp Berkman

Acknowledgments

I would like to take a moment to acknowledge people who have helped me with my thesis. I am grateful to my supervisor Prof. Pramod Bhatotia for giving me a chance to work on this thesis. This endeavor would not have been possible without Sebastian Reimers's and Theofilos Augoustis's help and support. I would like to express my deepest gratitude to them for both being understanding and holding me accountable at the same time. Finally I would like to extend my sincere thanks to Nicola Crivellin for answering my many questions.

Abstract

In the last ten years, a computer revolution has been happening. Once, the most prominent CPU architecture was x86. However, new CPU architectures like ARM and RISC-V are gaining more popularity day by day and replacing x86 CPUs. These new architectures are commonly employed in PCs and cloud servers and in most cases, these devices use older software that was designed for x86 architecture. However, it is not easy to replace the software that is running on these devices. Thus, computer software called emulators are being used to run x86 binaries on these architectures.

While these tools offer significant advantages, they are not without flaws. Accurately emulating a CPU is a very complex task prone to errors. As a result, many emulators are riddled with bugs. Generally, when trying to replicate a bug, it is not a good idea to run the whole program repeatedly. Especially since some bugs may take considerable time to manifest or may only occur under very specific conditions and inputs.

In such scenarios, having a program capable of isolating the specific instructions and data that lead to an error would be extremely helpful. To address this, we have developed an add-on to expand a verifier, a program that checks the correctness of virtual machines, with a reproducer. This reproducer add-on can use the output of the verifier, the instructions, and the data to output a program that can trigger the bug. In other words, we can isolate the bugs from larger programs. The main benefits are that this program always uses the same data, meaning there is no input to worry and the produced program is tiny, which means debugging it is easier. With this reproducer, we aim to make debugging emulators easier to help emulator developers perfect their tools and increase the longevity of available programs.

Contents

1 Introduction

In this chapter, we will give an introduction to our thesis in order to prepare the readers. We will start by giving context to our reproducer, to emphasize its usefulness. Following that, we will talk about our motivation and try to explain its importance. We will then outline our approach, and explain the methods we are planning to use. To wrap up, we'll discuss the impact we anticipate the reproducer will have, highlighting the positive changes we expect to achieve.

1.1 Context

Considering the changes in the computing industry, new CPU architectures are gaining importance and spreading to more users. Although x86 CPUs are still the most common personal and server computer processors, they are being replaced by ARM and RISC-V CPUs. Nowadays these architectures are continuously developed. And they are gaining more prominence in the market due to different features. For example, ARM devices are being used for more power-constrained cases, where compared to processing power, the duration of the operation is more important. This results in some software, both legacy and new, being incompatible with newer computers that support these architectures.

Another similar problem arises from smartphones and tablets. According to [Sta] the market share of phones and tablets is already 50% higher than desktops and according to [The21] the number of phones has nearly doubled the number of people. All of these devices use ARM architectures, so there is a huge amount of software that is written for smartphones and tablets but they are not available for desktop computers. And vice versa the software that has been written for x86 CPUs is also not available for them.

If we want to run these programs on different hardware there are multiple methods. First of all, if the source code is available and is architecture agnostic, it can be recompiled. This is generally the preferred method as it yields the most efficient programs. However, if the source code is architecture-dependent or doesn't exist at all then we need to use a program called an emulator/virtual machine to run the existing binaries. These programs emulate a different CPU architecture which enables the computer to run programs that are compiled for that architectures.

1.2 Motivation

Considering the previous paragraphs, emulators are indispensable tools for prolonging software life and running them on different devices. At the same time, they are incredibly complex programs. For example Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals [Int23] is more than 5000 pages long. And ARM Architecture Reference Manual for A-profile architecture [**ARM_manual**] is nearly 13000 pages long. Since these manuals are this big and complex turning them into software is very error-prone. Therefore we need multiple testing mechanisms to make sure the emulators work exactly like the emulated hardware itself.

The most common tests are the following: unit tests, integration tests, functional tests, and regression tests. All of these tests can find bugs, but they require the developers to write the tests themselves. This act itself is very error-prone as they write both the emulator and the tests according to what they have understood. Another common method is fuzzing, where you use random inputs to trigger a bug. This method is rather hands-free since the users only need to specify the input format. However, using random inputs to trigger a bug for any program isn't as simple as it sounds. A general-purpose register in x86 CPU is 64 bits long. This means there are more than $1.8 * 10^{19}$ different values. If an instruction uses multiple registers, our chance of triggering a very specific edge case bug becomes impossible.

1.3 High Level Approach

In this paper, we propose a different method to single out bugs that exist on emulators. By comparing an emulator's log with an oracle's symbolic log, we can pinpoint the part that causes the bug. This program, which we will henceforth call the verifier, was built by Nicola Crivellin. Our contribution was to extend the verifier with a reproducer that could produce assembly instructions that could trigger the same bugs.

At the start, we assumed that these bugs would cause wrong jumps therefore affecting the flow of the program. But we have come to notice that most of the bugs stem from single instructions doing wrong calculations. This did change the direction of the program slightly but we are still producing a code snippet that should trigger bugs that were found with symbolic execution. This snippet includes start and exit stubs, the basic block/faulty instruction along with a setup for the registers and other changes in the memory.

1.4 Impact

This program was designed with a clear goal: to pinpoint the instructions within an emulator that lead to bugs and extract them. By focusing on this objective, the reproducer aims to isolate specific bugs within the broader context of a program that triggers unexpected errors in an emulator, making it significantly easier to identify and fix the root causes of these issues. This aspect of the reproducer is particularly beneficial for situations where an application, which operates flawlessly on original hardware, encounters unexpected crashes or errors when run on an emulator. Such discrepancies can be notoriously challenging to diagnose and resolve, as the faults do not lie within the application itself but rather within the underlying emulator that seeks to replicate the hardware environment.

The complexity of debugging these emulator-specific errors cannot be understated. Unlike straightforward application bugs, which can often be traced back to specific lines of code or logic errors, emulator bugs are intertwined with the nuances of hardware emulation. This program, therefore, stands as a useful tool for developers, offering a simple way to single out errors stemming from emulators.

In the broader context, the significance of this reproducer extends beyond the immediate realm of emulator development. As virtual machines and emulators become increasingly prevalent in personal and cloud computing environments, the reliability and accuracy of these systems take on new levels of importance. Some newer personal computers like Apple's M series depend on virtual machines [**rosetta**] to run legacy code. While cloud-based applications and services rely heavily on the seamless operation of virtual machines, with any discrepancies or faults potentially impacting a wide range of users and services. By improving the accuracy and reliability of emulators, this program not only benefits emulator developers but also contributes to the stability and efficiency of personal computers and cloud computing platforms. In doing so, it addresses a need in the tech industry, helping to mitigate challenging debugging scenarios and ensuring that virtual environments more closely mirror the behavior of real hardware.

2 Background

In this chapter, we are going to dive into the basics of how emulators work and why they are important. Understanding the process behind emulators is key to figuring out why certain errors occur. Firstly, we will explain what basic blocks are. Then, we will explore how emulators convert binary code so it can run on different types of computers. This conversion process is quite important because mistakes during it are often the source of the errors we are trying to fix. Knowing how this translation works helps us get to the root of the problem.

Next, we will discuss our primary emulator targets. While the reproducer and the verifier need generic input, there are two main emulators we're concentrating on. We will provide some insight into these emulators, including how they function and the specific techniques they use.

Lastly, we will look into the execution methods used by the verifier. These methods are significant because the reproducer relies on data obtained through them. Understanding these execution strategies is essential for understanding how the reproducer turns this data into programs that trigger bugs.

2.1 Basic Blocks

In computer science, basic blocks are code sequences that have a single entry point and no branches except at the exit. In other words, basic blocks are chunks of code that will always run the same way and exit at the same point. There is no braching in the middle and the order of instructions is always the same. If a program starts to execute a basic block, it will always continue until the end. This simplicity makes basic blocks easy to analyze, therefore they are often used in compilers, optimizers, disassemblers, and reverse engineering tools.

2.2 Binary Translation

Binary translation is an advanced technique that enables the execution of code compiled for one CPU architecture (the guest architecture) on a different CPU architecture (the host architecture). This process involves analyzing and converting the binary

instructions from the guest architecture into a form that can be understood by the host architecture.

Binary translation is particularly useful in scenarios such as software emulation, where applications or entire operating systems designed for one type of hardware need to run on an entirely different one. For instance, running a binary compiled for ARM architecture on an x86-based system would be impossible without any change. This necessitates binary translation.

2.2.1 Static Binary Translation

Static translation is a binary translation method that involves converting the entire binary code from the source architecture to the target architecture before execution begins. This approach allows for thorough analysis and optimization of the translated code, potentially leading to better overall performance for the translated application. However, static translation faces challenges in handling dynamic aspects of program execution, such as just-in-time compilation or self-modifying code, which are better managed by dynamic translation techniques. Static translation is well-suited for scenarios where the complete binary image is available and the execution environment is stable and predictable.

2.2.2 Dynamic Binary Translation

Dynamic translation is a subset of binary translation. It involves translating binary code at runtime, as the program executes. Dynamic translation offers the advantage of adaptability as it can optimize the translation based on the actual execution path of the program, which may vary from run to run.

This method is especially useful in emulating complex software where it's impractical to predict all possible execution paths in advance. However, it has some drawbacks regarding memory and performance. When a dynamic translator is running it requires extra space for the program that is being translated. It needs to allocate space for both the disassembled program and the translated instructions. This space may balloon very quickly at the start. The other drawback is the performance penalty of needing to translate instructions before executing them. This drawback is also mostly evident in the starting phase where the cache is empty. Overall programs that use dynamic binary translation tend to take up a lot of resources at the start, which decreases after the initial starting phase.

