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 past decade, there has been a significant shift in computer architectures, with x86 being replaced by newer architectures like ARM and RISC-V. These new architectures are often employed in PCs and cloud servers, and in most cases, they must be able to run software designed for x86 architecture. However, because these newer ISAs are fundamentally different from any other architecture, they cannot run the binaries compiled for the x86. Thus, emulators/virtual machines are used to run these binaries on different architectures.

While these tools offer significant advantages, they are not without flaws. Accurately emulating an ISA 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 since some bugs may take considerable time to manifest or only occur under particular 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 verifier's output, instructions, and data to generate a program that triggers the same bug. In other words, we can isolate 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, 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 plan to use. To wrap up, we will 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 Instruction Set Architecture (ISA)s are gaining importance and spreading to more users. Although x86-based CPUs are still the most common processors in personal and server computers, they are being replaced by ARM and RISC-V CPUs. For example, ARM devices are used in cases where the duration of the operation is more important than the immediate processing power. This is thanks to the ARM CPU's unique ISA, which is more power-efficient than x86. However, since the ISAs are incompatible, software written for x86 cannot be used with these newer computers.

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 is nearly double the number of people. These devices use ARM architectures, so a considerable amount of software is written for smartphones and tablets, but it is not available for desktop computers. Vice versa, the software written for x86 CPUs is also not available to them.

Multiple methods exist to run these programs on hardware with different ISA. Firstly, they can be recompiled if the source code is available and is architecture agnostic. Recompilation is generally the preferred method as it yields the most efficient programs. However, if the source code is architecture-dependent or does not exist at all, then we need to use an emulator/virtual machine to run the existing binaries. These programs emulate a different CPU architecture, enabling the computer to run programs compiled for a different architecture.

## 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, the Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals [Int23] is more than 5000 pages long. Moreover, the ARM Architecture Reference Manual for A-profile architecture [**ARM_manual**] is nearly 13000 pages long. Since these manuals are so big and complex, turning them into software is error-prone. Therefore, we need multiple testing mechanisms to ensure the emulators work precisely like the emulated hardware.

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

## 1.3 High Level Approach

In this paper, we will explain the design of our reproducer, a program that we have implemented to replicate bugs found on emulators. It was built as an add-on for a verifier, which, given an executable and its log from an emulator, can detect discrepancies and show where they differ from the expected execution. This program's output is a symbolic expression, which our reproducer uses to create a minimal program that can replicate the detected bugs. The program that the reproducer outputs includes the memory, stack, and registers, along with the instructions that trigger the bug. These values are combined with start and exit stubs to prepare this program to be assembled and run.

## 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 that 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 is, therefore, a valuable tool for developers. It offers a simple way to identify errors caused by 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. Cloud-based applications and services rely heavily on the seamless operation of virtual machines, and any discrepancies or faults could potentially impact a wide range of users and services. This program benefits emulator developers by improving emulator accuracy and reliability. It 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 actual hardware behavior.

# 2 Background

In this chapter, we will examine the basics of how emulators work. Understanding the process behind emulators is crucial in determining why certain errors occur. Firstly, we will explore how emulators convert binary code so it can run on different types of computers. This conversion process is quite essential 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. Then we will explain what basic blocks are, since they are commonly used when translating code.

Next, we will discuss our primary emulator targets. While the reproducer and the verifier need generic input, there are two main emulators we are 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 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 the host architecture can understand.

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 a software layer. This necessitates binary translation or other techniques.

### 2.1.1 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. Another common problem with static translation is the complexity of the target ISA. For example some x86 instructions change behavior based on the context, such as the operating mode of the processor or the state of certain flags. Accurately translating these instructions requires knowledge of the program's state which is quite often not posible during static translation, preventing total translation of the program. However, static translation is well-suited for scenarios where the complete binary image is available and the execution environment is stable and predictable. The disassemby and translation process of static binary translation is visualized in figure 2.1.

### 2.1.2 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 program's actual execution path, which may vary from run to run.

This method is especially useful in emulating complex software where it is impractical to predict all possible execution paths in advance. However, it has some drawbacks regarding memory and performance. When a dynamic translator is running, extra space is required for the program that is being translated. It needs to allocate space for the disassembled program and the translated instructions. This space may balloon very quickly at the start. The other drawback is the performance penalty of translating instructions before executing them. This drawback is also mostly evident in the starting phase when 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 executions. This caching mechanism is a key factor in mitigating the performance penalty associated with runtime translation, making dynamic translation an efficient

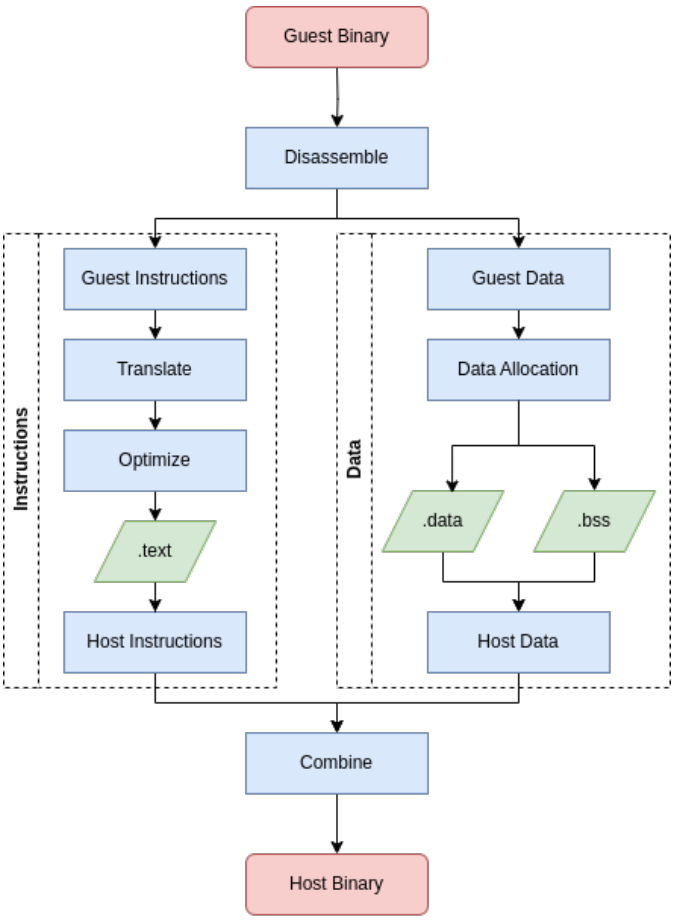


Figure 2.1: Static Binary Translation

and versatile approach for system emulation and virtualization environments.

