SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Execution State Comparison for Emulators using Symbolic Execution

Nicola Crivellin

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Execution State Comparison for Emulators using Symbolic Execution

Vergleich von Programmzuständen für Emulatoren mit Hilfe von symbolischer Ausführung

Author:	Nicola Crivellin
Supervisor:	Prof. Pramod Bhatotia
Advisor:	Sebastian Reimers, Theofilos Augoustis
Submission Date:	15.05.2024

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

Garching, 15.05.2024 Nicola Crivellin

Acknowledgments

I thank my supervisor Prof. Bhatotia for granting me the opportunity to work at his chair, as well as Sebastian and Theo for their direct support on the thesis and their excellent project management efforts. My gratitude goes out to Chris for his unwavering companionship on the grind to my greatest achievement at TUM.

Abstract

Emulators are important pieces of software, though naturally prone to errors and difficult to debug. As the output of emulators are programs, one can use symbolic execution to verify the correctness of that output and find errors in the translation. This thesis proposes *Focaccia*, an automatic testing tool for emulators based on this principle. It designs and implements a fully working prototype and explores advantages and shortcoming of this approach.

Contents

Acknowledgments **iv**

Abstract **v**

1 Introduction **1**

2 Background **2**
2.1 Symbolic Execution 2
2.2 Binary Translation and Emulation 3

3 Overview **5**
3.1 Focaccia's Approach 6
3.2 Focaccia's Position in the Software Testing Landscape 7

4 Design **9**
4.1 The Algorithm: An Abstract Description 9
4.1.1 Definitions 9
4.1.2 Testing Program Equivalence 9
4.1.3 Local Discreteness 11
4.1.4 Comparing Program States 12
4.2 Symbolic Execution 15
4.2.1 A Symbolic Representation 15
4.2.2 Verifying the Symbolic Execution Backend 17
4.3 Working on Complete Programs 18
4.3.1 Program Traces 18
4.3.2 Trace Mismatches 19
4.3.3 Obtaining Instructions 21
4.3.4 Experimental: A Technique for Perfectly Matching Traces 21
4.4 Recording Concrete State 21

5 Implementation **27**
5.1 Tools 27
5.1.1 Symbolic Execution Backend 27

5.1.2 Concrete Execution Tracer 28
5.2 Algorithms 28
5.2.1 Generating Symbolic Expressions 28
5.2.2 Calculating Truth States 30
5.2.3 Verifying a Program State's Correctness 32
5.3 Details 32
5.3.1 Environmental Differences: The Auxiliary Vector 32
5.3.2 Extending to Other Architectures 34

6 Evaluation **35**
6.1 Setup 35
6.2 Results 36

7 Related Work **38**

8 Conclusion **39**

9 Future Work **40**
9.1 The Oracle Problem 40
9.2 Environments 40
9.3 Systematic Test Case Generation 41

Abbreviations **42**

List of Figures **43**

Bibliography **44**

1 Introduction

In the development process of a binary translator, Arancini, we observed that, as is the case with all large-scale software projects, its growing complexity started to hinder maintainability and progress. A particular difficulty specific to the problem domain of binary translators is the fact that both input- and output data are binary—that is, non-human-readable data, which poses significant difficulties during debugging in the development cycle. The most logical way of dealing with binary data is to process it programmatically: our initial hypothesis is that automated testing tools could partially alleviate restrictions of human cognition and facilitate development efficiency.

Most of the current software testing approaches, from unit testing to formal verification, rely heavily on manual work, which is time consuming and prone to mistakes.

This thesis designs and implements an automatic verifier for emulators: *Focaccia*. It traces an emulated program execution at instruction-level granularity and predicts for each instruction the program state that it **should** produce based on the current emulator state and compares it to the state the emulator actually produces, thereby establishing whether the emulator's implementation of an instruction acts on program state in accordance with its specified semantics. To predict program states, Focaccia implements an oracle by harnessing concolic execution. Additionally, this verifier is designed to be practical and usable with minimal setup work; specifically, it is a standalone tool that does not require tight integration into an emulator's code base.

2 Background

2.1 Symbolic Execution

Symbolic execution is a program analysis technique originally developed for software testing in the mid '70s based on previous advances in the field of formal verification [BEL75; Kin75; How77]. The technique's proposition is to execute programs with symbolic values instead of concrete ones, and track their evolution throughout the program together with conditions that branches impose on the existence of certain values. This way, a program execution represents many to all possible values and/or paths at the same time. The results are symbolic equations for the contents of variables or memory, which allow theorem solvers to reason about general properties of the code. Statements like "Assertion X can never fail", "For all $Y > 5$, path A taken instead of path B", or "Statement Z never dereferences a `nullptr`" can be proven with formal certainty.

Listing 2.1 shows a code example. In a usual *concrete* execution, variable `a` is exactly one value and subsequent code uses that value to perform calculations. If, for example, we set `a = 4`, the program calculates the following values: `a = 4`, `b = -1`, `c = -3`. If, on the other hand, in a symbolic execution of the same code, `a` is set to a generic symbol α, a symbolic execution engine produces the following output:

```
a = α
b = α - 5
c = (((α - 5) % 3) == 0) ?  (α + 10) :  (-3).
```

Using theorem provers, one can now make general assertions about the equations.