As shown in figure 2.2 dynamic binary translators have a mechanism against the performance drawback. They often incorporate a cache to store recently translated instructions, reducing the overhead of re-translating those instructions on subsequent

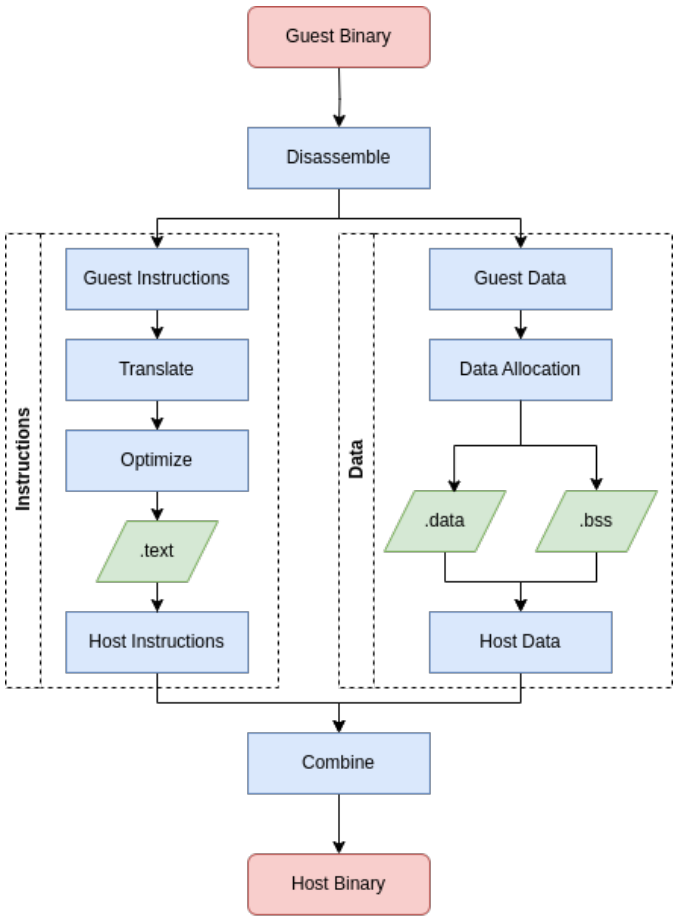

Figure 2.1: Static Binary Translation

executions. This caching mechanism is a key factor in mitigating the performance penalty associated with runtime translation, making dynamic translation an efficient and versatile approach for system emulation and virtualization environments.

2.3 Emulators

Computer emulators are software designed to mimic the hardware of a computer on another. This allows software designed for the guest system to operate on the host system. In this section, we will go into detail about our main targets. Then we will give a short explanation about the expected output of these emulators which we use for analysis.

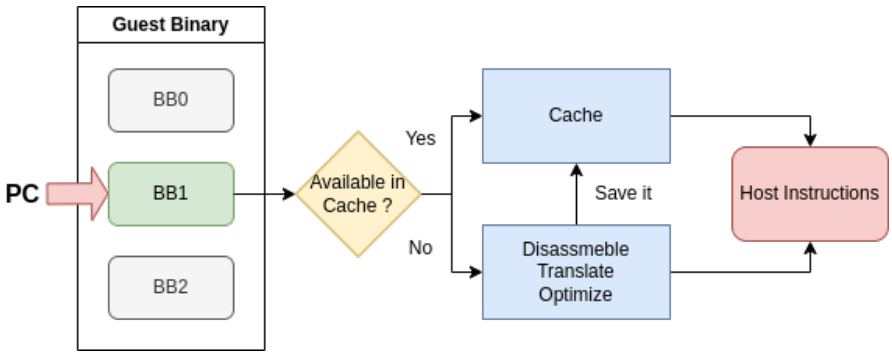

Figure 2.2: Dynamic Binary Translation

2.3.1 QEMU

QEMU is a well-known free and open-source emulator and a virtualizer. At its core, QEMU employs dynamic binary translation which lets the host machine run programs belonging to a different architecture. Enabling this is the Tiny Code Generator (TCG), an integral part of QEMU that dynamically generates native code for the host CPU. As shown in figure 2.3 the TCG is used to translate a foreign Instruction Set Architecture (ISA) into the host's architecture.

TCG

TCG [Devb] which began as a generic backend for a C compiler was later improved upon to be both portable and efficient, allowing QEMU to quickly translate the guest instructions into a form that can be directly executed by the host machine, thereby improving the speed and efficiency of the emulation process. This combination of dynamic binary translation and the flexibility of TCG enables QEMU to provide a high-performance and versatile solution for system emulation. Thanks to the flexibility of the TCG, many different architectures are supported by QEMU.

These include [Deva]:

- Arm
- MIPS (little endian)
- PPC
- RISC-V
- s390x

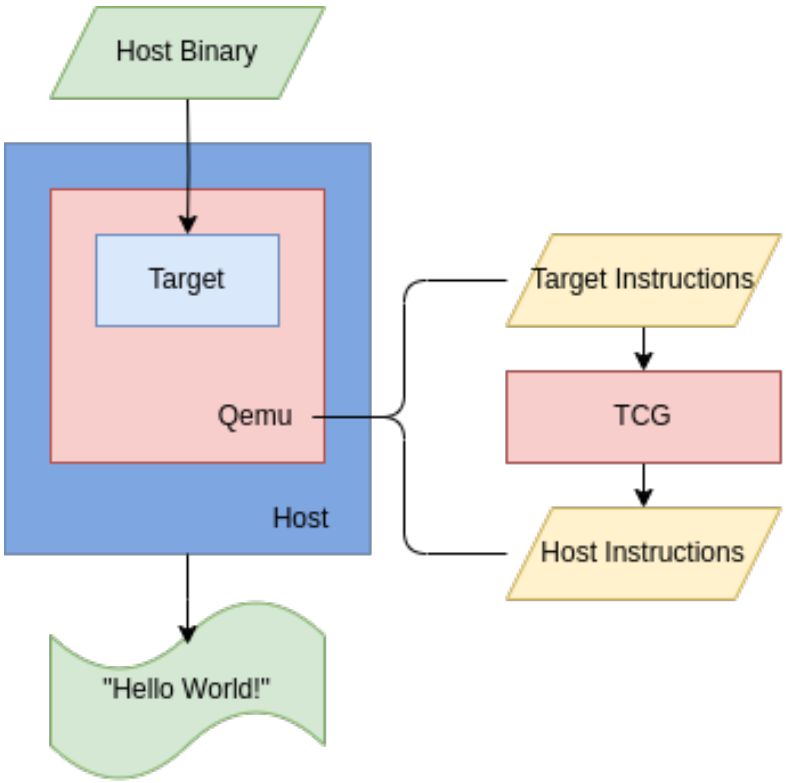

Figure 2.3: QEMU translation process with TCG

- SPARC
- x86

2.3.2 Arancini

Arancini is a project from the Systems Research Group at the Technical University of Munich (TUM). It builds on the knowledge gained from two earlier projects: Lasagna [Roc+22] which translates any x86 program statically to an Arm ISA and Risotto [Gou+22] which emulates x86 program dynamically on an Arm machine. Like these previous projects, Arancini focuses on making x86 programs work on Arm, but it also adds support for RISC-V systems. It uses a combination of LLVM, a well-known toolkit for building compilers, and its own custom translation technology to achieve this.

2.3.3 Emulator Logs

An emulator log is a detailed record generated by an emulator during its operation. They generally capture a wide range of information about the emulator's activities. For our reproducer to function properly we need to capture specific values either after every instruction or every basic block. These values include register values and all reads and writes in order. Generally, emulator logs include more information including the executed instruction, the translated code, and much more. However, the aforementioned values are enough to know the exact state the binary was in before executing the next step.

2.4 Execution Methods

2.4.1 Concrete Execution

Concrete execution refers to the traditional method of running programs where, the program operates on actual, specific input values to produce outputs. In this execution model, the program's instructions are carried out step by step, with each operation performed using concrete data values provided at runtime or predefined in the program. This method is straightforward and mirrors how programs are executed in real-world scenarios, making it intuitive and easy to understand.

Concrete execution is particularly useful for debugging, as it allows developers to trace the exact sequence of steps a program takes with a given set of inputs, observing the program's behavior and output directly. However, its reliance on specific inputs means that concrete execution can only explore one path through the program at a time, limiting its ability to uncover issues that may arise with different inputs or in untested execution paths.

2.4.2 Symbolic Execution

Symbolic execution, on the other hand, abstracts away from concrete input values, instead treating inputs as symbolic variables that can represent multiple possible values simultaneously. This approach allows the program to be executed in a way that explores multiple execution paths in a single run, by considering all the possible values that the symbolic variables might take. Symbolic execution builds a mathematical model of the program's execution paths, using symbolic expressions to represent the outcome of computations and decisions based on the symbolic inputs. As you can see in the figure 2.4 every possible branching instruction adds a new path.

This model can then be analyzed to identify potential bugs, security vulnerabilities,

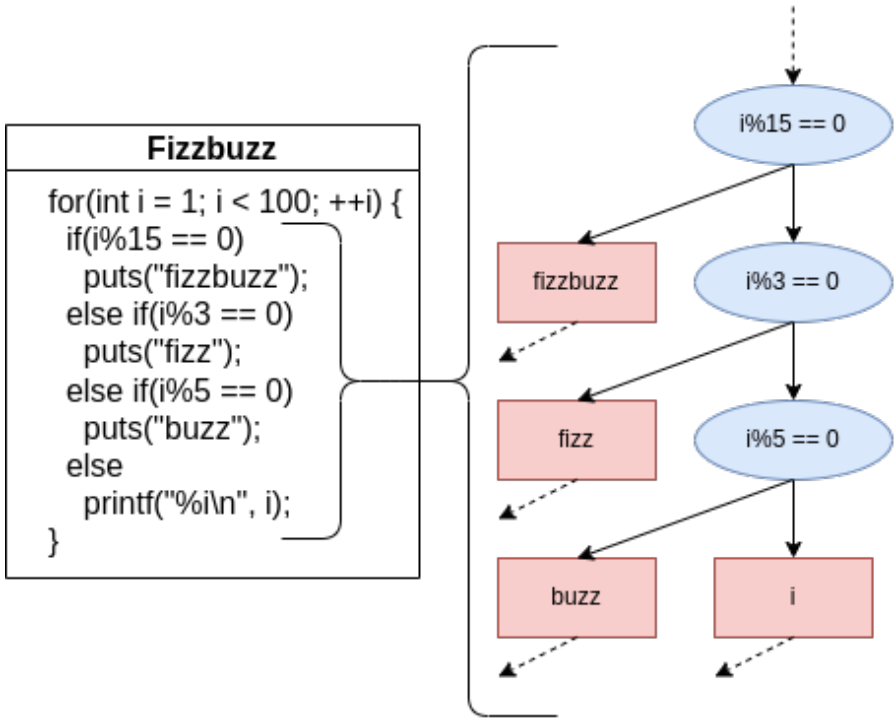

Figure 2.4: An example of a branching code in a tree

or performance issues across a wide range of input conditions without having to enumerate and test each one individually. While powerful, symbolic execution is computationally intensive and can face challenges like path explosion, where the number of possible execution paths grows exponentially with the complexity of the program.

2.4.3 Concolic Execution

Concolic execution, a hybrid approach combining concrete and symbolic execution, aims to mitigate some of the limitations of both methods. In concolic execution, the program is run with specific concrete input values, like in concrete execution, but at the same time, it tracks symbolic constraints derived from the execution path taken. By analyzing these constraints, concolic execution tools can systematically generate new concrete inputs that will explore different paths through the program, combining the depth of symbolic analysis with the actual values from concrete execution.

This approach allows for more efficient exploration of the program's execution space, making it possible to uncover subtle bugs or vulnerabilities that might not be evident

through conventional testing. Concolic execution has proven particularly useful in software testing and verification, providing a balance between the thoroughness of symbolic execution and the directness of concrete execution.

2.5 Verifier

In this section, we will go step by step and explore the tools that are used to build the verifier.

