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

Contents

Acknowledgments iii

Abstract iv

1 Introduction 1
- 1.1 Context 1
- 1.2 Motivation 2
- 1.3 High Level Approach 2
- 1.4 Impact 2

2 Background 4
- 2.1 Binary Translation 4
  - 2.1.1 Dynamic Binary Translation 5
  - 2.1.2 Static Binary Translation 5
- 2.2 QEMU 6
- 2.3 TCG 7
- 2.4 Arancini 8
- 2.5 Concrete Execution 8
- 2.6 Symbolic Execution 8
- 2.7 Concolic Execution 9
- 2.8 Miasm 10
- 2.9 Focaccia 10

3 Overview 12
- 3.1 Course of the Project 12
- 3.2 Design Goals 13
- 3.3 System Workflow and Component's Functions 13
  - 3.3.1 Inputs 13
  - 3.3.2 First Component: Focaccia the Verifier 14
  - 3.3.3 Second Component: the Reproducer 14

4 Findings 16
- 4.1 Distribution of Bugs in QEMU 17

4.2 x86 Translation Errors 17
4.2.1 Interpretation and Evaluation of the Bug Survey 18
4.2.2 Detailed Inspection of Bugs on Arm 19

5 Design 21
5.1 Focaccia Interface 21
5.2 Design of the Reproducer 21
5.2.1 Instructions and Basic Blocks 21
5.2.2 Registers 21
5.2.3 Memory 21
5.3 Shorcommings 22

6 Implementation 23
6.1 Additions to the Focaccia 23
6.2 Data Section 23
6.3 Code Section 23
6.3.1 Stubs 23
6.3.2 Instructions/Basic Block 23
6.3.3 Register Setup 23
6.4 Other Additions to Focaccia 23

7 Evaluation 24
7.1 Reproducing Regular Code Snippets 24
7.2 Reproducing Bugs 24

8 Related Work 25

9 Summary and Conclusion 26

10 Future Work 27
10.1 Fuzzing for More Test Case Generation 27
10.2 Support for ARM or RISC-V Binaries 28
10.3 Adding Support for Irregular Registers and Instructions 28

11 Example 29
11.1 Section 29
11.1.1 Subsection 29

Abbreviations 31

List of Figures 32

List of Tables **33**

Bibliography **34**

1 Introduction

In this chapter we will introduce our project. Firstly we will give some context to our project, in order to prepare for demonstrating it's necessitty. Following that, we will talk about our motivation and try to explain it's 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 our project 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 getting replaced by ARM and RISC-V CPUs. Nowadays these architectures are continuously developed on. And they are gaining more prominence on the market due to different features. For example ARM devices are being used for more power constrained cases, where compared to processing power, 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 have nearly doubled the number of people. All ofthese devices use ARM architectures, so there is this 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 are 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 recompilled. This is generally the preferred method as it yields the most efficent 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 an indispensable tools for prolonging software life and running them on different devices. At the same time they are incredibilly 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 [plc23] is nearly 13000 pages long. Since these manuals are this big and complex turning them to software is very error prone. Therefore we need multiple testing mechanisms to make sure the emulators work exactly like the emulated hardware itself.

Most common tests are the following: unit tests, integration tests, functional tests, regression tests. All of these test 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. However this method is also not very helpfull as some bugs might only appear in very specific cases.

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 causese the bug. This program, which we will henceforth call the verifer, was built by Nicola Crivellin. Our contribution to this project was to add the verifier the capability to produce assembly instructions that can cause the same bugs.

At the start, we have 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 stems 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 project was designed with a clear goal: to pinpoint the instructions within an emulator that lead to bugs and extract it. By focusing on this objective, the project aims to isolate specific errors within the broader context of a malfunctioning program, making it significantly easier to identify and rectify the root causes of these issues.

This aspect of the project 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 project, 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 project extends beyond the immediate realm of emulator development. As virtual machines and emulators become increasingly prevalent in cloud computing environments, the reliability and accuracy of these systems take on new levels of importance. 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 project not only benefits emulator developers but also contributes to the stability and efficiency of cloud computing platforms. In doing so, it addresses a critical need in the tech industry, helping to mitigate challenging debugging scenarios and ensuring that virtual environments more closely mirror the behavior of their real-world counterparts.

2 Background

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

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

Lastly, we'll 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 advanched 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 type of hardware. For instance, running a binary compiled for ARM architecture on an x86-based system would be impossible without any change. This necessitates binary translation. The technique is quite important in achieving compatibility across diverse hardware platforms, enabling a broader software reach and facilitating the preservation of legacy software on modern hardware.