However, the theoretical vision of perfectly formal symbolic execution turned out to be hardly feasible in practice. King, in one of the first papers describing symbolic execution as a novel formal verification technique, noted in 1976: "Many of the troublesome issues arising in program proving systems also occur in symbolic execution" [Kin76]. The problems he mentioned include array access based on a symbolic index, finite-precision of floating point numbers which inhibits mathematically sound, but practically invalid transformations of symbolic equations for the sake of simplification (e.g., floating-point addition is not generally commutative on computer hardware),

```
a = ?
b = a - 5
if (b % 3) == 0:
    c = a + 10
else:
    c = -3
```

Figure 2.1: Symbolic Execution Sample Code

and theoretical limitations in decidability for theorem provers. The situation has not improved since then—if anything, **more** problems have been uncovered, especially as the complexity of programming languages to which symbolic execution is to be applied has increased massively.

Despite the current problems in state-of-the-art symbolic execution tools like state/-path explosion on real nontrivial programs, finding appropriate models for symbolic memory (similar to King's array dereferencing problem), and side effects of library and system code, symbolic execution can still be used effectively when applied to reasonably restricted scenarios or when enhanced with advanced heuristic approaches [Bal+18]. *Concolic execution* [SMA05; Sen07], in which symbolic execution is selectively guided by concrete state to avoid combinatorial explosions while still achieving reasonably exhaustive results, is one of these more recent developments. Focaccia's algorithm is a customized variant of this concept.

2.2 Binary Translation and Emulation

The idea of binary translation arises when one considers the reality of multiple conflicting instruction set architectures in the world and the desire or even necessity to execute programs on all of them. The natural way to accomplish this is to compile a program for each target architecture, resulting in *native* executables for each, which also makes this approach generate the most efficient form of program execution as it incurs no runtime overhead at all. This works for entirely self-contained programs, but programs tend to depend upon third-party software of which the source code might not be available at all or may perform architecture-specific work so that it is not trivially adaptable to different architectures. Additionally, recompilation and the distribution of multiple binaries associated with it is inconvenient and also costly for very large projects.

Other approaches, which trade off different parts of the aforementioned pros and cons, can be assigned to the rough categories of microcoded emulators, software interpreters, and binary translators [Sit+93]. The first one is concerned with supporting multiple Instruction Set Architecture (ISA)s **in hardware** by translating their instructions to a common microcode language; this issue shall not trouble us here. The latter two are software solutions. Binary translation is further divided into static binary translation and dynamic binary translation [CM; Roc+22]. Static binary translation takes as input binary code of one ISA and compiles it to a different ISA in an offline process ahead of time. It may include a runtime component that maps operating system calls to the target platform. Static translation cannot deal with dynamic branching, self-modifying code, and many other restrictions, but incurs little runtime overhead. Dynamic binary translation, on the other hand, acts in the spirit of an interpreter by translating a binary on the fly during its execution. It handles dynamic branching without problems, but its runtime overhead is higher and optimization options for generated code are severely limited by its narrow, local view of the executed code—though many optimization techniques have been developed over the years [Gua+10; Sun+23; Haw+15; KB13]. Programs that perform dynamic binary translation are frequently called *emulators*.

3 Overview

The value of binary translation technologies has been firmly established [AKS00; Sit+93; Pro02]. These are large programs that additionally deal in an inherently complex problem domain, causing their development to be accordingly difficult. We want to aid this endeavour by **testing** binary translators, particularly dynamic binary translators (emulators).

Testing means to evaluate whether a system meets its originally specified requirements [Jam+16]. Possible specification requirements for emulators might be performance or size of feature set, but the specific ubiquitous requirement for emulators (and really any software) that we want to test for is *correctness*. As formalized in section 4.1.1, the fundamental definition of correctness for binary translators is semantic equivalence between input- and output programs, that is, of the translated program and the translation.

The traditional and most widely used way to test software, called unit testing, requires developers to divide the tested entity into atomic, self-contained units/functions (test cases), collect an as exhaustive as possible list of inputs to that unit, and compare the unit's output for all inputs to their respective expected outputs. This approach is usable and popular, though it carries with it significant shortcomings and difficulties:

- The procedure is not systematic. Ultimately, test quality depends on the developer's abilities and whim—"identifying which code to test" and the *oracle problem* have been reported as major difficulties in writing unit tests [DF14].
- The isolated nature of unit tests means that units have to be isolated in the first place, which is often not trivial.
- The expected value against which a test case's output is checked is not necessarily reliable as it is subject to human error just as much as the code that is being tested.
- Manual work is time consuming.

Many of these problems are amplified by the complexity of an emulator's domain, where the test development procedure could look like the following: We write test cases for each instruction for each ISA that the emulator supports. We let the emulator simulate an instruction for a number of different operands, which we obtain by thinking about classes of equivalent values and choosing an exhaustive list of representatives and corner cases. Then we check for each instruction-operand-combination whether the emulator's state after the instruction has been executed is equal to the state that we would expect it to reach, which we have calculated manually based on our understanding of the respective instruction's semantics.

The issues at every step of this approach are glaring: Expected test truths can easily be wrong, especially when the same developer that implemented the translation also writes the test and thus applies their own faulty understanding of how the instruction should operate to it. Further, complex architectures like x86 can have, in addition to a vast number of different instructions, a gigantic number of possible instruction-operand combinations for each instruction which is unlikely to be exhausted in manually written tests. Additionally, corner cases for inputs are hard to identify, the process is excessively time consuming, and, more important than is often admitted: developers don't want to go through all of this effort—and hence frequently just don't.

Many techniques have been developed to improve this process, especially on the part of **generating** test cases. Fuzzing, where tested units are essentially supplied with randomized inputs, is a popular and successful method for improving test coverage [Zhu+22]. Another technique in this area which also relates to Focaccia, though in a different manner, is symbolic execution. It can be employed as a formal method to generate truly exhaustive lists of test cases by finding inputs to tested functions that cover all possible solutions for generated path constraints [LGR11].