2.5.1 Theorem Provers

Theorem provers are computer programs that assist in proving mathematical theorems by formal methods. The core idea behind theorem proving is to represent mathematical statements and proofs as formal structures that a computer can manipulate. Theorem provers can then be used to check the validity of these proofs or even to automatically generate proofs for certain propositions within a given set of axioms and rules of inference.

2.5.2 Miasm

Miasm [Des12] is a framework primarily designed for reverse engineering and binary analysis. It features tools such as a disassembler and a symbolic execution engine. The framework operates by taking advantage of the features of the symbolic execution engine, where binary code is interpreted in terms of symbolic expressions rather than concrete values.

This symbolic approach allows the theorem solver to evaluate the logical and mathematical properties of the code, solving constraints and proving or disproving theorems about the code's behavior under various conditions. It also paints a clear picture of the transitions of the register and memory states between instructions and basic blocks. The produced symbolic expressions are invaluable when comparing different states as they can pinpoint the expected changes and the actual ones.

2.5.3 Focaccia

Focaccia is a specialized verifier program designed to assess the accuracy of emulators. It uses concolic execution to collect data from a binary and compare it with an emulator's log. At its core, Focaccia works by comparing these two data sets. The first set comprises the memory and register values obtained during a test run on actual hardware, which serves as the benchmark or oracle for expected outcomes. The

second set involves the detailed log produced by the emulator during its operation, which records various actions including register modifications, memory writes, and the current position of the Program Counter (PC). These logs are integral to the verification process as they provide a sequential record of the emulator's behavior, which lets Focaccia find the cutoff point where it starts to behave differently.

The verifier uses the Miasm [Des12] reverse engineering framework for breaking down the original binary code into symbolic expressions for each operational step, transforming the instructions into a more abstract and analyzable form. These symbolic expressions represent the ideal state changes that should occur step by step according to the software's design. Then Focaccia does a detailed comparison between these symbolic expressions and the actual state changes recorded in the emulator's log.

This comparison is the focal point of the verifier as it highlights any discrepancies between the expected behavior (as defined by the symbolic expressions) and the actual behavior observed in the emulator. Discrepancies signal potential bugs in the emulator, indicating that the emulator's reproduction of hardware behavior is not entirely accurate.

3 Overview

In this chapter, we aim to provide a clear overview of our project, outlining its progression, design goals, and its key components. We'll begin by discussing the shifts in direction that the project has taken, due to our new insights. This reflection is quite important as we have come to notice that our first assumption was erroneous. Next, we will talk a bit bout the design goals when extending the verifer. Following this, we will give an overview of how the whole verifier/reproducer combination works along with how individual components function.

3.1 Course of the Project

We started this project by evaluating the accuracy and reliability of various emulators. This entailed research into common emulator bugs along with recreating them. Our main targets were QEMU and Arancini and most of our research was on them.

Our part in this project was focused on developing a program that could produce tests by utilizing symbolic execution on erroneous programs. Initially, we assumed that a significant proportion of the flaws and inconsistencies discovered in QEMU could be attributed to issues within the TCG.

Specifically, we suspected that these bugs were a result of erroneous execution paths that led to incorrect jumps within the code. To address this, we planned to use symbolic execution to construct a tree. This tree was intended to serve as a map, guiding us through the execution path and therefore identifying these incorrect jumps.

Through this method, we had hoped to:

(A) Find the shortest path to the error

(B) Recreate the erroneous program by changing the inputs and making sure that it follows the aforementioned path

Through this method, we had hoped to automate test generation and simplify finding bugs in emulators. However we came to notice that most of the bugs stemming from TCG were not because of incorrect jumps, they were in fact caused by wrong implementation of instructions. Because of these findings, our project had a slight change of direction.

Because of our new findings, we stopped concentrating on following the erroneous path and instead concentrated on the offending instruction. Considering that most of the hard-to-find bugs stem not from the general functionality but the edge cases, we came to the understanding that we need to recreate the state where this bug occurs. This meant we needed to sample the memory and registers before the erroneous instruction. We used symbolic execution to find the symbolic expressions that were used in the instruction and then used concrete values to recreate a similar state. By combining these values, the offending instruction, and other code stubs that are used for running code, we managed to recreate a tiny executable that can cause the same bug.

3.2 Design Goals

Our main design goal for the reproducer was to extract bugs in the simplest manner possible. This meant the reproducer should need only the minimal amount of data and the resulting program should be as simple as possible. We hoped to replicate the bugs by utilizing only the provided symbolic traces and concrete values.

We also tried to abstract the environment of the bug from the required assembly instructions. This meant we tried to design the reproducer such that it could produce parts of the environment, for example, stack, registers, or memory layout without necessarily turning it to assembly instructions. We tried to make this information as flexible as possible to facilitate easier analysis.

3.3 System Workflow and Component's Functions

The reproducer is designed as an add-on to the Focaccia. Much of its functionality relies on this verifier. As shown in the figure 3.1 all the inputs for the reproducer come from it.

3.3.1 Inputs

Both the verifier and the reproducer depend on the same two inputs, namely the emulator log and the binary. However, there are some peculiarities for both of them. The first one is the emulator log, which can be structured in any way, as long as it contains the necessary information. These logs should either document every change in memory and register values based on the program counter PC or provide complete snapshots of memory and register states at each PC. These are necessary as Focaccia relies on these logs to accurately replicate the emulator's actions.

Also, the logs we use must come from binaries that are statically linked, not dynamically linked. This ensures that the program runs with the same set of instructions, regardless of the environment it's executed. For instance, discrepancies in the C Standard Library (clib), whether due to different libraries or versions, can lead to varied instructions. Such variations would cause the traces to not align, resulting in a trace output filled with errors unrelated to the actual faulty instruction.

3.3.2 First Component: Focaccia the Verifier

Focaccia is the verifier that we are using to detect errors in the emulators. It functions by comparing an emulator with an oracle. It uses the Miasm reverse engineering framework and the LLDB debugger from LLVM. There are two inputs required for the verification, namely the emulator log and the binary. Firstly, Focaccia analyzes the instructions in the binary to generate a symbolic trace. This trace maps out all modifications in memory, registers, and branches, effectively creating a tree-like structure that outlines potential changes for each instruction. However, this symbolic trace isn't sufficient on its own. The binary also provides concrete values, from which Focaccia takes snapshots. Later these transformations are compared with the emulator log. Then it compares the transformations and finds whether there is a mismatch between different states.

3.3.3 Second Component: the Reproducer

The reproducer is an add-on to the verifier, with its role being to prepare the assembly instructions needed for triggering a specific bug. If enabled by the verifier, it will be run after the verification process is done and the erroneous instructions are found. After the verifier finishes collecting the snapshots and the symbolic expressions. It is filtered for discrepancies and if found it is passed to the reproducer as shown in figure 3.1. After the reproducer receives the inputs, it will then try to extract only the necessary data from the snapshot which will be bundled with other assembly instructions. This output is significant as it not only points out which instruction is erroneous but also demonstrates the conditions under which the bug occurs. Ultimately it should make debugging easier by recreating both the environment and the instructions.

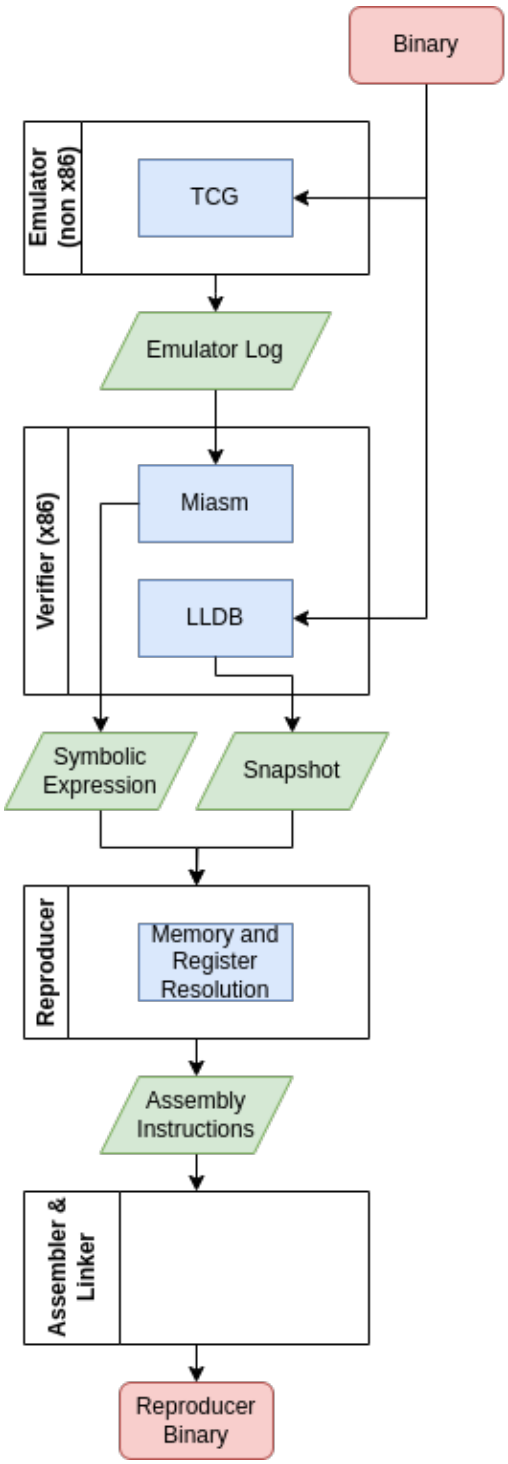

Figure 3.1: Overview of the verifier and the reproducer

4 Findings

In this chapter, we will discuss our findings on bugs related to accelerators and TCG. We will begin by examining the distribution of bugs in QEMU to understand the impact of accelerator bugs on development. Then we will talk a bit about bugs caused by accelerators, followed by a list of bugs found in different accelerator target architectures. Special attention will be given to x86 architectures as targets. Additionally, we will delve into bugs that specifically occur on Arm CPUs when running x86 binaries.

Before we start, it is important to note that this survey is based on data from QEMU's GitLab repository [Con]. As of this writing, there are a total of 2140 issues, with the oldest one dating back to 2021. Some bugs have been transferred from the previous repository, making it challenging to determine their exact date. Figure 4.1 shows the distribution of relevant bugs for reference.