## 2.2 Basic Blocks

In computer science, basic blocks are code sequences with 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 branching in the middle, and the order of executed instructions are always the same. This simplicity makes basic blocks easy to analyze. Therefore, they are often used in compilers, optimizers, disassemblers, and reverse engineering tools.

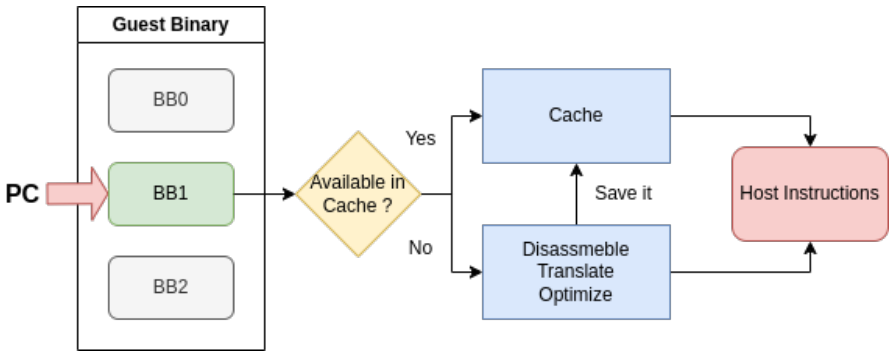


Figure 2.2: Dynamic Binary Translation

## 2.3 Emulators

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

### 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 guest 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

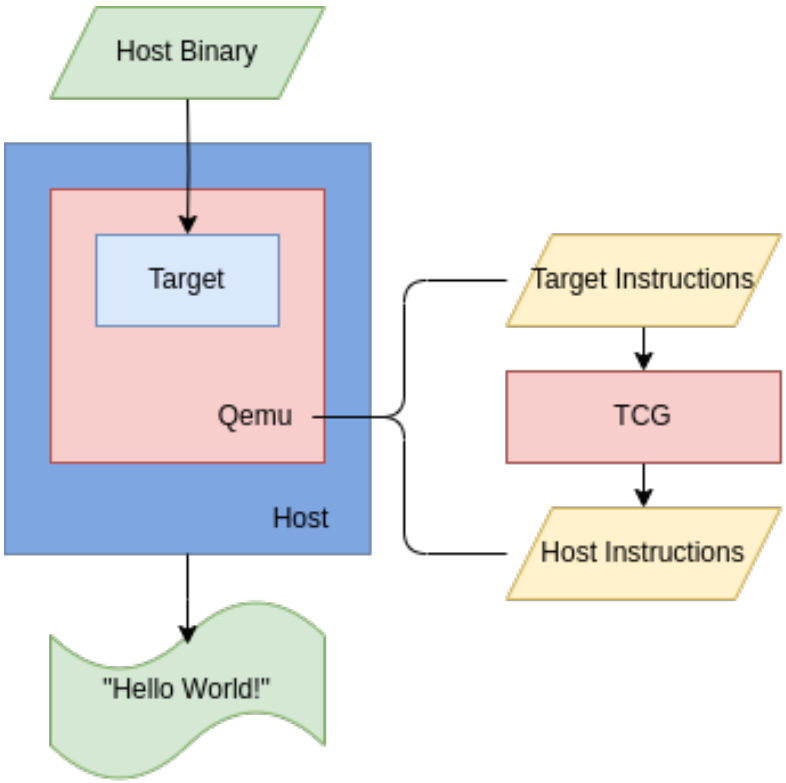


Figure 2.3: QEMU translation process with TCG

- MIPS (little endian)
- PPC
- RISC-V
- s390x
- 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 and adds support for RISC-V systems. It achieves this by combining LLVM, a well-known toolkit for building compilers, with its own custom translation technology.

### 2.3.3 Emulator Logs

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

## 2.4 Execution Methods

### 2.4.1 Concrete Execution

Concrete execution refers to the traditional method of running programs, in which 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 straightforward method mirrors how programs are executed in real-world scenarios, making it intuitive and easy to understand.

Concrete execution is particularly useful for debugging. 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. Figure 2.4 shows that every possible branching instruction adds a new path.

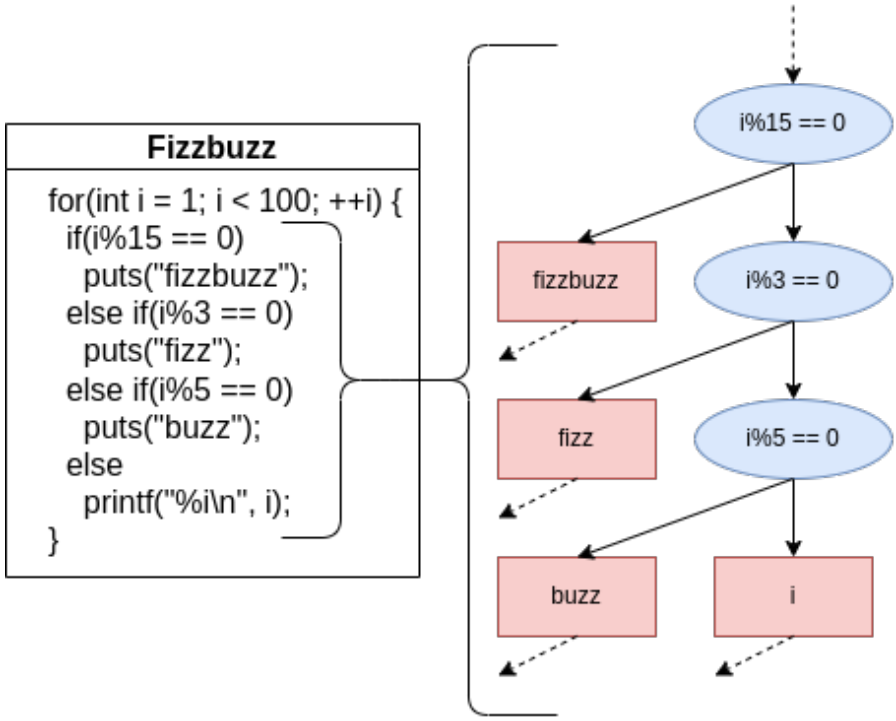


Figure 2.4: An example of a branching code in a tree

This model can then be analyzed to identify potential bugs, security vulnerabilities, 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 program's complexity.