Focaccia, on the other hand, focuses on tackling the *oracle problem*, that is the concern of generating reliable *test truths*. Special attention is paid to *practical utility* in emulator development and to avoid lengthy setup/integration work, which is a common problem of advanced automated testing technologies. Focaccia aims to be usable with minimal setup work and provide an 'out-of-the-box' experience, additionally capitalizing on a natural occurrence of test cases: native programs.

3.1 Focaccia's Approach

Focaccia implements an automatic verifier for emulators. The algorithm detects errors in an emulator's implementation by comparing its simulated program state with a *truth state* for which it implements an oracle that calculates expected program states which should result from executing instructions correctly. It accomplishes this by employing symbolic execution to capture instructions' true semantics.

The primary emulator verification algorithm requires as inputs:

- A concrete test trace, which is a trace of the program's execution via the tested emulator. It comprises a list of instructions executed and corresponding program state snapshots at each instruction, meaning the emulated process's register- and memory contents.

- A symbolic reference trace. This is the oracle, and it is computed by Focaccia. It is a trace of the same program, but, instead of program state snapshots, it defines symbolic equations that encapsulate the corresponding instruction's *correct* behaviour.

The verifier steps through the test trace and calculates for each program snapshot the truth state that **should** result from executing the corresponding instruction on the test state; it uses the respective equation from the symbolic trace to compute this truth. It then compares the next test state, meaning the actual state of the emulator after it executed the instruction, to the expected truth state. The emulator's implementation of that instruction is faulty if these states are not equal—in this case, a detailed error message is issued including the faulty instruction, the emulator's faulty state at that point, and the correct transformation that was expected to occur.

Focaccia is a command-line invocable program with a few helper tools. It takes as input a source of a concrete trace and a source of a symbolic trace. The former is usually an emulator's log. The latter is usually the traced program from which the verifier will generate the symbolic trace. Data from these sources are used to compute the comparison described above and finally print a summary of the results. Pre-computing symbolic traces for a program and exporting them to a file that can later be fed to the main program for verification is also possible. Additionally, we include a helper script that records concrete traces in the form of efficient minimal program snapshots from emulators that provide a `gdbserver` interface [24d] (QEMU, for example).

3.2 Focaccia's Position in the Software Testing Landscape

In the domain of automated testing, our specific interest can be categorized as *correctness testing*. We can further refine the classification by considering that the algorithm effectively compares the program's output to an expected oracle-based output, with no knowledge about the tested program's internal structure: it is a *black box* approach [SBC12]. It is noteworthy that symbolic/concolic execution is usually used as a white-box technique to prove properties of tested code, but Focaccia uses it to establish correctness not of the tested program itself, but of the tested program's **output**, because, in the special case of emulators, the output **is itself a program**. Thus, it inherits many of the common problems of black box testing:

- The set of testable input permutations is very restricted because test cases are created manually.
- Tests are unlikely to establish exhaustive certainty about the subject's correctness.
- Does not inherently maximize code-/branch coverage.

On the other hand, our approach does not require extensive setup work, thereby facilitating easy integration into the development workflow, is applicable to complex real-world test cases (particularly whole programs), and gives detailed information at the semantic level of the emulator's problem domain. It is thereby clearly distinguished from the unit testing approach because it establishes correctness on the higher level of macroscopic program function.

Although Focaccia covers the oracle problem, the necessity of generating the test cases themselves persists. It is therefore combinable with many of the existing techniques, paradigms, and structured approaches to generate test cases like equivalence partitioning, fuzzing, et cetera [Jan+16; YCW11].

4 Design

4.1 The Algorithm: An Abstract Description

4.1.1 Definitions

Emulators, or binary translators in general, take as input machine instructions of an ISA X and produce semantically equivalent instructions for an ISA Y. In the special case of emulators, we call X the *guest* ISA and Y the *host* ISA.

A program $P_X = (x_i)_{1 \leq i \leq n}$ of an ISA X is a series of n instructions, each of which is a mapping of program states $S_i \mapsto x_i(S_i)$. Thus, a program can be expressed as $P_X : S_1 \mapsto S_{n+1} = x_n(S_n)$ with an arbitrary fixed input state S_1. Two programs are **semantically equivalent** if $P_X(S) = P_Y(S)$ for all input states S. In this case, we write $P_X \equiv P_Y$.

A binary translator (which may be an emulator) can be modelled as a function $E_{X \to Y} : P_X \mapsto P_Y$. It is **correct** if the output program is semantically equivalent to the input program: $P_X \equiv E_{X \to Y}(P_X)$. A binary translator is **locally discrete**, or context-independent, if it achieves $P_X \mapsto P_Y$ by mapping each guest instruction individually to a semantically equivalent host program:

$$E_{X \to Y}(P_X) = E_{X \to Y}((x_i)_{1 \leq i \leq n}) = (E_{X \to Y}(x_i))_{1 \leq i \leq n} = P_Y \tag{4.1}$$

with $E_{X \to Y}(x_i) \in \mathcal{P}_Y$ (where $\mathcal{P}_Y$ is the set of all programs of Y). In other words, $E_{X \to Y}$ is distributive with respect to function application in X. Note that single instructions are programs by the definition above ($\forall x \in X : x \in \mathcal{P}_X$), meaning the argument to an emulator function E can be a single instruction or a series of instructions, i.e., a program.