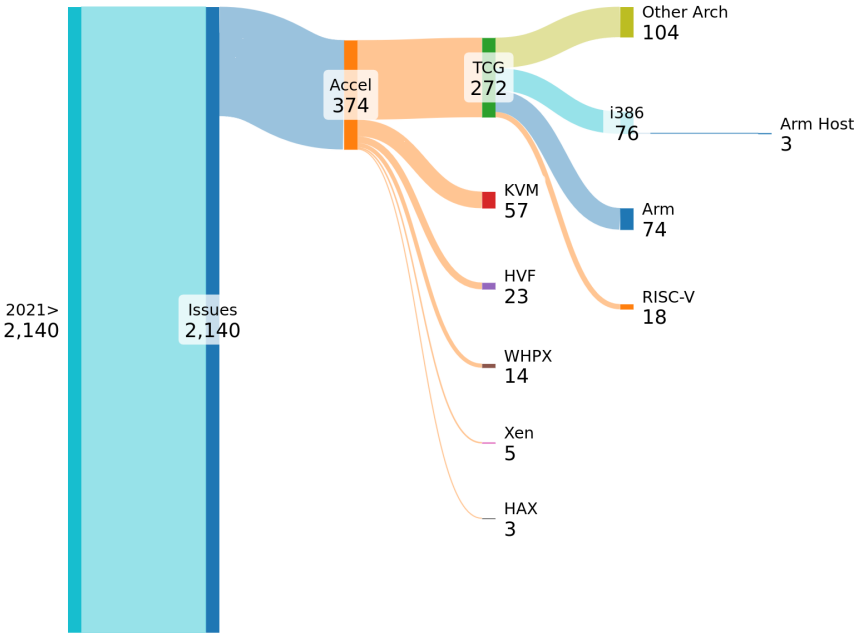

Figure 4.1: Sankey diagram showing the distribution of relevant issues

4.1 Distribution of Bugs in QEMU

As is visible in figure 4.1 QEMU has more than 2000 bugs. The current version of QEMU has 2038147 single line of code (sloc) split between different programming and scripting languages according to the line counting program cloc [Dan]. Drawing from Steve McConnell's research on software metrics [McC93], we can compare QEMU's bug frequency to that of commercially released products, indicating that QEMU has a relatively clean codebase.

Out of all the bugs, 374 are related to accelerators, accounting for about 15% of the total. This suggests that the accelerator-related issues are a significant part of the total amount of bugs. Among these accelerator bugs, the majority, or about 70%, stem from TCG, which is expected given that TCG is the primary and most widely used accelerator. KVM related bugs make up another 15%, with the remaining 15% spread across other accelerators.

Regarding TCG specific bugs, those involving x86 (i386) and Arm architectures are the most common, each being roughly around 30% of the total. This should not come as a surprise since these architectures are being utilized most commonly, leading to extensive usage and testing.

4.2 x86 Translation Errors

In this section, we will go over the bugs encountered when running x86 binaries using the TCG. Most of these bugs are host architecture and operating system independent and tend to arise from incorrect implementation of individual instructions.

We've organized these bugs into six categories based on how they affect the emulation:

- Calculation Error: These are mistakes in instructions that either lead to incorrect calculations or issues with flag registers either being incorrectly set or not set at all. However, they don't usually interrupt the flow of the program.
- Exceptions: These bugs trigger an exception in QEMU, causing the emulation to stop abruptly.
- Errors: Similar to exceptions, these issues cause QEMU to halt the emulation process.
- Segmentation Faults: These occur when the program attempts to access memory areas it doesn't have permission to. While this doesn't happen on actual hardware, it is somehow triggered during the emulation.

- Hardware Problems: These bugs impact external hardware, potentially making it unusable or significantly less efficient.
- Other Bugs: This category includes bugs that don't fit into the other groups, either because their origin is unclear or because they were introduced in newer versions by mistake.

The results of the survey are expressed in the table 4.1. In the next sections, we will mainly focus on calculation errors, since this is where the verifier shines the most. It would be challenging to test the other bugs since they don't necessarily finish their execution normally, therefore leaving the emulator logs incomplete.

Table 4.1: Distribution of TCG errors for x86.

Type	Number	Closed	Open
Calculation Error	18	12	6
Exceptions	6	4	2
Errors	3	2	1
Segmentation Faults	14	10	4
Hardware Problems	1	0	1
Other	33	26	7

4.2.1 Interpretation and Evaluation of the Bug Survey

Other: As is visible in table 4.1 most errors that originate from x86 emulation belong to this category. And these errors, make up nearly 45% of the total amount. Unfortunately, the bugs that cause these problems are multiple and complex therefore they cannot be easily traced to simple instructions. Depending on the complexity the verifier can find the bug and the reproducer might be able to reproduce it. However, it is more than likely that the steps that result in these bugs cannot be simply repeated to pinpoint the actual reason. Therefore it is quite difficult to reproduce them, making our project a bad match against them.

Hardware Problems: There's only a single reported hardware problem, and due to limited information, it's hard to analyze it. Considering that it is a hardware problem errors of this kind are more likely to be a part of the IO and have nothing to do with emulated instructions.

Segmentation Faults: These types of bugs are another frequent issue, accounting for nearly 20% of all bugs. A segmentation fault (segfault) happens when a program attempts to access a nonexistent or restricted area. These accesses can be classified into three categories:

- Read
- Write
- Execute

Generally, the best way to solve these bugs is to trace them and find the location where they caused the segfault. In most cases an emulator will keep running and outputting the emulator log until one of the aforementioned events happens. After the segfault happens the the program will be stopped abruptly and the emulator will stop. This means the emulator log will be stopped before reaching the end. Even though in the case of a segmentation fault the emulator trace will be cut off, considering that we only need to find the address where this happens along with the used instructions the verifier can be helpful.

Although the verifier can prepare the snapshot and the symbolic expression, the current version only finds errors by comparing states. This means it will stop before noticing that the emulator log is short. Theoretically, if the segfault is happening because of an instruction the reproducer should be able to reproduce it. However, in most cases, this is not possible since the reproducer ignores some details. A segfault happens for multiple reasons, either because the memory location doesn't have the required permissions or because it doesn't exist at all. However, neither the verifier nor its symbolic log has any knowledge about a memory location's permissions. Therefore even if we can extract the offending instruction and the used values, without being able to set the memory location's permissions we cannot set an equivalent environment.

In this case, we have two choices. We can handle it just like other memory access instructions where we allocate space on the data section and then use it for reading or writing. Or we can keep the original address. In the first case, we are very likely to not trigger the fault since we use the location that we have declared and made sure it exists. In the second case, we have no idea whether this original address exists and what its value and permissions are. Therefore this method would cause undefined behaviour. Because of these reasons, the reproducer is not a good match for this type of error. The best we can expect to do is to try the first way and see whether we can trigger the segfault consistently.

Errors and Exceptions: Errors and exceptions are relatively common in QEMU, leading to the program stopping. These issues likely stem from the emulator's internal

state, suggesting that alternative debugging methods might be more effective.

Calculation Errors: Finally, we have the calculation errors. They make up slightly less than 25% of total errors, but they are theoretically the most challenging to detect as they involve instructions behaving slightly differently than expected. For example, some instructions might set a bit to a wrong value or change another bit that it shouldn't touch.

The main problem with these instructions is that these values are not necessarily used. This means these programs can go a long way before exhibiting the bug since the difference might have happened multiple instructions ago. Or maybe the result is close enough to the expected answer and therefore it is not found out.

However, in a good case, this instruction just calculates wrong values, and this discrepancy is detected. Our verifier and reproducer combination is a good match for these instructions since it uses symbolic execution we can catch every detail and compare it with the emulator log.

4.2.2 Detailed Inspection of Bugs on Arm

In the following subsections, we will go over specific bugs that only appear on Arm devices. We are inspecting these bugs extra thoroughly because we are interested in seeing whether some bugs appear depending on the host hardware. Out of 76 bugs that we have seen so far, only 3 of them are Arm specific. Considering this fact, we can assume that hardware-specific bugs are rather rare. After taking a closer look at the following subsections, it should be clear that these bugs are not because of the host architecture but because of other reasons.

Issue #1659

The first issue specific to Arm was discovered in an aarch64 system running Darwin. This bug caused the emulator to freeze and enter a continuous shutdown loop. It was traced back to the floatx80_div instruction. Upon closer examination, it was determined that the problem was due to a miscompilation by Clang.

Therefore, this issue is not directly related to the emulator itself but to the compiler, meaning we can cross this issue from the Arm only list.

Issue #2101

The second issue was identified on a system using Fedora Linux as the host. This bug manifests itself when executing the ls command within QEMU. It results in incorrect

output that omits several directories. Due to the lack of more information, it is not possible to find the actual cause.

Issue #2168

The final issue also happens in Linux. This time when running grep on Gentoo Linux inside QEMU, a segmentation fault occurs. Similar to the previous issue, there is limited information available, making it difficult to provide more insights.

5 Design

The following chapter presents the core design decisions that were taken to extend the Focaccia verifier with the reproducer. Extra attention is given to the reproducer interface and the data that is taken from the verifier and how it relates to the whole reproducer. After a brief look into the interface, we will explore the reproducer and go into more detail about how we deal with creating the environment for the reproduced program. This step will go over instructions that we are trying to reproduce, along with setting up the registers, memory, and stack.

5.1 Focaccia Interface

The reproducer which was designed as an add-on comes after the verification process is done. The output of the verification process is a long list of calculations that happen in every step of the program execution.

The result of these calculations is the following:

- pc: The program counter can be used as a pseudo key that maps the calculations to the instruction or the basic block it belongs to. However, it is not a unique key since the same block can be repeated multiple times.
- txl: This is the difference between the current snapshot and the next one. This difference only includes registers.
- ref: These are the reference changes that happen during the execution of the instruction or the basic block. They are in the form of symbolic expressions and play a very important role when creating the reproducer program.
- errors: This is a list of errors found while comparing the emulator log with the symbolic execution. There are multiple levels of errors depending on their severity.
- snap: The snapshots are the concrete values that have been extracted. These include both register values and memory values.

Of the five entries mentioned, only two are essential for the reproducer, and an additional one is useful but not mandatory. Another entry is used for identifying bugs, whereas the final entry is not used at all.

errors: The verification process generates five different types of outputs, with errors being one of the key outputs. They are used to filter through the output list to find exactly which steps have problems in them. They are not necessarily used in the reproducer but rather they are used to pinpoint the places where the reproducer should be used.

txl: The next output, txl, doesn't directly contribute to the reproducer. Since this output only shows the differences between the current and the next snapshot, it doesn't give any hints regarding what instructions are used. Neither does it help find values that belong to the previous state of the execution process. Therefore these values are not used in the reproducer.

pc: This entry is the program counter and it is used to check whether the symbolic expression aligns with the snapshot. We can use it this way since it is supposed to have the same value as the RIP register in x86. Previously we have used it to find the location of the basic block that was turned into the symbolic expression. However, updates to the verifier have made this step redundant, since we can now directly convert symbolic expressions back into assembly instructions.

snap: The snapshot is one of the three inputs that the reproducer takes. They are called the snapshot because they represent the exact state of the CPU, just before the instruction is executed. However, snapshots contain far too much redundant information. Most of these register and memory values are not used when executing our instruction/basic block. We need a way to filter them in order to extract only the necessary values used in our setup. This filtering step can be done by using symbolic expressions.

ref: The final output is the ref which is the symbolic expression that represents the changes our snapshots go through when we execute the next instruction. These symbolic expressions have multiple purposes. Firstly they can be used to extract the assembly instructions that are needed for our program. As mentioned before we used to use the PC to extract the basic block, but with updates to the verifier, it gained the ability to pinpoint the problematic instructions instead of the whole basic block. This meant without getting the proper cutoff point it would be difficult to extract the used

instructions. Instead of trying to guess the actual point the symbolic expression is used to get the exact instructions.