### 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, thus eliminating state explosion in many cases.

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 explore the tools used to build the verifier step by step.

### 2.5.1 Theorem Provers

Theorem provers are computer programs that assist in proving mathematical theorems through 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 prover 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 and actual changes.

### 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 two data sets. The first data 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 data set involves a 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. After collecting these symbolic expressions, they are compared with the state changes recorded in the emulator's log.

This comparison is the verifier's focal point, 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

This chapter provides a clear overview of our project, outlining its progress, design goals, and critical components. We will begin by discussing the shifts in direction that the project has taken due to our new insights. This reflection was essential as we have noticed that our first assumption was erroneous. Next, we will discuss the design goals for extending the verifier. Following this, we will give an overview of how the whole verifier and reproducer combination works and 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 researching common emulator bugs and 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 caused by 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

We wanted to automate test generation and simplify finding emulator bugs through this method. However, we noticed that most of the bugs stemming from TCG were not because of incorrect jumps; they were 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 instructions. Considering that most of

the hard-to-find bugs stem not from the general functionality but the edge cases, we understood that we needed to recreate the state where this bug occurs. This meant we needed to sample the memory and registers before the erroneous instructions. We used symbolic execution to find the offending instructions and then leveraged concrete execution to find the actual values. Combining these two techniques and other code stubs used for running code, we created a tiny executable that could cause the same bug.

## 3.2 Design Goals

Our main design goal for the reproducer was to extract bugs in the most straightforward manner possible. This meant the reproducer should need only minimal 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 abstracting the detected bug's environment from the required assembly instructions. This meant we tried to design the reproducer to produce parts of the environment, such as the stack, registers, or memory layout, without necessarily turning them into assembly instructions. We made this information as flexible as possible to facilitate a more straightforward 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. Any emulator log can be used, regardless of the format, as long as it contains the necessary information. These logs should either document every change in memory and register values based on the PC or provide complete snapshots of memory and register states at each PC. These are necessary as Focaccia relies on these logs to replicate the emulator's actions accurately.

Another important point about the emulator log is whether it comes from statically linked binaries or dynamically linked binaries. They need to come from statically linked binaries, which ensures that the program runs with the same set of instructions

regardless of the environment in which it is 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 not to align, resulting in a trace output filled with errors unrelated to the faulty instruction.

### 3.3.2 First Component: Focaccia the Verifier

Focaccia is the verifier we use to detect errors in the emulators. It functions by comparing the change of states in an emulator with that of an oracle. This oracle is the binary running on its own original ISA. It uses the Miasm reverse engineering framework and the LLDB debugger from LLVM. Two inputs are required for the verification: 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. However, this symbolic trace is not sufficient on its own. The binary also provides concrete values from which Focaccia takes snapshots. Later, these transformations are compared with the emulator log. If any mismatch is detected, the verifier alerts the users about a bug.

### 3.3.3 Second Component: the Reproducer

The reproducer add-on's role is to prepare the assembly instructions 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. The verifier passes the collected snapshot and the symbolic expressions to the reproducer. The reproducer will use the symbolic expressions to determine which values are needed to trigger this bug and extract these values from the snapshot. The data flow is depicted in figure 3.1. After the memory and register values are determined, they will be combined with the erroneous instruction and other code stubs to produce an assembly program that can be executed. Ultimately, the reproducer should make debugging easier by recreating an environment similar to the one that caused the original bug.

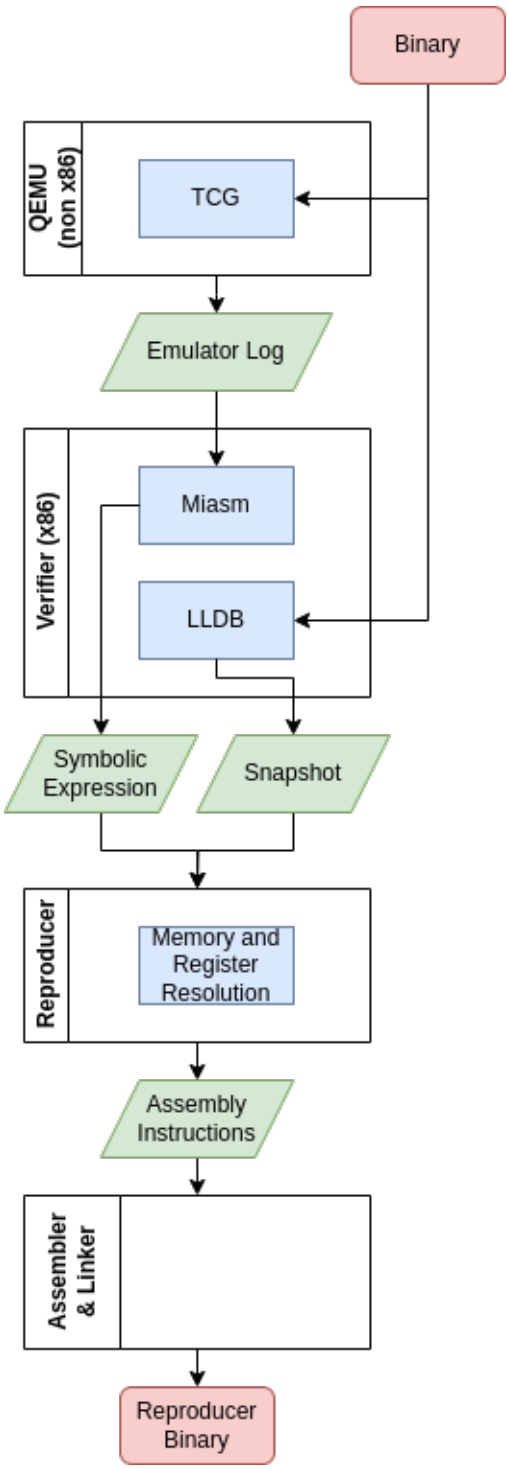


Figure 3.1: Example flow of the verifier and the reproducer with QEMU

# 4 Findings

This chapter 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 discuss 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 in order to explore whether QEMU has additional bugs depending on the host architecture.

Before we start, it is essential to note that this survey is based on data from QEMU's GitLab repository [Con]. As of this writing, there are 2140 issues, with the oldest 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.

## 4.1 Distribution of Bugs in QEMU

As shown in figure 4.1, QEMU currently has over 2000 bugs. According to the line counting program cloc [Dan], the newest version of QEMU has 2038147 single line of code (sloc) split between different programming and scripting languages. 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.