If the semantic equivalence condition holds for each translation unit individually, it follows that it also holds globally:

$$x_i \equiv E_{X \to Y}(x_i) \forall i \implies (x_i) \equiv (E_{X \to Y}(x_i)) \implies P_X \equiv P_Y \tag{4.2}$$

4.1.2 Testing Program Equivalence

Focaccia tests the correctness of emulators (refer to section 3.2): formally, it takes on the endeavour of implementing the equivalence check $P_X \equiv E_{X \to Y}(P_X)$. To make this problem tractable, we can exploit the fact that binary translators are always implemented as locally discrete translators and combine that with the result from equation 4.2: We now check whether $x_i \equiv E_{X \to Y}(x_i)$ for all $x_i \in P_X$, which implies $P_X \equiv E_{X \to Y}(P_X)$.

Attempts to solve this equivalence directly bear the following problems:

1. We need to find x. Usually, an ISA does not have a canonical way to represent its instructions as manipulable functions.

2. We need to find $E_{X \to Y}(x)$.

3. Determining when $x \equiv y$ is not enough, even if it were feasible, because the translation $E_{X \to Y}(x) = (y_j)_{1 \leq j \leq m} \in \mathcal{P}_Y$ may be a series of **multiple** instructions in Y, so the true question is: When is $x \equiv (y_j)$?

Theorem solvers that can tackle problems 3 and 4 exist; an example is Microsoft's Z3 theorem prover [Z3p24]. Problem 2, however, makes this direct approach impossible. To elaborate: Static binary translators output the translation as generated machine code, but dynamic binary translators, on which our focus lies, do not. Emulators implement guest instructions opaquely in complex high-level language code and cannot extract the abstract equation that is implemented by this code, meaning it cannot know or communicate $E(x)$—the impossibility of directly solving problem 2 is the premise on which Focaccia is useful in the first place.

To circumvent this unsolvable requirement, we can restate the problem and test whether $x(S) = [E_{X \to Y}(x)](S)$, which by definition implies $x \equiv E_{X \to Y}(x)$. This means that we have to find the emulator's **state** rather than its action on states, which is doublessly practical. The set of problems to solve reduces to:

1. We need to find $S' = x(S)$. We call S' the *truth state*: it is the state that results when an instruction x is correctly applied to an initial program state S.

2. We need to find $S^E = [E_{X \to Y}(x)](S)$. We call S^E the *test state*; it is obtained by executing the translation of x on the same initial program state S as the one from which S' is computed. Section 4.4 is concerned with this task.

3. We need to determine when $S = S^E$. As program states S are constants, their equality is theoretically trivial. Practically, however, the respective initial states of the native program and the translated program are often not precisely equal. Therefore we implement an equivalence operator $S \equiv S^E$ which represents an equality with respect to differing start states. Section 4.1.4 explores this situation in detail.

4.1.3 Local Discreteness

It was argued that requiring a tested translator to satisfy local discreteness is imperative to verifying whether $P_X \equiv P_Y$ when dealing directly on the level of equivalences; proving propositions for programs is the concern of formal verification methods, which are rarely realistically applicable to nontrivial programs as a whole and certainly not in an automated way. Testing methods that can falsify these assumptions are superior to Focaccia and should be preferred in every case. Otherwise, as long as a pratical replacement for these techniques is demanded, a divide-and-conquer approach is inevitably necessary to break this monolithic problem open into manageable subunits: equation 4.2 enables this.

However, when we use the definition of semantic program equivalence to transform the problem from $P_X \equiv E_{X\to Y}(P_X)$ to $P_X(S) = [E_{X\to Y}(P_X)](S)$, the global comparison of result states becomes possible again. While this removes the theoretical necessity for the local discreteness assumption, it is still practically useful, if not essential. An outcome of comparing result states of full program executions would be a statement like "The program's translation is erroneous". This is too general of an assertion to be useful to developers. When we instead use local discreteness and check for all intermediate states whether $S_i = S_i^E$ (where $S_i = x_{i-1}(S_{i-1})$ and $S_i^E = E_{X\to Y}(x_{i-1})(S_{i-1}^E)$ for $i \neq 1$, with $S_1 = S_1^E$ an arbitrary initial state), we can generate much more detailed information, such as "Instruction x_i is translated incorrectly because $S_i \neq S_i^E$ ". This is exactly the type of feedback that Focaccia seeks to provide.

Additionally, it lets the algorithm recover from errors: Even if an instruction x_i turns out to be implemented incorrectly, thus producing an incorrect state S_{i+1}^E, that state is only incorrect from the viewpoint of program semantics. The verifier, however, can still treat it as a valid input state to all subsequent instructions $x_{j>i}$ because the translation $E_{X\to Y}(x_j)$ is required to be *locally correct* for a locally discrete translator, that is, it must work correctly at the level of instruction semantics on **any** program state.

A first version of a verification algorithm thus looks as follows (figure 4.1 visualizes this algorithm): We run P_X on one initial state S_1 and the translation $E_{X\to Y}(P_X)$ on the same initial state, during which we record intermediate states of each execution, obtaining two respective lists of program states $(S_1, S_2, \dots)$ and $(S_1^E, S_2^E, \dots)$, S_i and S_i^E defined as above. For each pair of states (S_i, S_i^E), we test whether $S_i = S_i^E$. If this equality does **not** hold, then $x_i \not\equiv E_{X\to Y}(x_i)$, meaning the translator's implementation of x_i is faulty.

Note that it is indeed valid to assume an emulator's implementation to be locally discrete because it is not only by far the most feasible strategy to implement any binary translator (at least disregarding optimization strategies), but also the core axiom of dynamic binary translation itself: an emulator is **defined** as an online interpreter that operates with only a local view on the program.