Secondly, the symbolic expression can be filtered to identify which registers are used for which cases. These registers can be later matched with the correct values using the snapshots. However, there is a special case of registers where this is not so simple. If the registers are used to address something in the memory, then we cannot simply copy the original value since these addresses would be wrong. But since the symbolic expressions can show exactly which memory values are changed we can try to match and handle them accordingly.

5.2 Design of the Reproducer

In this section, we will go over the design of the reproducer. Since we have already explained the outputs of the verifier and how they relate to the reproducer we will mainly concentrate on the different parts of the verifier. The inner workings of the reproducer are depicted in figure 5.5 in order to provide a clear picture of the data flow. The creation process of the data section is shown in figure 5.1 while figure 5.2 shows the text section.

5.2.1 Instructions and Basic Blocks

The verifier was designed to reproduce bugs that come up in bigger programs. This meant, that it needed to find the exact instruction that triggered it and run it without changing anything. In our case, as depicted in the figure 5.2 this instruction is supplied by the symbolic expression and directly appended to the text section. No adjustment is needed in this, so we append it without any change.

However, there is a slight problem in handling instructions that way. This way of reproducing the bugs only lets us handle calculations. If we try the same thing with other instructions that change the program flow we would be jumping to places that we don't know and this would have unknown consequences. The same goes for instructions that affect hardware. These special cases are handled in the following section.

5.2.2 Registers

Setting up registers according to their original values is also a very important part of the reproducer. Some of the bugs might be only triggered when the registers are filled with special values. To set the correct environment we have to handle these values correctly. Registers are used for two main purposes. Firstly they are used for general

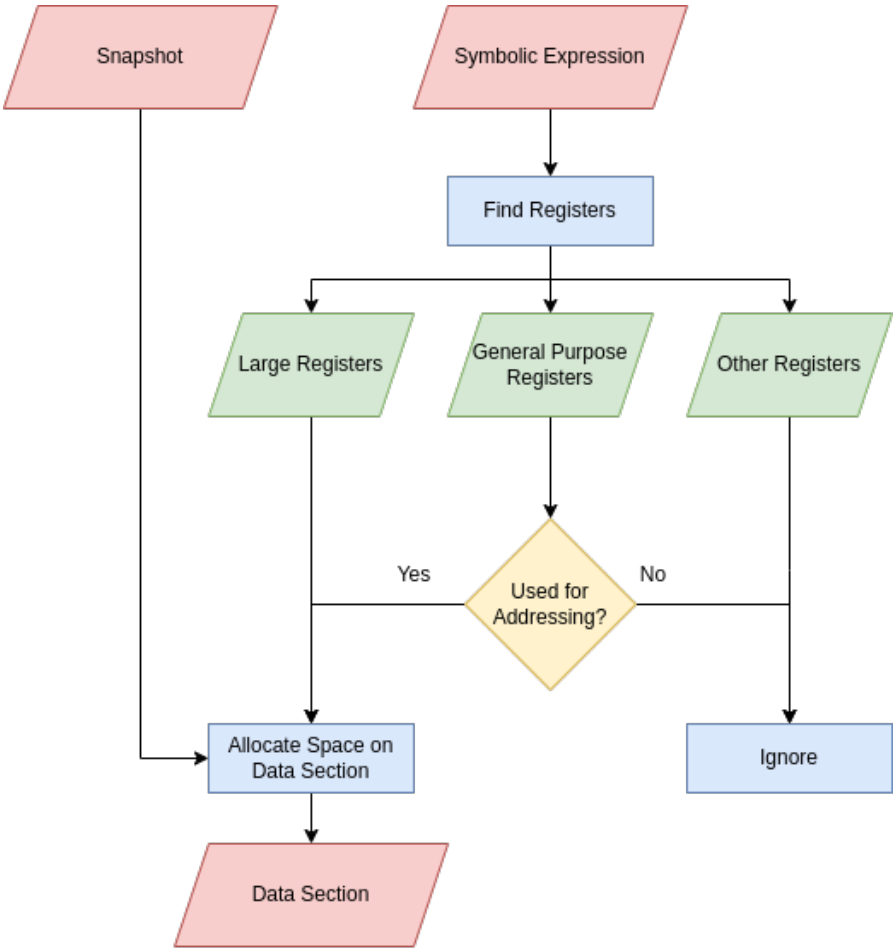

Figure 5.1: Process of creating the data section.

calculation. In this case, we don't need to put too much thought into restoring these values. However, in the second case, namely for addressing memory, we need to be careful about which values we use. The original values are most definitely wrong as they are yet to be allocated. Instead, we need to calculate different addresses where we know we can read from and write to. Handling these registers is done in both the data section and the text section. Figure 5.1 shows where they are used in the data section and figure 5.2 depicts the register setup in the code section.

The values of registers can be split into three categories as shown in figure 5.1. These categories are:

- Large register values

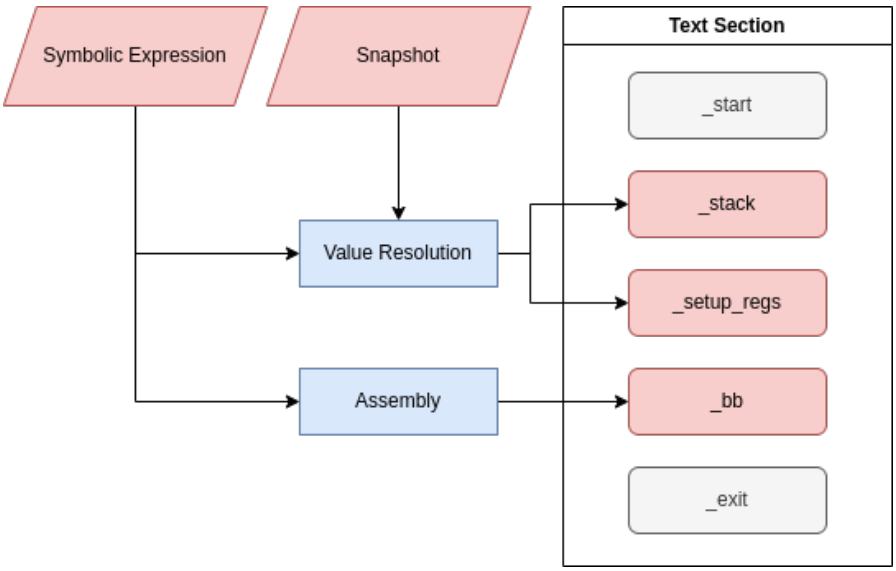

Figure 5.2: Process of creating the text section.

- Addreses
- Other register values

All of these categories have different ways to calculate, with some being rather straightforward and others needing multiple calculations where we leverage symbolic values for resolution. In the following paragraphs, we will go into more detail about how they are calculated.

Large register values: Large registers are the common names we have given to registers that cannot be filled with immediate values directly. In these cases, even though we can easily extract the value from the snapshot we cannot put them into assembly instructions. Instead, we have to save the actual value in the memory as a constant and then use a special move instruction with the constant value's address to move it to the register.

We handle these in two steps. In the first step, we save these register values to the data section. Then when we are writing the values into the registers, we use the aforementioned special move instructions with the addresses. This way we can simply set these large registers.

Addreses: These values are generally inside the 64 bit general purpose registers. They are used by the assembly instructions for read or write operations. Unlike the other

types of instructions, we cannot simply copy them, because they point to memory locations that only exist for the original program. Instead, we need to allocate space in the data section and then change every address that points to the original address with the new one.

When we are doing this there are a couple of details that we need to take care of. Firstly we can have both a read and write to the same memory location. In this case, we need to be aware that this address is used in multiple places and not create a new one.

The second case is about the readable and writeable chunk sizes and how they align. We might need to read and write to a continuous memory chunk, however, the data we receive from the symbolic expression might just point to a random order of reads and writes where they don't even align with the size. In this case, we might order these addresses but the size of the reads and writes might cause a confusion. To prevent problems arising from these, we split all of these memory locations into bytes. This means we always have an address for any byte that is read or written to. In figure 5.3 we can see a comparison of the naive approach and our approach.

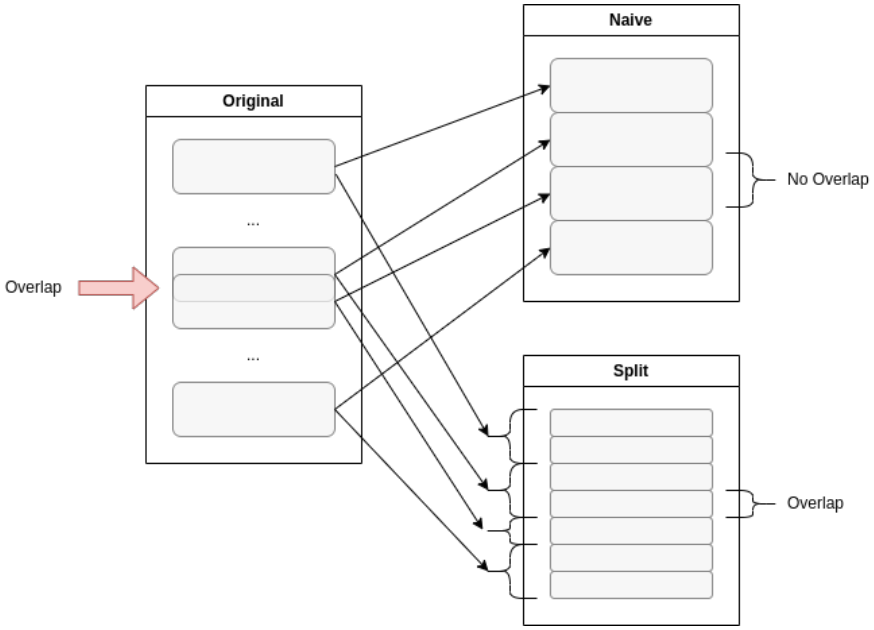

Figure 5.3: Naive approach to data allocation versus our strategy.

Other register values for calculation: Finally, we have all of the other registers. These registers can be set by immediate values. This makes setting them up rather easy since we just need to read the values from the snapshot and use the correct move instruction

according to their type.

5.2.3 Memory

In the last subsection, we have an overview about using addresses but the actual method of finding these addresses is a bit more complicated. We can gather three subsets of symbolic information from the one we have been given. These are:

- Used memory addresses
- Changed memory addresses
- Used registers

By using the first two items we can find which values are used to address memory. However, these would be just the addresses and not the actual values of the registers. In x86 an address can be made out of base, index, scale, and displacement. Out of these 4 parts, the first two of them are registers and the latter two are constants. Only the base is necessary while the other ones can be omitted. This means there are multiple ways an address's symbolic expression can look.