There are 374 bugs related to accelerators, accounting for about 15% of the total. This percentage suggests that the accelerator-related issues are a significant part of the total 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 be surprising since these architectures are most commonly utilized, leading to extensive usage and, therefore, testing.

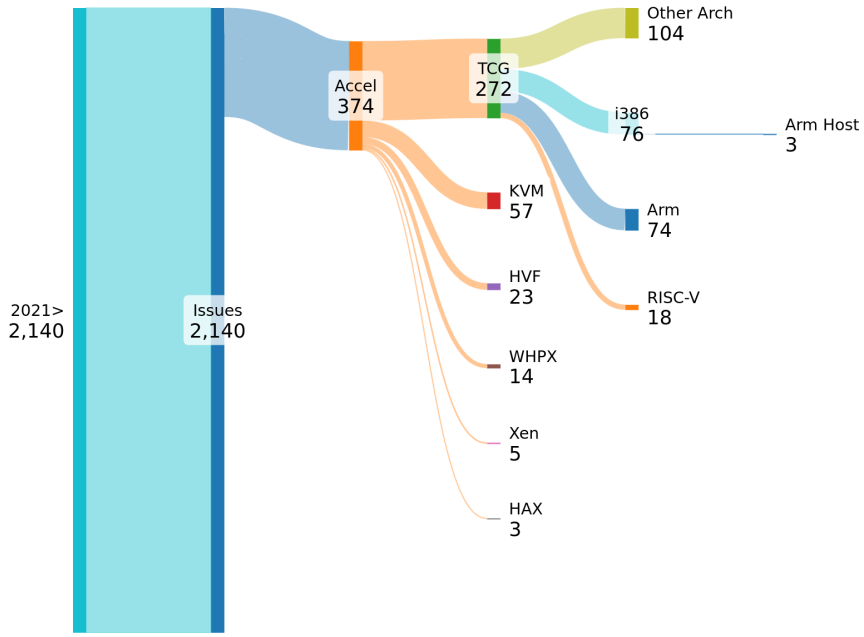


Figure 4.1: Sankey diagram showing the distribution of relevant issues

## 4.2 x86 Translation Errors

This section will review the bugs encountered when running x86 binaries using the TCG. Most of these bugs are independent of host architecture and operating system and tend to arise from incorrect implementation of individual instructions.

We have 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 being incorrectly set or not set at all. However, they do not usually interrupt the flow of the program. They happen because of wrong instruction implementation.
- 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 does not have permission to access. While this does not happen on actual hardware, it is triggered during the emulation.

- Hardware Problems: These bugs impact external hardware, potentially making it unusable or inefficient.
- Other Bugs: This category includes bugs that do not fit into the other groups 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. The following sections will mainly focus on calculation errors since the verifier excels at this area. Testing other bugs would be challenging since they do not necessarily finish their execution normally, leaving the emulator logs incomplete. This topic will be discussed in chapter 10.

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 shown in table 4.1, most errors originating from x86 emulation belong to this category. Moreover, 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. A good example of these types of bugs would be bug #661 [Bon], which only appeared after version 6.1 of QEMU. This bug would cause QEMU to freeze after enabling 5 level paging, and it was traced to missing bit masks that prevented consistency checks for CR4. In this case the verifier should be able to trace the bug to a move instruction which enables 5 level paging. However, since QEMU freezes without transitioning to the next state, the emulator log would be insufficient, and the verifier cannot detect it.

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 repeated to pinpoint the actual reason. Therefore, it is difficult to reproduce them, making our project a lousy match against them.

**Hardware Problems:** These kinds of bugs are likely to result from problems in IO. Therefore, they are out of this project's scope since we are explicitly interested in translation errors.

**Segmentation Faults:** These 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 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 program will be stopped abruptly, and the emulator will stop. This means the emulator log will be stopped before reaching the end. Even though the emulator trace will be cut off in the case of a segmentation fault, 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 does not have the required permissions or does not exist. 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 like other memory access instructions, allocating space on the data section and then using it for reading or writing. Alternatively, we can keep the original address. In the first case, we will likely avoid triggering the fault since we use the location we declared and ensured it exists. In the second case, we have yet to determine whether this original address exists and

its value and permissions. Therefore, this method would cause undefined behavior. 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 should not touch.

The main problem with these instructions is that these values are not necessarily used. They may stay hidden since they might not be used or overwritten by different instructions. This means these programs can go a long way before exhibiting the bug since the difference might have happened multiple instructions ago. Alternatively, the result may be close enough to the expected answer so the bug might not be detected.

However, in a good case, this instruction calculates wrong values, and this discrepancy is detected. Our verifier and reproducer combination matches these instructions well since it uses symbolic execution to model every state transformation and compare it with the emulator log.

### 4.2.2 Detailed Inspection of Bugs on Arm

In the following subsections, we will review specific bugs that only appear on Arm devices. We thoroughly inspect these bugs because we want to see whether some bugs appear depending on the host hardware. Out of 76 bugs we have seen, only 3 are Arm-specific. Considering this fact, we can assume that hardware-specific bugs are relatively rare. After examining the following subsections, it should be clear that these bugs are not caused by the host architecture but by other factors.

#### Issue #1659 [Pol]

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 but to the compiler, meaning we can cross it off the Arm-only list.

**Issue #2101 [Yin]**

The second issue was identified on a system using Fedora Linux as the host. This bug manifests 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 [For]**

The final issue also happens in Linux. This time, a segmentation fault occurs when running grep on Gentoo Linux inside QEMU. Similar to the previous issue, limited information is 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, the data taken from the verifier, and how it relates to the whole reproducer. After a brief look at the interface, we will explore the reproducer and discuss in more detail how we create the environment for the reproduced program. This step will go over the instructions 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 verification process's output is a long list of calculations that occur at every step of the program's 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 vital 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 to identify 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, does not directly contribute to the reproducer. Since this output only shows the differences between the current and the next snapshot, it does not give any hints about the instructions. 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 used it to find the location of the basic block that was turned into a 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 snapshots 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, a symbolic expression representing the changes our snapshots undergo when we execute the next instruction. These symbolic expressions have multiple purposes. Firstly, they can be used to extract the assembly instructions 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 became able to pinpoint the problematic instructions instead of the whole basic block. This meant that extracting the instructions used would be difficult without getting the proper cutoff point. 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 more complex. 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. However, since the symbolic expressions can show exactly which memory values are changed, we can try to match them 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 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, 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 tried the same thing with other instructions that changed the program flow, we would jump to unknown places with 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 bugs might only be 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 calculation. In this case, we do not need to put too much thought into restoring these values. However, in the second case, namely for addressing memory, we must be careful about which values we use. The original values are most definitely wrong, as