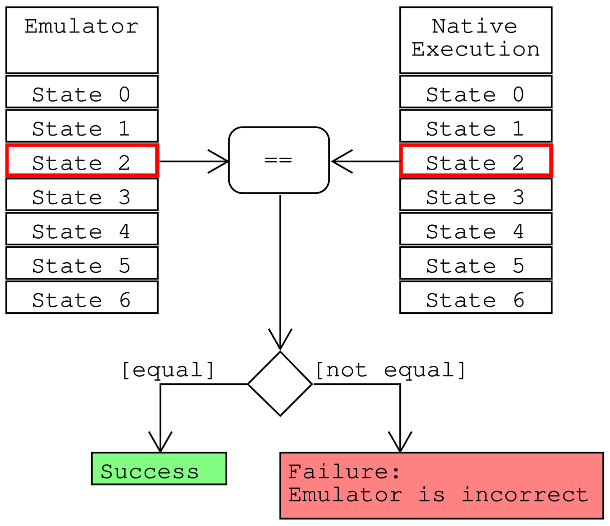


Figure 4.1: A naive verification algorithm

4.1.4 Comparing Program States

The algorithm requires an equality operator on program states: $S = S'$. An intuitive way of implementing this operator is by comparing the states' register- and memory contents. These values are what constitute the state of a program, and they are numeric values with a canonical condition for equality. However, as section 4.1.2 indicated, comparing program states is more complex in reality. That is because the starting states for the execution of programs are not always equal: the previous assumption that $S_1 = S_1^E$ is not necessarily true. In fact, it is most commonly false. Possible contributing factors (subsequently collectivized as a general *difference in environment*) include:

- Different initial stack pointers.
- Different addresses of heap allocations.
- Different environment- and auxiliary vectors. The latter is particularly interesting. It turns out that the auxiliary vector provided by QEMU, for example, routinely differs from the one provided by the operating system. See section 5.3.1 for more details.

- Operations on time, operating system state, network, etc. deal with effectively random values.

Therefore, instead of comparing $P_X(S) = P_Y(S')$, which does **not** imply $P_X \equiv P_Y$ if $S \neq S'$, the comparison must take into account the initial difference $\Delta_S = S - S'$ and establish a **state equivalence** $P_X(S) \equiv_{\Delta_S} P_Y(S') \implies P_X \equiv P_Y$ with respect to it.

The way we calculate this equivalence is by re-introducing information about the guest instructions x_i to the algorithm which, in order to simplify the problem, we have discarded when we transformed the central question from $x \equiv E_{X\to Y}(x)$ to $x(S) = [E_{X\to Y}(x)](S)$. The new algorithm works as follows: Instead of running guest program and translation in parallel and comparing their intermediate states, only run the translation $E_{X\to Y}(P_X)$ on a start state S_1^E, thereby obtaining the translation's intermediate states S_i^E. Then, for each S_i^E, use the guest instruction x_i to calculate a corresponding **expected state** $S_{i+1} = x_i(S_i^E)$. These represent truth states that would result from executing x_i on S_i^E if x_i was implemented correctly. Finally, compare the expected state to the actual translation state: $S_{i+1}^E = S_{i+1}$. Again, this works because it does not matter to the verifier whether S_{i+1}^E is a *correct* state with regards to whole-program semantics. Figure 4.2 shows a high-level overview of this algorithm. This concept is what Focaccia implements; only details and practical necessities will change from here on.

Decomposing the pre-translation program (x_i) into its instructions and applying them selectively to the emulator's test states as opposed to executing it natively on a semi-random starting state thus allows us to use the same starting state for both the translation and the truth program at each instruction, eliminating Δ_S. This enables the desired equality comparison $x_i(S_i) = E_{X\to Y}(x_i)(S_i)$. The drawback of this approach is that we now require an additional piece of information: We need to know x_i.

Not only do we need to know what every x_i is, but we also demand a way of applying it individually to an arbitrary program state we determine or, more precisely, which is being determined by the translation's execution. One approach would be to generate and run a small program or piece of code that sets up the correct machine state, then runs the instruction in question natively on a hardware implementation of X, and finally reads the resulting state back. We chose a different path: We use symbolic execution tools to translate instructions into equations, which we manually apply to the states we want to test. Section 4.2 explains how this works.

Another way to approach this issue would be not to adapt the algorithm to the environment but to adapt the environment to the algorithm: manually set up environments for both the concrete and the emulated execution. This is a potential topic for further research and relates to containers, though it may not be feasible in all contexts. Section 5.3.1 gives an example of how environmental differences are notorious.