2.1.1 Dynamic Binary Translation

Dynamic translation, a subset of binary translation, involves translating binary code at runtime, as the program executes. This approach contrasts with translating the entire program, all at once, before execution begins. 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 usefull in emulating complex software where it's impractical to predict all possible execution paths in advance. As shown in figure 2.1 dynamic binary translators tend to work on basic blocks and 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 and versatile approach for system emulation and virtualization environments.

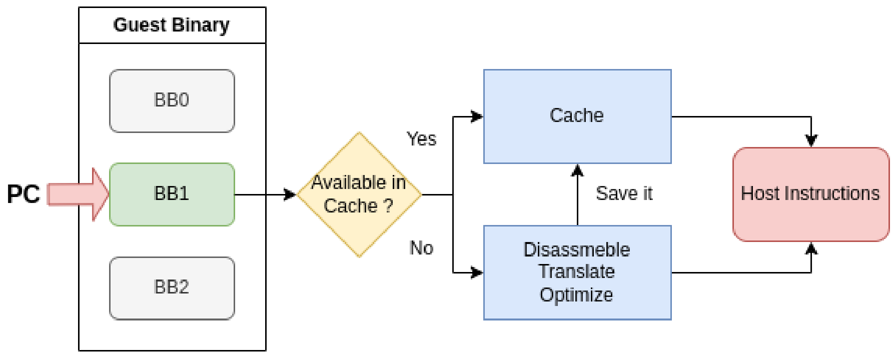


Figure 2.1: Dynamic Binary Translation

2.1.2 Static Binary Translation

Static translation, in contrast to dynamic translation, 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.

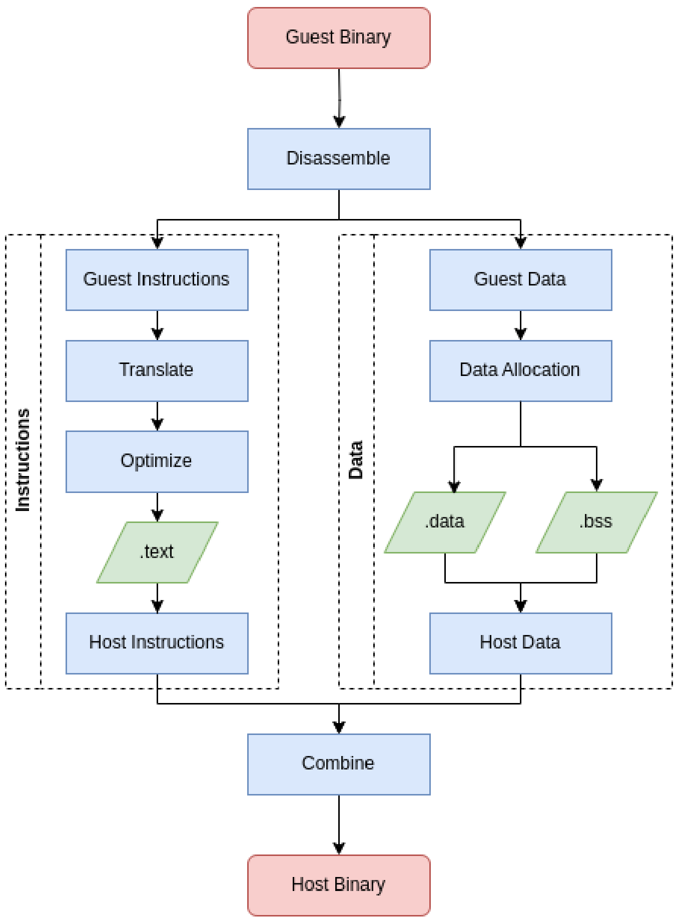


Figure 2.2: Static Binary Translation

2.2 QEMU

Quick Emulator (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 a different architecture. Complementing this capability 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 ISA into the host's architecture.