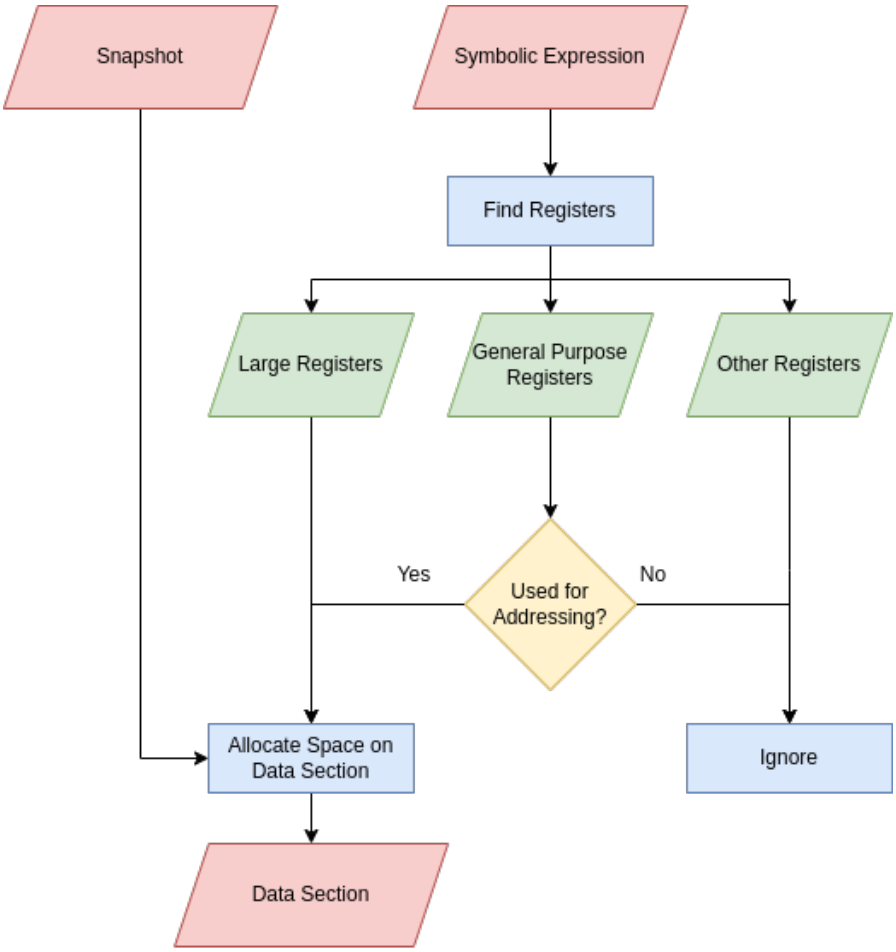


Figure 5.1: Process of creating the data section.

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 and text sections. 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
- Address
- Other register values

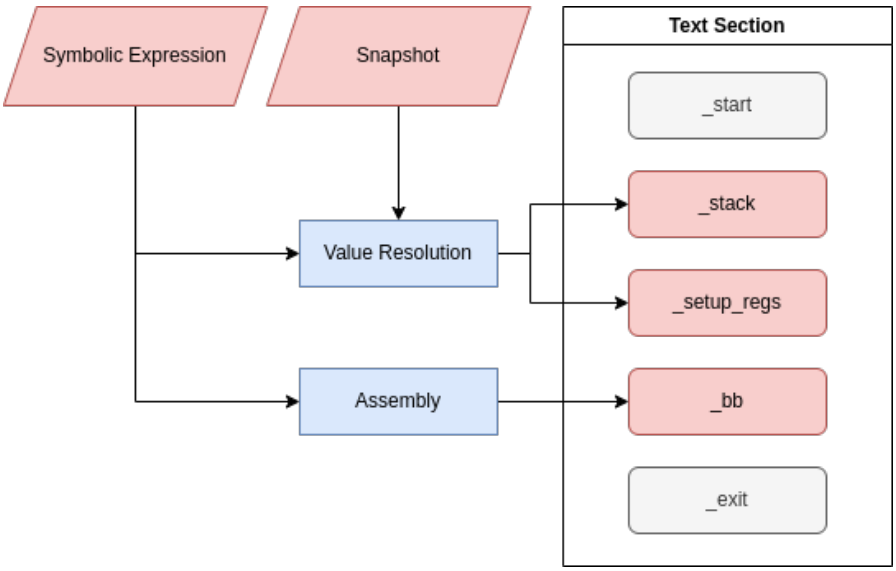


Figure 5.2: Process of creating the text section.

All of these categories have different calculation methods, 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 registers that cannot be filled directly with immediate values. 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 write 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. The assembly instructions use them 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 pointing to the original one with the new one.

When doing this, we need to take care of a couple of details. Firstly, we can both read and write in 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 concerns 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 from the symbolic expression might point to a random order of reads and writes that 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 confusion. To prevent problems from arising, 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.

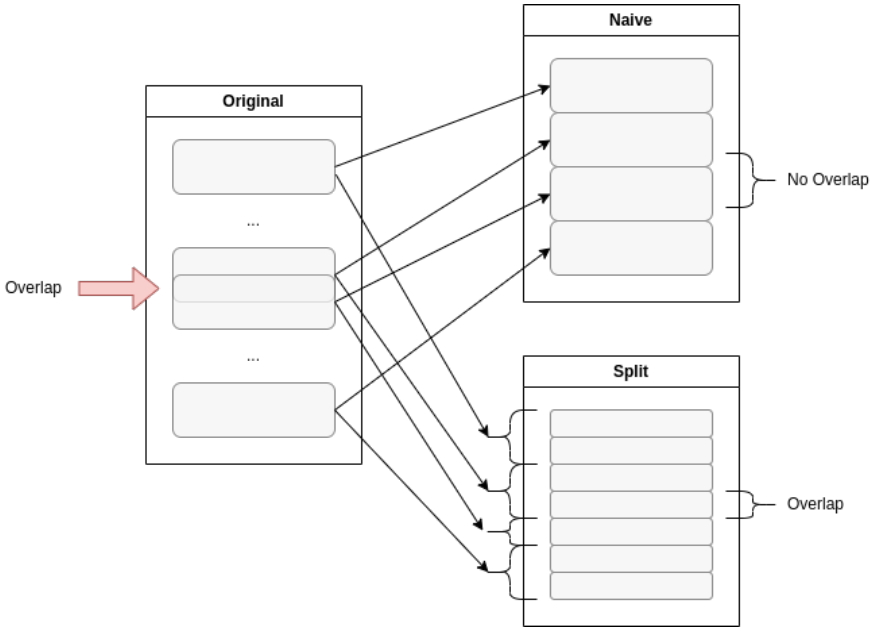


Figure 5.3: Naive approach to data allocation versus our strategy.