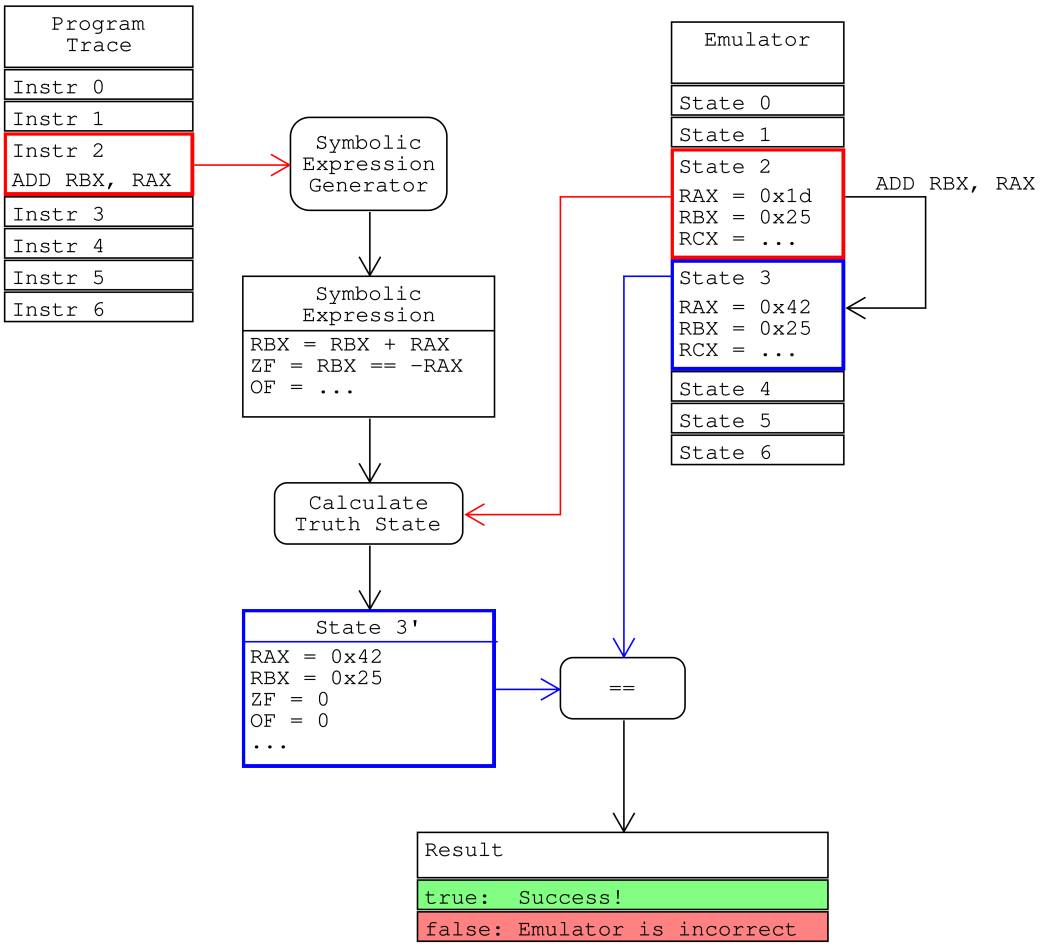


Figure 4.2: Overview of the Algorithm

4.2 Symbolic Execution

The task at hand is to find a representation of any instruction x that encapsulates the semantics of the transformation it applies to program states on which it is executed and is able to calculate that operation on synthetic program states. Focaccia uses symbolic execution tools to translate instructions into symbolic equations that do precisely that. It turns out that if one executes an instruction via a symbolic execution engine on an input state that is not only partly but entirely symbolic, the resulting equation fully represents that instruction's semantics.

This is an uncommonly seen use of symbolic execution as inputs to a program are usually symbolized selectively, as otherwise, the branching complexity of nontrivial programs quickly leads to exponential inflation of the equations' size and the number of explored paths. This problem is moot if we only trace a single instruction at any time: no symbolic branching conditions are ever propagated through multiple instructions. Overall, only one path through the program is considered, and the effort thus remains linear.

Formally, we denote a **symbolic expression generator** for an ISA X as a map G from an instruction $x \in X$ to a symbolic entity σ which is composed from a symbolic alphabet Σ and is itself a function of program states: $G_{X \to \Sigma} : x \mapsto \sigma$ with $x(S) = \sigma(S)$. The perceptive reader will have noticed that this translation resembles a locally discrete binary translator (though a generalized version whose target language is not specifically an ISA, but an abstract symbolic language), with the addition of a *correctness condition*. This is, in fact, the case. Essentially, Focaccia relies on the existence of one correct binary translator $G_{X \to \Sigma}$ that translates the guest ISA X to a symbolic language Σ. It uses G as an oracle to verify all other binary translators. See section 4.2.2 for further discussion of problems that this approach bears.

4.2.1 A Symbolic Representation

The component concerned with everything symbolic execution related is the **symbolic expression generator**. Its task is to translate instructions into corresponding symbolic representations which capture the instructions' semantics. Figures 4.3 and 4.4 show examples of symbolic equations generated from a `MOV` instruction and an `ADD` instruction, respectively (note that logical state flags in x86's `RFLAGS` register are being treated like individual variables/regsiters here). Section 5.2.1 details how this abstract expression generator is implemented in Focaccia and discusses the challenges involved in that.

```
MOV EDI, DWORD PTR [RSP + 0xC]
```

```
RDI = {@32[RSP + 0xC] 0 32, 0x0 32 64}
RIP = 0x40102D
```

Figure 4.3: Symbolic equations for `MOV` instruction

```
ADD QWORD PTR [RSP + 0x20], 0x9
```

```
@64[RSP + 0x20] = @64[RSP + 0x20] + 0x9
zf = @64[RSP + 0x20] == 0xFFFFFFFFFFFFFFF7
nf = (@64[RSP + 0x20] + 0x9)[63:64]
pf = parity((@64[RSP + 0x20] + 0x9) & 0xFF)
cf = (@64[RSP + 0x20] ^ ((@64[RSP + 0x20]
     ^ (@64[RSP + 0x20] + 0x9))
     & (@64[RSP + 0x20] ^ 0xFFFFFFFFFFFFFFF6))
     ^ (@64[RSP + 0x20] + 0x9) ^ 0x9)[63:64]
of = ((@64[RSP + 0x20] ^ (@64[RSP + 0x20] + 0x9))
     & (@64[RSP + 0x20] ^ 0xFFFFFFFFFFFFFFF6))[63:64]
af = (@64[RSP + 0x20] ^ (@64[RSP + 0x20] + 0x9) ^ 0x9)[4:5]
RIP = 0x401889
```

Figure 4.4: Symbolic equations for `ADD` instruction

4.2.2 Verifying the Symbolic Execution Backend

As has been shown above, we rely on one translator (in this case, one which translates instructions into symbolic equations) to be implemented correctly: It is an oracle. Fundamentally, it allows Focaccia to predict the outcome of applying an instruction to an arbitrary program state, thus providing a truth against which an emulator's state can be tested. The disadvantage of a reliance on a correct program is the virtual impossibility for nontrivial programs to be correct.