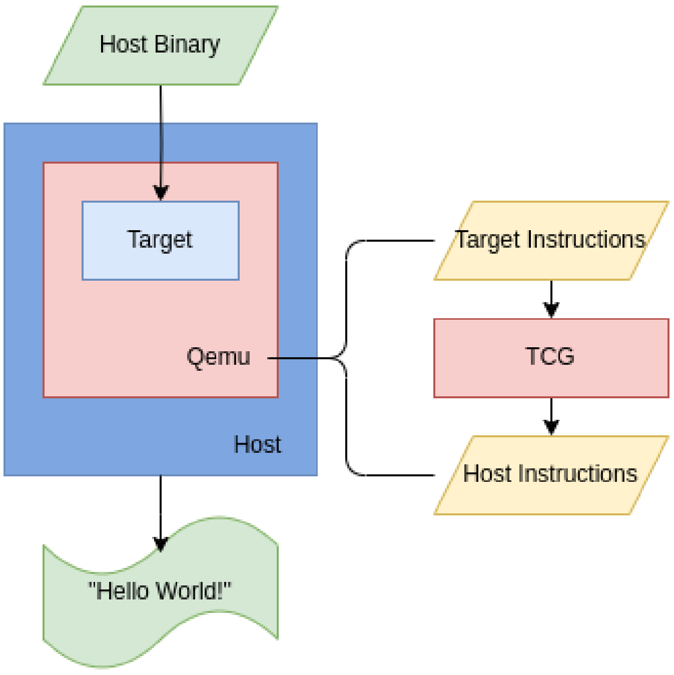


Figure 2.3: QEMU translation process with TCG

2.3 TCG

TCG [Devb] which began as a generic backend for a C compiler was later improved upon 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

- SPARC
- x86

2.4 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.

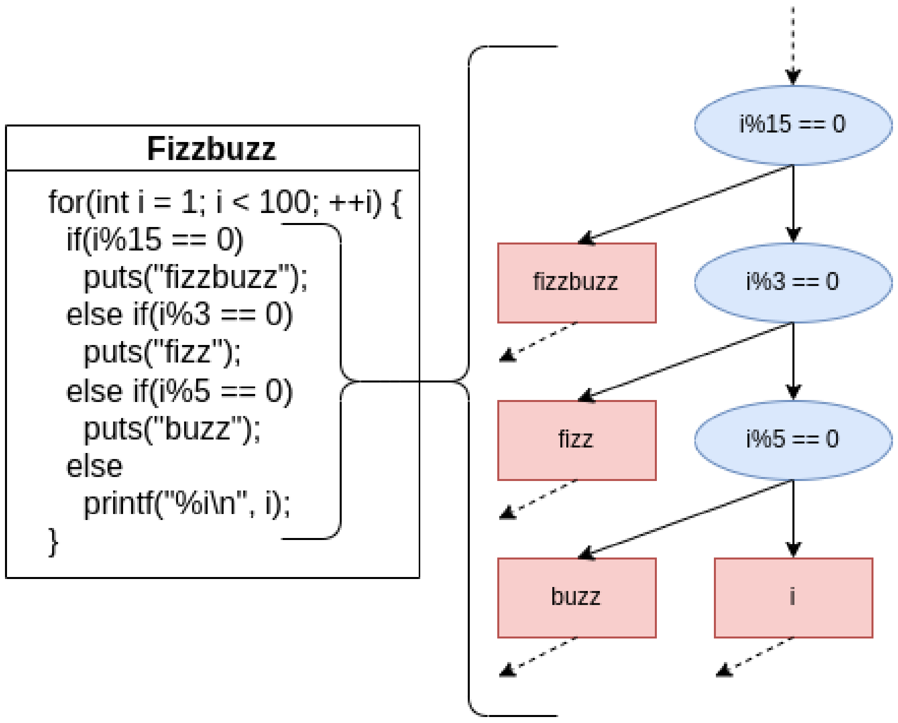
2.5 Concrete Execution

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.

2.6 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.



Fizzbuzz

```
for(int i = 1; i < 100; ++i) {
  if(i%15 == 0)
    puts("fizzbuzz");
  else if(i%3 == 0)
    puts("fizz");
  else if(i%5 == 0)
    puts("buzz");
  else
    printf("%i\n", i);
}
```

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 complexity of the program.

2.7 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.

2.8 Miasm

Miasm [Des12] is a framework primarily designed for reverse engineering and binary analysis. It features tools such as 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 about the transistion 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.

2.9 Focaccia

Focaccia is a specialized verifier program designed with the goal of assessing the accuracy of emulators. It uses concolic execution to collect data from a binary and comapare 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. Focaccia then 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 the its key components. We'll begin by discussing the shifts in direction that the project has taken, due our new insights This reflection is quite important as we have came 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 with the primary goal of evaluating the accuracy and reliability of various emulators. This entailed a research in common emulator bugs along with recreating them. Our main targets were Qemu and Arancini but we designed to make it as generic as possible. In order for other emulators to work with the reproducer they just need to fulfill some basic requirements.