In order to solve this address evaluation we use a shortcut. By evaluating the base register and the whole address we can separate the offset from the base. And when we are preparing the new address we can subtract this offset from the displacement. This will allow us to allocate space in the data section and not think about the offset when we are running the code. Here in figure 5.4, this calculation is depicted. By using this method we can keep the original values for everything except the base and we don't need to think about how the addressing is made.

5.2.4 Stack

Setting the stack state is the final step in configuring the reproducer. This step isn't required for every instruction but is essential for operations that involve the stack, such as push, pop, or any memory addressing that relies on the stack pointer. The method for finding the values in the stack is the same as the memory values, we evaluate expressions related to memory reads or writes. However, we pay special attention to whether these symbolic expressions involve the stack pointer.

Although the approach for identifying stack values mirrors that used for general memory operations, initializing these stack values demands a different method. Unlike other memory operations where the exact location is flexible as long as there's enough space and the registers have the correct addresses, the values in the stack need to be in

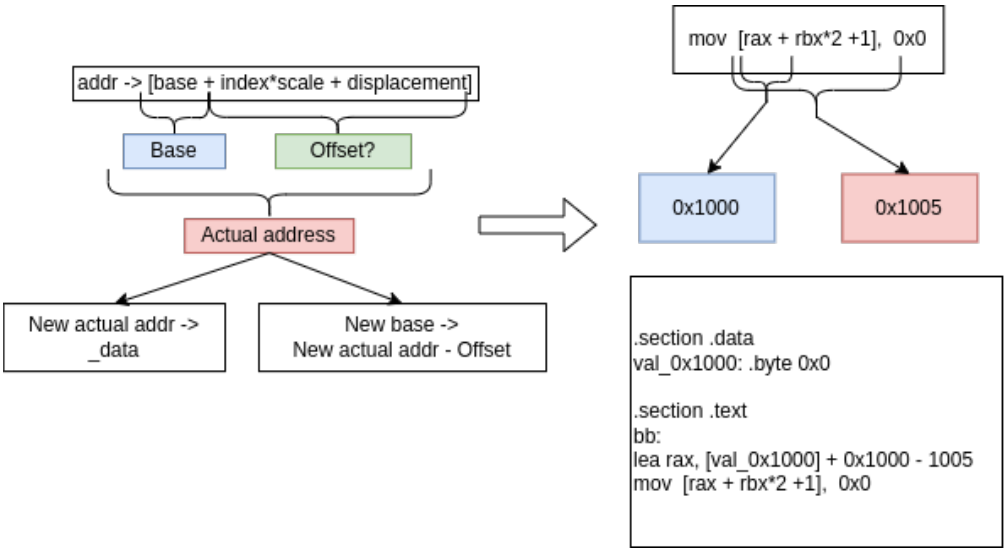

Figure 5.4: Calculating the offset to match the allocated address.

the correct positions. Therefore the values related to the original stack must maintain their relative positions to the stack pointer.

To accurately reconstruct the stack, we first need to gather the correct values. Generally, not all of the values on the stack are used for given instructions. Therefore we cannot recognize them. In that case, we substitute these values with zeros. When collecting these values and adding the padding, it is important to keep in mind that the values are not only before (in higher addresses) the stack pointer but also after (in lower addresses) it. This means we need to put all of the values into the stack and then make sure the stack pointer is in the correct place. We do this by first pushing all of the values to the stack and then subtracting the number of bytes whose addresses are smaller than the original stack pointer from the new stack pointer. By using this method we have recreated the stack.

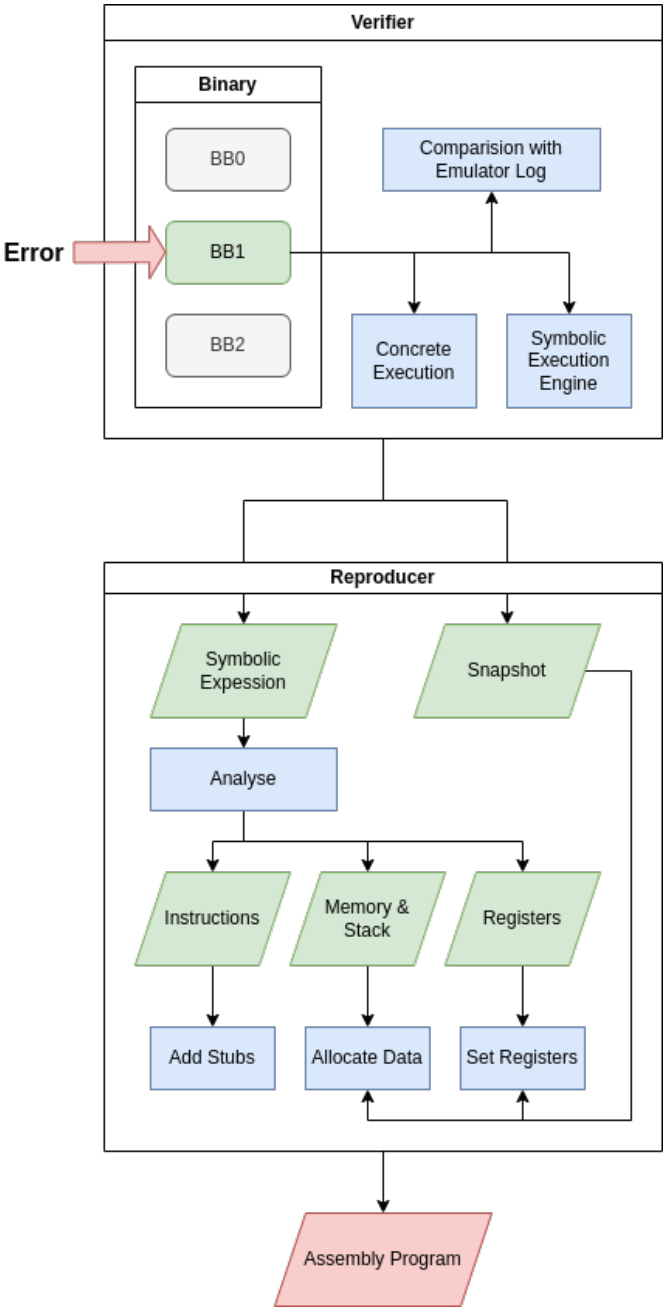

Figure 5.5: Overview of the Reproducer.

6 Implementation

In this chapter, we will go over low-level implementation details and explain what we had to do in order to get the reproducer working. Firstly we will talk about our additions to the verifier. Then we will talk about our Python program and explain the functionality of the two classes we have designed. Finally, we will go over the cases where the verifier might not work as expected.

6.1 Additions to the Focaccia

Focaccia is a full-fledged verifier that can be used to find bugs in emulators that stem from the wrong implementation of instructions. But because it was designed as a verifier it lacks some useful features that would be of help to the reproducer.

Firstly the original verifier would not return the actual state of the program before the execution of an instruction. This of course made it quite difficult to know what was inside the memory and registers. We patched the verifier to return the snapshot of the program.

Our second addition was to the symbolic execution part of the verifier. The symbolic expressions that the verifier can isolate are quite helpful. It can return the following values from a given symbolic expression:

- Symbolic expressions of the used and changed memory addresses
- A list of used registers

Both of these results are quite helpful since we can know which registers need to be restored and which memory locations need to have which values. If we could change anything without considering the underlying hardware mechanisms like paging, permissions, and the actual location of the program in the memory, this might have been enough. However since this is not possible in the scope of this project, we had decided to allocate space on the data section and use it for memory. This meant we needed to find the registers that were specifically used for addressing the memory and change the values of these registers to point to the addresses in the data section.

We made this by adding a function to the symbolic execution program, which would evaluate any given expression except a register. This meant that given a symbolic

expression that points to an address, we could extract the used registers. Then we can be sure that these registers are for addressing memory and handle them as such.

The same mechanism also works for the stack. We filter the symbolic expressions that point to memory for the stack pointer, thereby assessing them for whether they were used in the stack.

6.2 Python Classes

Our reproducer was implemented in the Python programming language and is made up of two classes. The first one which is target agnostic is called ReproducerEntry. This class is used to extract information from the snapshot and the symbolic expressions. The second class is called x86Reproducer. As the name suggests this class is x86 architecture-specific and it is tasked with producing assembly instructions for its architecture. The flow of data can be seen in figure 6.1. As the figure suggests ReproducerEntry splits the snapshot using the symbolic expression and sets the x86Reproducer with the necessary values. After which the x86Reproducer prints the assembly code.

6.2.1 ReproducerEntry

The backbone of our reproducer is the ReproducerEntry class. It lets us find all of the necessary data using the snapshot and the symbolic expressions. The main functionality of this class is shown in figure 6.1, but we will go over the details:

- get_instructions: This function returns a list of instructions that make up the erroneous basic block. These instructions are received from only the symbolic expression.
- get_rw_addr: This function returns a dictionary of reads that will happen during the execution of the basic block and the writes that will happen as a result. Each entry is for a single byte and its key is the address. The reads keep the original values while writes are initialized with zeros.
- get_regs: This function returns the dictionary of registers that are needed for the execution of the basic block.
- get_addr_regs: This function also returns a dictionary of registers. However, these are dictionaries that were only used for addressing memory. And the values of these dictionaries are not the register's values but the actual address that they were used to address.

- get_stack: This function returns two values. Firstly it returns the number of bytes that were subtracted from the stack pointer (either due to push operations or just subtraction from the stack pointer). Secondly, it returns the values that were in the stack. If a particular value was not used, then its place is filled with zeros.

As we have shown the ReproducerEntry is a versatile class that can be used to extract everything that we need from the snapshot and the symbolic expression.

6.2.2 x86Reproducer

This class is the part that produces the assembly instructions. It is specifically designed to produce x86 assembly. It receives all the information from the ReproducerEntry and doesn't directly use the snapshot or the symbolic expression.

In the data section, we use the register dictionary and the read/write dictionary to initialize the values. In the text section, the basic block is combined with stack and register setup code along with start and exit stubs. The stack setup only uses what was returned from the stack values, while the register setup uses both the regular register dictionary and the addresses register dictionary.

6.3 Shortcomings

When trying to understand the shortcomings of the reproducer it is important to keep in mind that it was designed as an add-on to the verifier, which in turn depends on Miasm to work properly. As we have mentioned before our reproducer does have some shortcomings regarding special registers and some instructions. In this section, we will go over these cases and try to explain why they are difficult to deal with.

When thinking about why it is not possible to perfectly replicate bugs, there are two very important points to consider. First of all computer programs run step by step. Each step changes the state of the program, values are written to the memory, and registers are changed. However, this is not all of the changes. When a program is running stack frames are built or destroyed, new pages are added, permissions are updated, and data that is handled by the kernel is changed.

A program is more than just its address space, when trying to understand bugs that stem from emulation we need to keep in mind that the hardware in the background is also a part of the program. To perfectly replicate the state, all of the aforementioned data needs to be changed. However this is not possible, there is no mechanism to replicate all of these except to run the exact program until that point. This means the best we can do is approximate the state that causes the bugs.