**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 four parts, the first two are registers, and the latter 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. We can separate the offset from the base by evaluating the base register and the whole address. When we prepare the new address, we can subtract this offset from the displacement. This will allow us to allocate space in the data section and ignore the offset when running the code. This calculation is depicted in figure 5.4. Using this method, we can keep the original values for everything except the base and don't need to consider 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 the correct positions. Therefore, the values related to the original stack must maintain their relative positions relative to the stack pointer.

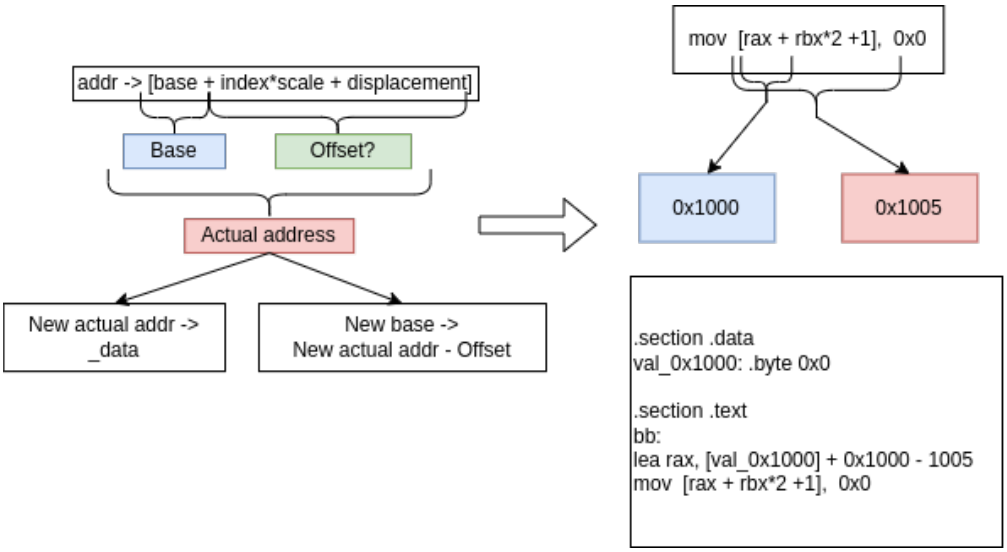


Figure 5.4: Calculating the offset to match the allocated address.

We first need to gather the correct values to reconstruct the stack. Generally, not all of the values on the stack are used for the instructions given. Therefore, we cannot recognize them. In that case, we substitute these values with zeros. When collecting these values and adding the padding, it is essential to remember 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 the values into the stack and then ensure the stack pointer is in the correct place. We do this by pushing all the values to the stack and subtracting the number of bytes whose addresses are smaller than the original stack pointer from the new stack pointer. By using this method, we recreated the stack.

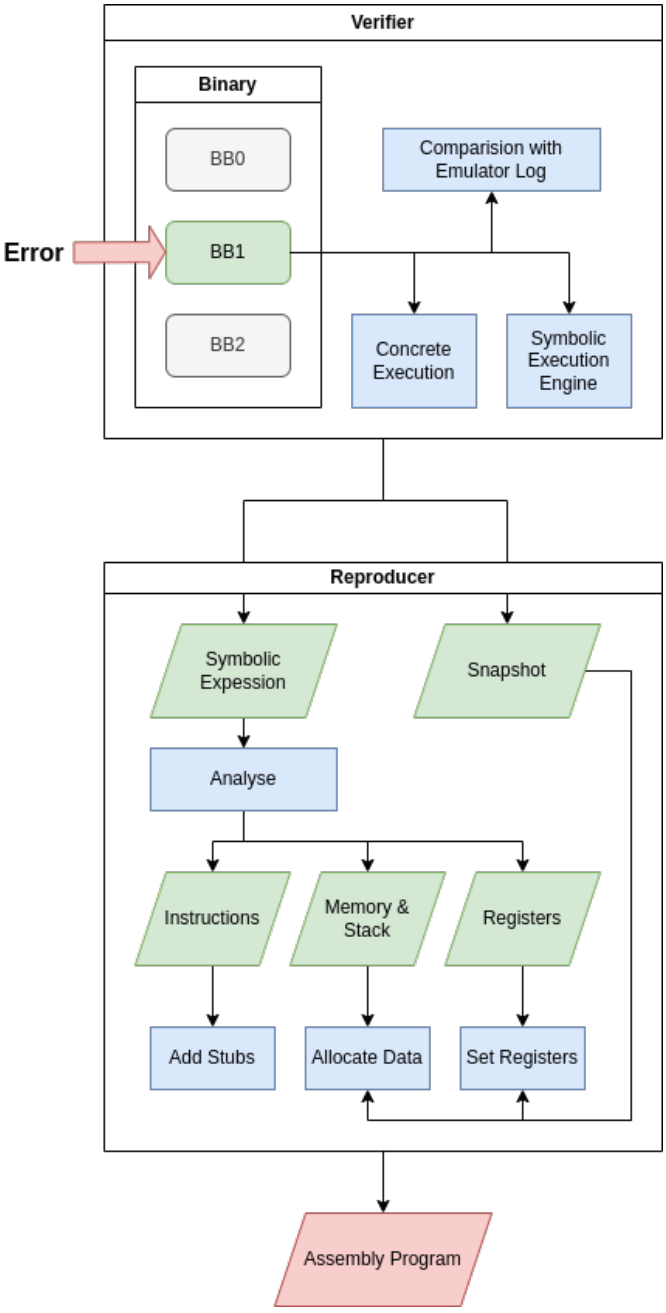


Figure 5.5: Overview of the Reproducer.