Our part on this project was focused on the developing a program that could produce tests by utilizing symbolic execution on erroneous programs. Initially, our assumption was that a significant proportion of the flaws and inconsistencies discovered in Qemu could be attributed to issues within the Tiny Code Generator.

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 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 the erroneous program by changing the inputs and making sure that it would follow the aforementioned path

(C) Use fuzzing to create multiple tests, and therefore increase the chance to find additional bugs

Through this method, we had hoped to automate test generation and simplfy finding bugs in emulators. However we came to notice that most of the bugs stemming from Tiny Code Generator are 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 are stemming not from the general functionality but the edge cases, we came to the understanding that we need to recreate the state where this bug occured. 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

With the reproducer our main goal was to extract bugs in the simplest manner possible. This means the reproducer shoudn't require any additional data beyond what the verifier supplies. It utilizes the provided symbolic traces and concrete values to replicate the bug. Another key aim is to offer users flexibility. Instead of generating a binary, the reproducer outputs assembly instructions. This allows users to both see the problematic instruction directly or, if they prefer, compile it into a binary using an assembler of their choice.

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 depends on the same two inputs, namely the emulator log and the binary. However there are some pecularities for both of them. The first one is the emulator log, which can be structured in any way, as long as it contains 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 in. 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 erros 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 comapred with the emulator log. Then it compares the transformations and finds wheter 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 by recreating both the environment and the instruction.

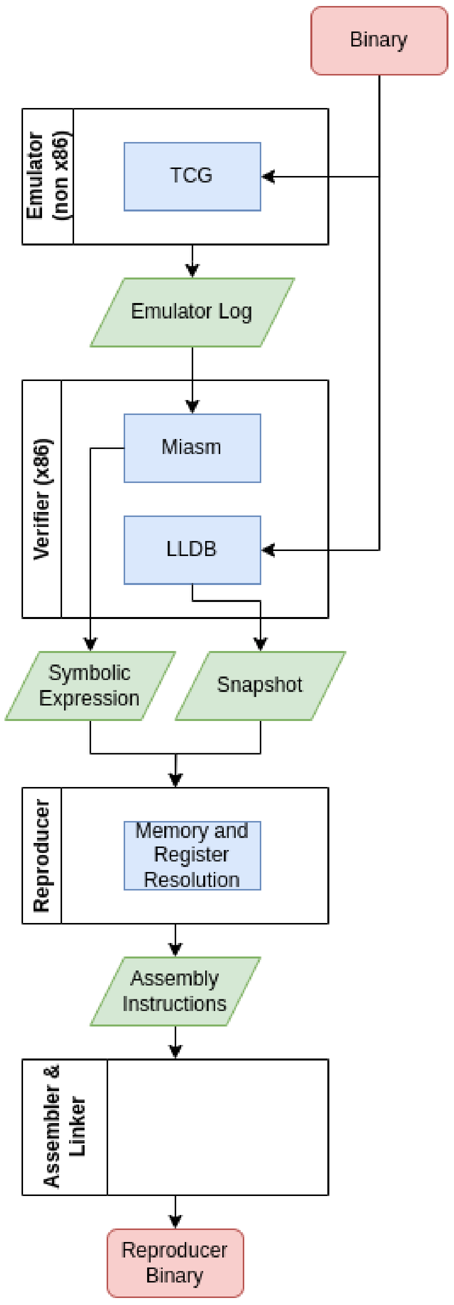


Figure 3.1: Overview of the verifier and the reproducer

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 accelarator 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. 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.