The second point is the symbolic expressions. They are quite good at showing state transitions and building a tree for the execution path. But they are an abstraction and therefore lack details that might be contributing to the bugs. They do show what instructions do but they are all on a transactional level. For example, a symbolic expression might show a read operation, but it doesn't necessarily show whether the read address was an IO port or it was on a page that didn't exist. Therefore we use it to guide us the best we can.

6.3.1 Shortcomings of the Reproducer

The reproducer has managed to reproduce bugs and code snippets that generally use simpler instructions that do calculations, however, it still has some shortcomings regarding more complicated instructions that affect the program flow or some registers that are used for controlling the hardware.

Instruction Pointer

In CPUs, the instruction pointer is used to select the next instruction which will be executed. Normally each instruction increments it by that instruction size. However, there are some instructions like jump, call, and return that change the instruction pointer arbitrarily. In these cases, the execution path also changes to a different location.

Most emulators and binary translators work either on an instruction basis or a basic block basis. In both cases, only the last instruction can be one of the aforementioned instructions. Since we can just ignore that last instruction and still get the same calculation, we have chosen to ignore it. This way we can keep the verifier simpler.

Segment Registers

These registers were designed to let the original x86 CPU address more than 64 KB of memory. However, these are currently used for other purposes like thread-local storage and canary-based stack protection. We have left out these registers because changing them is quite likely to cause any program to crash.

Return Address

When we are building the stack for the erroneous basic block, we set the stack by using the values directly from the original snapshot. And we push these values directly after the start section. However, this means that the return address of the function is wrong. It is either zero, if it is not used at all, or it is the original return address. Both of these

values are likely to crash the program if used for returning from the start section but since we directly use the exit syscall it should not have any effect on the program.

Indirect Memory Access

Although our program can find memory access by using reads and writes, we only look for them in the registers. This makes sense since to address a memory location we need to use at least one register. However in cases where a memory location has a pointer to a different address our program cannot recognize them and it will copy the same value, meaning it would be pointing to an unknown location.

This cannot happen on a single instruction since the address needs to be already in the register. In case a basic block is used and this problem arises, the best way to deal with it would be to run the reproducer on a single instruction basis. This method should prevent programs from indirect memory access in a single basic block.

6.3.2 Segmentation Faults

As we have mentioned previously we cannot replicate segfaults even though we have all the necessary information because without the state that happens after the segfault the verifier cannot notice them. This weakness can be solved by adding a segfault error to the verifier that happens when the emulator log is shorter than expected. In that case, the reproducer can theoretically produce a program that can trigger the same segfault.

However, this is not as simple as it sounds. This might not always work because some segfaults happen on special cases like alignment of the data or permissions of the memory section. Unfortunately, the reproducer is oblivious to these things and therefore it is difficult to replicate them.

6.3.3 Shortcomings of the Symbolic Execution Engine

We have decided to add this section here to mention that our project relies on the verifier to function which in turn relies on the symbolic execution engine. This means if the symbolic execution engine has any problems like unimplemented instructions our reproducer also suffers from it.

We have tested our reproducer with many different programs and we came to notice that most instructions that should have caused the bugs were not implemented in Miasm. This had multiple different effects on the reproducer. Sometimes instructions would be translated wrongly and sometimes they would be missing. This has no simple solution except to fix Miasm itself.

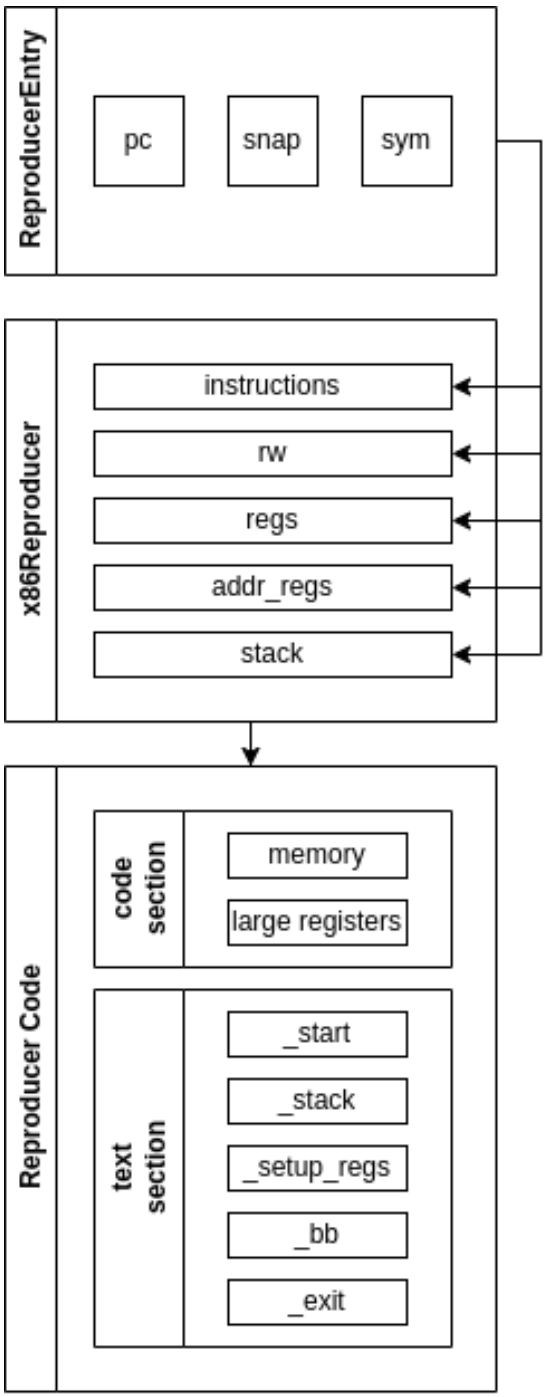

Figure 6.1: Flow of the reproducer including the ReproducerEntry and x86Reproducer classes.

7 Evaluation

In this chapter, we will discuss how we tested the reproducer and how well it met our initial goals. We have selected around 10 bugs from QEMU's GitLab repository, each with programs known to trigger them. We will go over the bugs we have tried to recreate and see how our results stack up against the original programs. We will also go over the foundation of the reproducer, namely Miasm, and explain its importance.

7.1 Experimental Testbed

In the testing phase of our project, our goal was to trigger the same errors as the original code. This meant when we ran the verifier with the reproducer binary, the output should match with the original program. The testing process is made up of 5 steps:

- Obtain the symbolic log from the original program's execution on the guest architecture (x86).
- Launch QEMU on the host machine and capture the emulator's trace.
- Run the verifier to identify bugs.
- Repeat the same process using the code generated by the reproducer.
- Compare the output of the reproducer's binary with the output of the original binary.

If the outputs match, it indicates that we have successfully replicated the bug. There may be minor differences in address values due to how memory is allocated, but the verifier should still pinpoint the problematic instruction. This approach helps us evaluate the effectiveness of the reproducer.

7.2 Example Test Case

In this section, we will go over a concrete example to demonstrate how our program fares against the original reproducers we have found in GitLab. One of the bugs that

we encountered was related to the cmpxchg instruction. Here we will go over this bug and its original reproducer. The original reproducer code is shown in figure 7.1. It was written in C with inline assembly. This program itself is already written as a reproducer, meaning it was designed just to trigger this bug and it doesn't do anything else. If this program is used along with the reproducer the output will look like in figure 7.2.

```c
#include <stdio.h>

int main() {
  int mem = 0x12345678;
  register long rax asm("rax") = 0x1234567812345678;
  register int edi asm("edi") = 0x77777777;
  asm("cmpxchg %[edi],%[mem]"
      : [ mem ] "+m"(mem), [ rax ] "+r"(rax)
      : [ edi ] "r"(edi));
  long rax2 = rax;
  printf("rax2 = %lx\n", rax2);
}
```

Figure 7.1: Original reproducer program for the cmpxchg instruction with inline assembly.

As is visible the output is in assembly instructions and is quite minimal. It just sets the stack and the registers, executes the instruction, and then exits. Since all this program does is trigger the error, it is quite smaller than the original reproducer, which is already supposed to be a minimal example.

Table 7.1: Size comparision of various logs.

Reproducer	Binary Size	Symbolic Trace Size	Emulator Log Size
Original	30,4 KiB	1,4 MiB	484,1 KiB
Automatically Generated	4,8 KiB	23,0 KiB	3,5 KiB

The comparison of various logs can be seen in table 7.1. First of all the binary size of our program is less than one-sixth of the original program. This size reduction can be attributed to not linking the clib to our program since we don't use printf, unlike the original code. This demonstrates how our program effectively eliminates most of the non-essential code from the given program.

This efficiency has additional advantages in program analysis. The symbolic trace of

our program is significantly smaller than the original's. This reduction results from avoiding the execution of unnecessary instructions. Since a symbolic trace consists of changes in registers and memory, minimizing instruction execution means fewer changes occur. Similarly, the emulator log for our program is much shorter than the original's. To quantify, the symbolic trace of our program is about 1/57th the size of the original, and the emulator log is about 1/122nd the size. This illustrates the effectiveness of our reproducer at minimizing the workload in the analysis of results.

7.3 Testing on Larger Scale

In the introduction to this section, we mentioned our attempt to replicate around 10 bugs. We have chosen these bugs because they already had reproducer programs that we could use to test our reproducer. This testing required compiling an older version of QEMU, gathering traces, and comparing the outputs. However, we soon discovered that reproducing most of the bugs was not feasible. While Miasm supports many common instructions, it lacks proper implementation for some of the more niche instructions. However, these instructions are often more prone to errors. As a result, the verifier was unable to identify the faulty instructions; they were either overlooked, misinterpreted, or incorrectly translated. This issue highlighted our program's reliance on Miasm having comprehensive instruction support. Unfortunately, due to these limitations, we were unable to test bugs involving less common instructions.

7.4 Results

To sum up the results of our experiments, we can say that the reproducer works for some cases but its functionality is limited by the verifier, which in turn depends on Miasm. There is no simple way to fix this except to implement the missing operations in Miasm or to use a different symbolic execution engine. However changing the basis of the verifier is not a simple task and even if we were to use a different symbolic execution engine, there would be no guarantee that it would also not miss any instruction.

But when the instructions are implemented and our reproducer works, it can create simple assembly programs that can trigger the same bugs. Since these programs bundle the bugs with a minimal amount of setup they are simpler to test. This should show that our program can be used to improve the development cycle of emulator developers as a tool that can simplify the debugging process.

In summary, our experiments show that the reproducer is effective in certain situations, but its functionality is limited by the verifier, which, in turn, relies on Miasm. The only straightforward solutions are to fill in Miasm's missing operations or switch to a

different symbolic execution engine. However, changing the foundation of the verifier is a complex endeavor, and opting for another engine doesn't guarantee it would cover all instructions either. In that sense only feasible option is to add the missing operations to Miasm. But unless we do this for every instruction, we cannot be sure that it really works. Adding support for every missing instruction is a very difficult task, as it either requires a tremendous amount of knowledge in the x86 architecture or reading the manual, which is also error-prone.