# 6 Implementation

This chapter will review low-level implementation details and explain what we had to do to get the reproducer working. Firstly, we will talk about our additions to the verifier. Then, we will discuss our Python program and explain the functionality of the two classes we designed. Finally, we will review 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. However, because it was designed as a verifier, it lacks some useful features that would help the reproducer.

Firstly, the original verifier would not return the actual state of the program before the execution of an instruction. Naturally, this made it quite challenging 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 see which registers need to be restored and which memory locations need 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 within the scope of this project, we decided to allocate space in the data section and use it for memory. This meant we needed to find the registers specifically used for addressing the memory and change their values to point to the addresses in the data section.

We made this by adding a function to the symbolic execution program to 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 address 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 and assess whether they were used in the stack.

## 6.2 Python Classes

Since we were extending the verifier, we used the same programming language it was implemented in, namely Python. Our reproducer is made up of two classes. The first one, which is target agnostic, is called ReproducerEntry. This class extracts 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 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 this, 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; its key is the address. The reads keep the original values, while the 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. Moreover, the values of these dictionaries are not the register's values but the actual address they were used to point to.

- 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 is unused, its place is filled with zeros.

As we have shown, the ReproducerEntry is a versatile class that can extract everything 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 does not directly use the snapshot or the symbolic expression.

In the data section, we initialize the values using the register dictionary and the read/write dictionary. 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 reproducer's shortcomings, it is important to remember that it was designed as an add-on to the verifier, which in turn depends on Miasm to work properly. As mentioned, our reproducer has some shortcomings regarding special registers and instructions. In this section, we will review these cases and explain why they are difficult to deal with.

Two very important points should be considered when explaining why it is not possible to replicate bugs perfectly. First of all, computer programs run step by step. Each step changes the program's state, writes values to memory, and changes registers. However, these are 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 handled by the kernel are 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 part of the program. All of the aforementioned data needs to be changed to perfectly replicate the state. 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 good at showing state transitions and building a tree for the execution path. Nevertheless, they are an abstraction and lack details that might point 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 does not necessarily show whether the read address was an IO port or it was on a page that did not exist. Therefore, we use it to guide us in the best way we can.

### 6.3.1 Shortcomings of the Reproducer

The reproducer has managed to reproduce bugs and code snippets that generally use simpler instructions to do calculations. However, it still has some shortcomings regarding more complicated instructions that affect the program flow or some registers 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's size. However, some instructions like jump, call, and return 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 or a basic block basis. In both cases, only the last instruction can be one of the aforementioned instructions. Since we can 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 will likely cause any program to crash.

#### Return Address

When building the stack for the erroneous basic block, the stack was set by using the values directly from the original snapshot. 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 to return from the start section, but since we directly use the exit syscall, they should not affect the program.

#### Indirect Memory Access

Although our program can find memory access using reads and writes, we only look for them in the registers. This makes sense since we need to use at least one register to address a memory location. 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 must already be in the register. If a basic block is used and this problem arises, the best way to deal with it is 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, making it difficult to replicate them.

### 6.3.3 Shortcomings of the Symbolic Execution Engine

We have added 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 that if the symbolic execution engine has problems like unimplemented instructions, our reproducer also suffers.

We have tested our reproducer with many different programs and noticed 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 mistranslated, and sometimes, they would be missing. There is no simple solution except to fix Miasm itself.

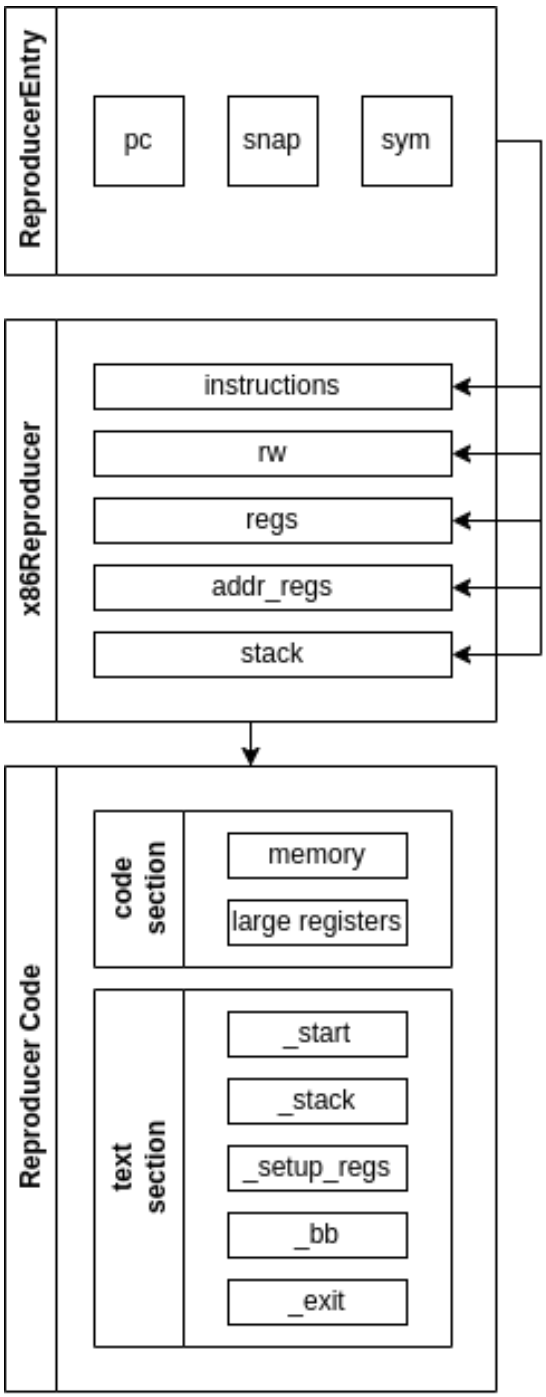


Figure 6.1: Flow of the reproducer including the ReproducerEntry and x86Reproducer classes.

# 7 Evaluation

This chapter will discuss the testing process of the reproducer and how well it met our initial goals. We have selected around ten bugs from QEMU's GitLab repository, each with programs known to trigger them. We will review the bugs we have tried to recreate and compare our results to those of the original programs. We will also review the reproducer's foundation, namely Miasm, and emphasize its importance.

## 7.1 Experimental Testbed

During the testing phase of our project, we aimed to replicate the errors present in the original code. This required ensuring that the output from running the verifier with the reproducer binary was identical to that of the original program. The testing process is made up of 5 steps:

- Obtain the symbolic log from the original program's execution on x86 architecture.
- Launch QEMU on the host machine (ARM) and capture the emulator's trace.
- Run the verifier on the original hardware 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.