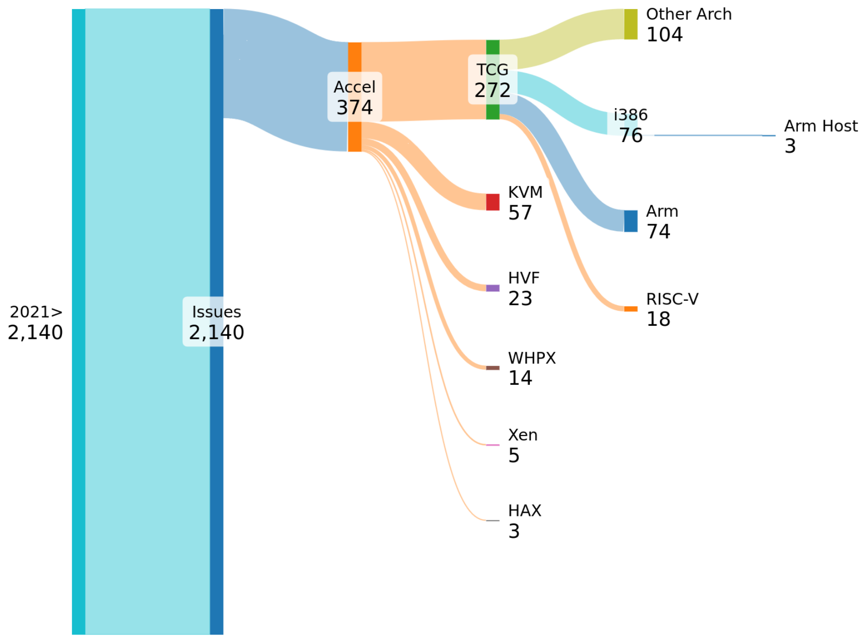


Figure 4.1: Sankey diagram showing the distribution of relevant issues

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 is 2038147 single line of code (sloc). 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 accelerator related issues are a significant part of bugs. Among these accelarator 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 areas.

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 for. 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 normaly, 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

As is visible in table **??** most errors in x86 emulation belong to the other category. These hard to classify bugs, make up nearly 45% of the total. Since these errors are complex, is ti quite difficult to test and reproduce them. Making our project a bad match against them.

There's only a single reported hardware problem, and due to limited information, it's hard to analyse it. Considering that it is a hardware problem it might not have anything to do with emulated instructions.

Segmentation faults are another frequent issue, accounting for 20% of all bugs. These are typically caused by incorrect read and write operations. Generally the best way to solve these bugs is to trace them and find the location where a read or write causes the segmentation fault. Even though in case of a segmentation fault the emulator trace will be cut off, considering that we only need to find address where this happens the verifier can be helpful. However when the reproducer is run it allocates space in the data section. This means it doesnt consider the cases where some addresses might have special meanings. For example writing outside the boundaries will be seen as a regular write and will be handled as such, making it difficult to reproduce the bugs accurately. In these cases the reproducer isn't very helpful.

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.

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 instruction might set a bit to a wrong value or change another bit that it shouldn't touch. The main problem with these instructions are that these values are not necessarily used. Which 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 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 using 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 throughly 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.

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 used 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 verifier. When trying to understand some design choices, it is important to keep in mind that the reproducer was deisgned as an add-on.

5.1 Focaccia Interface

The reproducer comes in the last section of the verifier. This means it is run after verification is done and errors are found. When running the reproducer the higher level interface gets two inputs.

These are:

- Minimum error severity
- Result of verification including:
  - pc
  - txl
  - ref
  - errors
  - snap

5.2 Design of the Reproducer

5.2.1 Instructions and Basic Blocks

5.2.2 Registers

5.2.3 Memory

The reproducer is a direct extension for the verifier. Therefore it is important to first understand how the verifier works, as all of the inputs of the reproducer stem from the verifier.

map the addresses code stubs
more like overall structure

5.3 Shorcommings

In this section we will go over some instruction types that cannot be properly reproduced. These instructions are generally flow control type instructions.

6 Implementation

In this chapter we will go over low-level implementation details and explain what we had to do in order get the reproducer working. These include some minor additions to the verifier, a basic algorithm to find and set the memory values and some assembly stubs both to prepare the state and run the executable. All of these additions are made in python as the verifier is written in it.

6.1 Additions to the Focaccia

Focaccia is a full fledged verifier that can find the bugs in emulators. However because it is desinged to just be the verifier, it lacks some usefull features that the reproducer needs. For example it can return

actual merging some assembly details? what I get from the verifier, symbolic expression? address matching, assembly, syscalls my additions to Focaccia

SOmething about arancini logs and qemu logs

6.2 Data Section

6.3 Code Section

6.3.1 Stubs

6.3.2 Instructions/Basic Block

6.3.3 Register Setup

6.4 Other Additions to Focaccia

7 Evaluation

7.1 Reproducing Regular Code Snippets

7.2 Reproducing Bugs

talk about the error types and if we have recreated an error maybe test all errors from last two years a survey about the bugs

maybe talk about hte change

8 Related Work

Selective Symbolic Execution Fast PokeEMU MultiNyx

9 Summary and Conclusion

Extend focaccia make debugging emulators easier

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 dircetions. Firstly we can add fuzzing capabilities to our verifier. With the integration of fuzzing techniques we can generate a wider array array of test cases, leading to the discovery of more errors within faulty instructions.