Despite these challenges, when the bugs occur in supported instructions, our reproducer successfully generates simple assembly programs that replicate the original bugs. These programs, designed with minimal setup, are easier to test, demonstrating the potential of our tool to streamline the development and debugging processes for emulator developers.

```
.section .text
.global _start

_start:

_stack:
mov ax, 0x1234
push ax
mov ax, 0x5678
push ax
mov ax, 0x0000
push ax
mov ax, 0x0000
push ax
sub rsp, 0

_setup_regs:
mov rdi, 0x77777777
mov rax, 0x1234567812345678

_bb:
cmpxchg dword ptr [rsp + 0x4], edi

_exit:
mov rax, 60
mov rdi, 0
syscall
```

Figure 7.2: Shortened output of the reproducer for the cmpxchg instruction.

8 Related Work

In this chapter we will explore similar technologies, that can be used along with the reproducer to improve the bug reproduction.

8.1 Increasing the Scalability of Symbolic Execution

One of the biggest problems symbolic execution suffers from is the path explosion. Path explosion is the exponential increase in the number of feasible paths, with increasing code size. It may even result in an infinite number of paths. Path explosion puts a limit on the number of branches in a program. This means larger programs or ones that don't necessarily finish may suffer from it, which would make extracting the symbolic trace very resource-heavy or simply impossible.

Chipounov et al. [Chi+09] suggest a method that they call selective symbolic execution. Their method can transition between symbolic and concrete execution, which means they can designate parts of the code for one of the two execution methods. Correct utilization of this method can negate the effects of path explosion by turning to concrete execution.

This method can be used with the verifier to skip parts of the code that might not be of interest. This would simplify the log gathering process at the start, and when analyzing the code, the comparison would take less time since there would be fewer symbolic traces to go through.

8.2 Implementing Symbolic Execution on Emulators

A different method to gather more information about the execution of emulators might be to directly add symbolic execution to them. Poeplau et al. [PF21] built SymQEMU on top of QEMU by modifying the intermediate representation of the target program before translating it to the host architecture. While Jeon et al. [JMF12] implemented a symbolic execution engine that works with Dalvik bytecode. Both of these methods work on internal representations in order to have a simpler way of dealing with the quirks of the instructions.

Although this method might be able to simplify the symbolic execution, it requires the emulators to implement this method. Therefore it is not something that can be added to the verifier or the reproducer. But both of these tools can be extended to accept these internal representations in order increase the likelihood of discovering bugs.

8.3 Code Coverage of Libraries

In their paper, Gao et al. [**ao2018android**] built a dynamic symbolic execution engine for Android applications that would analyze libraries when they were used and produce a representation of it. They would later run this representation multiple times to create an accurate representation of the symbolic expression that would be context-specific.

Giving extra attention to standard libraries might be useful for our project. So far we have always used programs that were statically linked but, if we could analyze standard libraries separately and combine them with the available program's symbolic log, we could also use the reproducer on dynamically linked programs.

8.4 Instruction Chaining

Yan et al. [YM18] concentrated on test efficiency in their paper. They have developed a method to combine many instruction tests into a single program. This method has the advantage of amortizing overheads. They also added a Feistel network to make each step invertible.

In our case, it might be beneficial to add such a feature to be able to run multiple tests in quick succession. While our program's main goal is to pinpoint bugs from a larger program, QEMU already has a large test suite. Although each QEMU version is supposed to pass these tests before getting published, there might be small bugs that are getting ignored during the test process. Using our program might help detect more bugs and a feature like state chaining might help this process.

8.5 Multi-Level Symbolic Execution

Lastly, Fonseca et al. [FWK18] implemented a new framework called MultiNyx to analyze hypervisors. This project is aimed at finding bugs in processor extensions that are used for emulation, therefore its primary target is not the regular instructions but the special emulation instructions.

These methods can be used to extend the verifier and the reproducer for nested emulation where the outer emulator has to emulate these special instructions.

9 Summary and Conclusion

In this thesis, we have strived to build an add-on for a verifier that checks the correctness of emulators. This add-on, which is also called the reproducer, was designed to replicate bugs that the Focaccia verifier detects. The work on the reproducer mostly went on transforming the output of the verifier and turning it into assembly instructions. This entailed preparing the memory, stack, and registers. We had to deal with symbolic expressions and use them to produce code that could trigger the detected bugs. We have built a generic interface that should be able to extract the aforementioned values and an x86-specific program that could print x86 assembly code.

At the start of our project, we had the incorrect assumption that most of the translation bugs stemmed from incorrect execution path. However, we came to notice that in fact most of the errors stem not from changes in the execution path but rather from incorrect implementation of general instructions. This revelation led us to adjust the reproducer accordingly.

We have tested our reproducer on real bugs and managed to create a binary that can trigger the same erroneous behavior while being tiny compared to the original program. This binary is only one-sixth the size of the original and its symbolic trace is one 57th of the original trace. Even the emulator log is only one 122th of the original log.

We started this project in the hopes that we might create a useful tool that can help emulator developers pinpoint bugs and make reproducing them easier. Although our project hasn't seen any real-world usage we have managed to single out existing buggy instructions and reproduce them. We hope that the reproducer can be of help in discovering bugs. The source code of this project can be found in `https://github.com/TUM-DSE/focaccia` under the reproducer branch.

10 Future Work

As we wrap up this project, there are several directions where it might go for its future development. Among the options, we see three main directions. With the integration of fuzzing techniques, we can generate a wider array of test cases, leading to the discovery of more erroneous behavior within faulty instructions.

A different direction would be to add support for additional architectures like Arm. Given the vast number of devices powered by Arm processors, as we have mentioned in our introduction, improving emulators for these systems could significantly ease the process of running Arm specific programs on personal computers, extending the available software for them.

Lastly, it might be worthwhile to add support for segmentation faults These bugs often introduce complex edge cases that can be challenging to debug. Developing new tools to address these issues would be invaluable, though it demands a deep understanding of the x86 architecture and presents a formidable challenge. Each of these paths not only builds on the foundation we've established but also opens up new opportunities to enhance the functionality and reach of emulators.

10.1 Fuzzing for More Test Case Generation

Fuzzing is a technique used in software testing where random data is used as input. The main goal of fuzzing is to break the software and, therefore find bugs or security loopholes that might not be discovered with standard testing methods. In our case, we can use this technique to preload the registers and memory values with random data to trigger further unexpected errors. This approach can be especially useful because it can uncover hidden issues that might be difficult to notice.

For example, some bugs may only manifest when multiple registers are set to certain values. While our verifier is capable of detecting such scenarios, it might not catch additional bugs associated with the same instruction. However, by using fuzzing we can introduce random inputs into the binaries. And we can potentially trigger these hidden bugs. Thus, fuzzing extends our ability to test the software more thoroughly. Nonetheless, it is important to keep in mind that fuzzing can also take lots of time before finding any useful result and it is not a surefire way to find all the bugs.

10.2 Support for ARM or RISC-V Binaries

The landscape of computing is rapidly evolving, with Arm devices getting more popular and RISC-V emerging as a new technology. As these technologies get even more common, we will need emulators for these architectures. And just like the case with x86, these Arm and RISC-V emulators need to be faithful to their respective architectures. Considering this fact, tools like the verifier and the reproducer would be invaluable for the development and polishing of other emulators.

However, this is not a simple task as each architecture uses its respective assembly instructions. The reproducer has some code regarding memory address detection and registers, that can be shared with other architectures. However, it is still necessary to change the assembly instructions that are produced.

10.3 Adding Support for Segfaults

As we have discussed before segmentation faults are difficult to debug with the current verifier. However, we can add support to the verifier and add detection algorithms to the reproducer to increase the likelihood of reproducing errors. This would firstly entail adding a mechanism to the verifier where if the emulator log is shorter than expected then it is considered a segfault. Then in the reproducer, if the error is designated as a segfault we might try to replicate the exact memory access details. This includes where the memory address belongs, its permission, page size, and alignment. The addition of these details would increase the likelihood of triggering the same bugs.

Abbreviations

TUM Technical University of Munich

TCG Tiny Code Generator

PC Program Counter

clib C Standard Library

TCG Tiny Code Generator

QEMU Quick Emulator

sloc single line of code

ISA Instruction Set Architecture

segfault segmentation fault

List of Figures

List of Tables

Bibliography

[Chi+09] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. "Selective symbolic execution." In: *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*. 2009.

[Con] Q. Contributers. *Qemu Issues*. URL: https://gitlab.com/qemu-project/qemu/-/issues/ (visited on 02/29/2024).

[Dan] A. Danial. *cloc*. URL: https://github.com/AlDanial/cloc (visited on 03/03/2024).

[Des12] F. Desclaux. "Miasm: Framework de reverse engineering." In: *Actes du SSTIC. SSTIC* (2012).

[Deva] T. Q. P. Developers. *Supported build platforms*. URL: https://www.qemu.org/docs/master/about/build-platforms.html#supported-host-architectures (visited on 02/25/2024).

[Devb] T. Q. P. Developers. *TCG Intermediate Representation*. URL: https://www.qemu.org/docs/master/devel/tcg-ops.html (visited on 02/26/2024).

[FWK18] P. Fonseca, X. Wang, and A. Krishnamurthy. "Multinyx: a multi-level abstraction framework for systematic analysis of hypervisors." In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–12.

[Gou+22] R. Gouicem, D. Sprokholt, J. Ruehl, R. C. Rocha, T. Spink, S. Chakraborty, and P. Bhatotia. "Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 2022, pp. 107–122.

[Int23] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. 2023.

[JMF12] J. Jeon, K. K. Micinski, and J. S. Foster. "SymDroid: Symbolic execution for Dalvik bytecode." In: *University of Maryland, Tech. Rep* 7 (2012).

[McC93] S. McConnell. *Code Complete*. Microsoft Press, 1993.

[PF21] S. Poeplau and A. Francillon. "SymQEMU: Compilation-based symbolic execution for binaries." In: *NDSS 2021, Network and Distributed System Security Symposium*. Internet Society. 2021.

[Roc+22] R. C. Rocha, D. Sprokholt, M. Fink, R. Gouicem, T. Spink, S. Chakraborty, and P. Bhatotia. "Lasagne: a static binary translator for weak memory model architectures." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 888–902.

[Sta] StatCounter. *Desktop vs Mobile vs Tablet Market Share Worldwide, Jan 2023 - Jan 2024*. URL: `https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet` (visited on 02/26/2024).

[The21] I. The Radicati Group. "Mobile Statistics Report, 2021-2025." In: (2021), pp. 1–3.

[YM18] Q. Yan and S. McCamant. "Fast PokeEMU: Scaling generated instruction tests using aggregation and state chaining." In: *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2018, pp. 71–83.