To mitigate the uncertainty that is thereby introduced into a tool that is supposed to **facilitate** certainty, we employ an online verification strategy that checks generated symbolic equations against the concrete reference state while recording the symbolic trace. The system warns the user when it encounters an incorrectly implemented instruction so that they shall not rely on the verifier's results regarding that particular instruction.

We do this by reusing the exact same procedure that Focaccia uses to test binary translators in the first place, but this time use concrete states, i.e., correct states by definition, as inputs for calculating the expected state after an instruction as well as the state against which this prediction is tested: We test whether $\sigma_i(S) = S_{i+1}$. If the prediction made by a symbolic expression is unequal to the concrete state calculated by the processor (which, though it could *theoretically* violate an ISA's specification, is still the most true calculation that we are able to obtain), then the symbolic expression generator $E_{X \to \Sigma}$ is incorrect. The pseudocode in Figure 4.5 illustrates this algorithm.

```
while program.is_running():
    instr = program.current_instruction
    state = program.current_state

    # Verify the previous instruction's predicted state
    if state != expected_state:
        warn_incorrect(program.prev_instruction)

    # Predict the next program state
    symb_expr = gen_symb_expr(instr)
    expected_state = symb_expr(state)

    program.step()
```

Figure 4.5: Verification of the Symbolic Expression Generator

4.3 Working on Complete Programs

At this point, we can represent single instructions' semantics as function-like equations, use these to manipulate program states at will, and compare program states to each other. Now we need to tie that mechanism into the context of a complete program execution.

4.3.1 Program Traces

When a program is executed with concrete values as inputs, its abstract control flow is instantiated to exactly one specific sequence of instructions: this is called the *program trace*.

As noted in the introduction to section 4.2, recording an entire program as one symbolic equation is impossible in almost all practical cases. Not only do branches cause the equations to blow up exponentially, but any kind of loop will never be able to halt at all as the termination condition can never resolve to a concrete `true` or `false` answer. This limitation has been pointed out before many times, and the practical solution is always to sacrifice accuracy (allow false positives or false negatives or both) for performance [Bal+18].

Focaccia solves this problem by borrowing concepts from concolic execution, that is, it selectively guides symbolic execution with concrete values as additional information to decide branching. The specific strategy employed by Focaccia is to run a native execution on the physical machine alongside the symbolic expression generator and to create symbolic expressions for all instructions contained in the concrete trace. In this scenario, the concrete execution defines which instructions to symbolize and in what order, while the symbolic expression generator computes those instructions' symbolic representations. This tactic avoids branching in the symbolic execution entirely, and yields one specific linear *symbolic trace*, which is the translation $E_{X \to \Sigma}(P_X) = P_\Sigma = (\sigma_i)$ of a specific run of a program into symbolic entities.

This approach eliminates the most blatant shortcomings of symbolic execution, though in doing so induces a dependency on a certain amount of concreteness. Besides sacrificing soundness (not all inputs for which the program is faulty are actually found), that concreteness contains remnants of the conceptually eliminated initial state difference Δ_S (see section 4.1.4). The way this difference manifests itself is by modifying the concrete execution's path through the program, and therefore also that of the symbolic trace.

4.3.2 Trace Mismatches

Nondeterministic Branching

One can imagine code that behaves effectively as the following:

```
n = random()
if n > 0.5:
    func_a()
else:
    func_b()
```

Figure 4.6: Nondeterministic branching

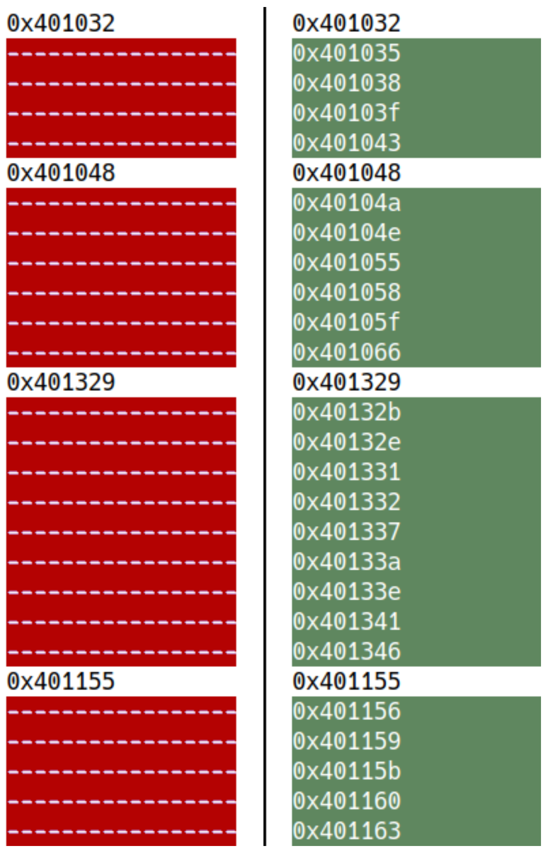
These situations do happen, be it during iteration over environment arrays or their content, literally deciding branches based on randomness, or generally any branching involving nondeterministic values, such as network, time, system state, . . . A specific situation in which this routinely happens is the libc initialization code, where auxiliary and environment vectors are processed. These nondeterministic mutations can cause the symbolic program trace to differ from the tested program trace.