A different direction would be to add support for additional architectures. Given the vast number of devices powered by Arm processors, as we have talked about 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 worthwile to add support for irregular registers and instructions. They 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, therefore find bugs or security loopholes that might not be discovered with standard testing methods. In our case we can use this technique to prelode 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 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 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 is 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 faithfull to their respective architectures. Considering this fact, tools the verifier and the reproducer would be invaluable for development and polising of other emulators.

However this is not a simple task as each architecture uses their respective assembly instructions. The reproducer has some code regarding memory address detection and registers, that can be shared with other architectures. But it is still necessary to change the assembly instructions that are produced. It wouldn't be an easy task however it should be possible.

10.3 Adding Support for Irregular Registers and Instructions

And finally, this project can be improved upon by adding support for irregular registers and flow control instructions. With irregular registers we mean the registers which are used either to control the hardware or the registers which control the program flow or the registers that control the stack. These are some of the segment registers, the instruction pointer and the stack pointer. They are not easy to manipulate and making a wrong change would just crash the program. However if the reproducer could be added support for them, then it would be even more valuable.

In the case of flow control instructins we mean the call, return and jump instructins. They are also difficult to reproduce since they change the control flow. however the verifier should be able to show the errors stemming from them.

11 Example

11.1 Section

Citation test [Lam94].

Acronyms must be added in `main.tex` and are referenced using macros. The first occurrence is automatically replaced with the long version of the acronym, while all subsequent usages use the abbreviation.

E.g. `\ac{TUM}, \ac{TUM}` $\Rightarrow$ TUM, TUM

For more details, see the documentation of the `acronym` package[1].

11.1.1 Subsection

See Table 11.1, Figure 11.1, Figure 11.2, Figure 11.3.

Table 11.1: An example for a simple table.

A	B	C	D
1	2	1	2
2	3	2	3

$$\begin{tikzcd}
R_1 \arrow[r, dash] \arrow[d, dash] & R_2 \arrow[r, dash] \arrow[d, dash] & R_5 \\
R_3 \arrow[r, dash] & R_4 &
\end{tikzcd}$$

Figure 11.1: An example for a simple drawing.

[1]`https://ctan.org/pkg/acronym`

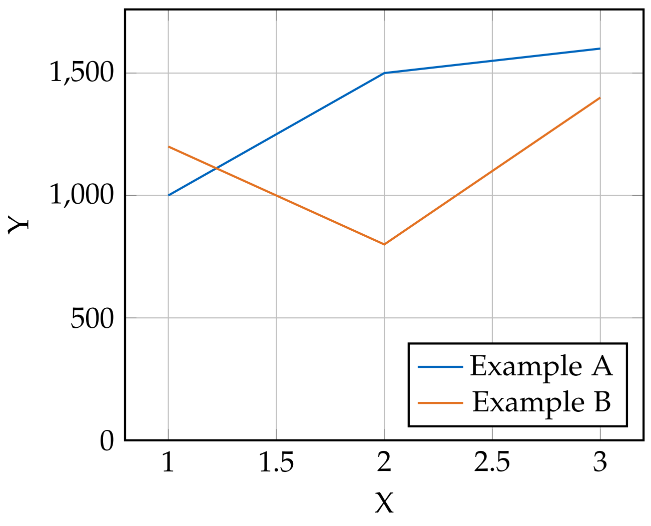


Figure 11.2: An example for a simple plot.

```
SELECT * FROM tbl WHERE tbl.str = "str"
```

Figure 11.3: An example for a source code listing.

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

List of Figures

2.1 Dynamic Binary Translation 5
2.2 Static Binary Translation 6
2.3 QEMU translation process 7
2.4 Branching in symbolic execution 9

3.1 Verifier and reproducer 15

4.1 QEMU bug distribution 16

11.1 Example drawing 29
11.2 Example plot 30
11.3 Example listing 30

List of Tables

4.1 x86 TCG error distribution 18

11.1 Example table 29

Bibliography

[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).

[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.

[Lam94] L. Lamport. *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Professional, 1994.

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

[plc23] A. H. plc. *Arm Architecture Reference Manual for A-profile architecture*. `https://developer.arm.com/documentation/ddi0487/ja/`. 2023.

[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.