Matching outputs indicate that we have successfully replicated the bug. Minor differences in address values may result from memory allocation, but the verifier should still pinpoint the problematic instructions. This approach helps us evaluate the effectiveness of the reproducer.

## 7.2 Example Test Case

In this section, we will review a concrete example to demonstrate how our program performs against the original reproducers we found in GitLab. One of the bugs that we encountered was related to the cmpxchg instruction. It is a very common

instruction that is used in multi-threaded programs. It is used to implement lock-free data structures and prevent race conditions. Figure 7.1 displays the original reproducer code, which used the aforementioned cmpxchg instruction. This reproducer program was written in C with inline assembly. Since it was just written to trigger a bug, it was designed to be minimal. If this program is used 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 shown in figure 7.2, the output is in assembly instructions and is relatively minimal. It sets the stack and the registers, executes the instruction, and exits. Since all this program does is to trigger an 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, unlike the original code, we do not use printf. This example 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 our achieved size reduction, our program's symbolic trace is about 1/57th the size of the original, and the emulator log is about 1/122th 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 ten 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 could not identify the faulty instructions; they were either overlooked, misinterpreted, or incorrectly translated. This issue highlighted our program's reliance on Miasm. Unfortunately, we could not test bugs involving less common instructions due to these limitations.

## 7.4 Results

To summarize the results of our experiments, the reproducer works for some cases. However, 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 instructions.

Nevertheless, simple assembly programs that trigger the same bugs can be produced when the instructions are implemented and our reproducer works. Since these programs bundle the bugs with a minimal setup, they are more straightforward 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 does not guarantee it would cover all instructions either. In that sense, the only feasible option is to add the missing operations to Miasm. However, we must do this for every instruction to ensure it works. Adding support for every missing instruction is challenging, as it either requires a tremendous amount of knowledge of 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 replicating 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 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 do not necessarily finish may suffer from it, making extracting the symbolic trace resource-heavy or simply impossible.

Chipounov et al. [Chi+09] suggest a method 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 reduction in symbolic execution would simplify the log-gathering process. Later, 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 add symbolic execution to them directly. 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 simplify symbolic execution, it requires the emulators to implement it. Therefore, it cannot be added to the verifier or the reproducer. However, both of these tools can be extended to accept these internal representations, increasing 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. They would later run this representation multiple times to create an accurate representation of the symbolic expression that would be context-specific. Although their goal was to automate analyzing ever-changing Android libraries, we might be able to leverage it to analyze different C standard libraries.

Giving extra attention to standard libraries might be useful for our project. So far, we have always used programs that were statically linked. However, 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 run multiple tests in quick succession. While our program's primary goal is to pinpoint bugs from a larger program, QEMU already has an extensive test suite. Although each QEMU version is supposed to pass these tests before being published, minor bugs might be ignored during the test process. Our program might help detect more bugs, and a feature like state chaining might help with this process.

## 8.5 Multi-Level Symbolic Execution

Lastly, Fonseca et al. [FWK18] implemented a new framework called MultiNyx to analyze hypervisors. This project aims to find bugs stemming from processor extensions used for emulation; therefore, its primary target is not software accelerators but hardware ones. It employs selective, multi-level symbolic execution to optimize the process.

Although our target is different, it would still be beneficial to add support for such a tool since emulators can have different backends. It would also give better coverage against bugs.

# 9 Summary and Conclusion

In this thesis, we have strived to build a reproducer add-on for the Focaccia verifier. This verifier checks the correctness of emulators, while our add-on replicates bugs that were detected. The work on the reproducer mostly involved transforming the verifier's output into an assembly program. This process entailed using symbolic expressions to pinpoint necessary values and then setting up the memory, stack, and registers accordingly. We have built a generic interface that should be able to extract the aforementioned values and an x86-specific program that could print the respective 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 noticed that 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 actual bugs and created 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 to create a useful tool that can help emulator developers pinpoint bugs and make reproducing them easier. Although our project has yet to see any real-world usage, we have managed to single out existing buggy instructions and reproduce them. We hope that the reproducer can help discover bugs. The source code of this project can be found in `https://github.com/TUM-DSE/focaccia` under the reproducer branch. The final version can be downloaded from `https://github.com/TUM-DSE/focaccia/releases/tag/alp-berkman-thesis-final`.

# 10 Future Work

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

A different direction would be to add support for additional architectures like Arm. As we mentioned in our introduction, given the vast number of devices powered by Arm processors, 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 builds on the foundation we've established and opens up new opportunities to enhance emulators' functionality and reach.

## 10.1 Fuzzing for More Test Case Generation

By adding fuzzing capabilities to our reproducer, we can preload the registers and memory values with random data to trigger further unexpected errors. This approach can be beneficial because it can uncover hidden issues that might be difficult to notice.

For example, some bugs manifest only when multiple registers are set to specific values. While our verifier can detect 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. Furthermore, 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 a lot of time before finding any useful results, and it is not a surefire way to find all the bugs.

## 10.2 Support for ARM or RISC-V Binaries

The computing landscape is rapidly evolving, with Arm devices getting more popular and RISC-V emerging as a new technology. As these technologies become more common, we will need emulators for these architectures. Moreover, like x86, these Arm and RISC-V emulators need to be faithful to their respective architectures. Considering this, tools like the verifier and the reproducer would be invaluable for developing and polishing 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 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 first entail adding a mechanism to the verifier where if the emulator log is shorter than expected, 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. Adding 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


[Bon] P. Bonzini. *Unable to enable 5 level paging*. URL: https://gitlab.com/qemu-project/qemu/-/issues/661 (visited on 03/13/2024).

[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).

[For] C. Fore. *qemu-x86_64: segfault when running grep on arm64 host*. URL: https://gitlab.com/qemu-project/qemu/-/issues/2168 (visited on 03/13/2024).

[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.

[Pol] D. Poluyanov. *x86 vm fails to stop on Darwin aarch64 when qemu compiled with -O1/-O2*. URL: `https://gitlab.com/qemu-project/qemu/-/issues/1659` (visited on 03/13/2024).

[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.

[Yin] J. Yin. *[qemu-user/qemu-x86_64] run x86_64 'ls /' on aarch64 platform get wrong result*. URL: `https://gitlab.com/qemu-project/qemu/-/issues/2101` (visited on 03/13/2024).

[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.