This is a problem for Focaccia's standard algorithm where a symbolic trace is computed from a concrete trace, the process of which can be conceptualized as pre-recording because it never interacts with the tested emulator's state directly (which is purposefully so as the emulator's correctness cannot be trusted and should therefore not partake in *truth* generation), and is afterwards applied to an emulator's program trace. If the latter contains instructions that were never executed in the concrete execution because of different branching behaviour, the symbolic trace will not contain information about these instructions and cannot test them.

As this problem is unresolvable by a lack of information, the verifier resorts to skipping instructions that are not included in the symbolic trace and trying to find a point where both traces agree again. Warnings are issued to the user when instructions are skipped. The problem is improvable by ensuring similar initial conditions for the emulator's execution and the symbolic trace recorder (this is effort that the user has to bear), yet rarely avoidable. Section 4.3.4 discusses an experimental attempt to improve this situation, though it does suffer from major flaws.

Emulation Granularity

Another kind of trace mismatch happens if the tested emulator provides its trace on a different granularity level than Focaccia does. A common example is emulators stepping the program forward by **basic blocks**, whereas Focaccia always generates symbolic traces at single-instruction granularity. Figure 4.7 shows a diff view of a real-life trace granularity mismatch. On the left side of the diff, we see an emulator's basic-block-based instruction trace (rows are addresses of executed instructions), whereas the right side shows Focaccia's single-instruction symbolic trace generated from a concrete execution of the same executable. The first basic block ranges from instruction `0x401032` through `0x401043`. The second basic block starts at `0x401048`, etc.



```
0x401032         | 0x401032
---------------- | 0x401035
---------------- | 0x401038
---------------- | 0x40103f
---------------- | 0x401043
0x401048         | 0x401048
---------------- | 0x40104a
---------------- | 0x40104e
---------------- | 0x401055
---------------- | 0x401058
---------------- | 0x40105f
---------------- | 0x401066
0x401329         | 0x401329
---------------- | 0x40132b
---------------- | 0x40132e
---------------- | 0x401331
---------------- | 0x401332
---------------- | 0x401337
---------------- | 0x40133a
---------------- | 0x40133e
---------------- | 0x401341
---------------- | 0x401346
0x401155         | 0x401155
---------------- | 0x401156
---------------- | 0x401159
---------------- | 0x40115b
---------------- | 0x401160
---------------- | 0x401163
```

Figure 4.7: Different Program Trace Granularities

This mismatch is easily resolvable: In the higher-granularity symbolic trace, merge symbolic equations for all instructions of the same basic block into a single equation, i.e., transform the higher-granularity trace into a lower-granularity one. This will yield larger equations, though they will not blow up exponentially as basic blocks by definition do not include branches. A simple post-processing algorithm operating on the two traces can detect basic block boundaries reliably because one basic block never includes the same instruction multiple times.

Problematic is the case where both granularity mismatch **and** trace divergence happen in the same trace; this is fundamentally unsolvable in a trace post-processing scenario because one cannot differentiate between excess instructions resulting from different branching behaviour and excess instructions from higher trace granularity.

4.3.3 Obtaining Instructions

The symbolic expression generator takes as input an instruction, which it then turns into a symbolic expression. A prerequisite to that is to know which instruction is to be executed at each point in the trace. The first prototype used a disassembly framework to load a binary and disassemble it in its entirety, which was slow and produced many unnecessary computations (it disassembled instructions that were never touched by a specific program flow). The next iteration used the same mechanism to disassemble instructions on demand, i.e., at each program counter only parse a single instruction. This works for statically linked binaries, as all information is in one file and can be loaded from there. In the third iteration, in order to support dynamically linked programs and even Just-In-Time (JIT)-compiled code, we forego the binary file entirely and instead read the current instruction directly from the running program's memory; this is usually the concrete execution running alongside Focaccia that dictates the trace.

4.3.4 Experimental: A Technique for Perfectly Matching Traces

Focaccia could read current instructions directly from an emulator's memory, i.e., take the emulator as a source for the program trace instead of a separate concrete execution to guide the symbolic trace. This technique can give the best and most user friendly results that Focaccia is able to compute, free of all trace-based limitations. On the other hand, it fails whenever the emulator decodes instructions incorrectly; the technique is instead useful to verify mature emulators or hand-picked selections of test cases. Additionally, it requires either an online remote interface to the debugger or full information about the translated instruction in static data sources like emulator logs. Details about data exchange methods are discussed in section 4.4.

4.4 Recording Concrete State

The final principal task, now that Focaccia can produce entire symbolic program traces, is to gather snapshots of the tested emulator's concrete state such that symbolic expressions can calculate truth states from them, they can be compared to these truth states, and to do so efficiently enough so that this part does not become a bottleneck for the entire verifier. The latter part in particular turned out to be surprisingly challenging.

What is Program State?

The momentary state of a program comprises the process's register- and memory content. The latter specifically being all virtual memory pages allocated to the process in question. Everything other than these values, such as disk state, network state, or state of peripheral hardware, is considered **input to**, not **state of** the program.

Obtaining Snapshots from Emulators

Focaccia needs a way to interface with emulators to read their state. This, of course, depends almost exclusively on what sort of logging or debugging interfaces that are implemented by the emulator in question. The first snapshotting mechanism that we implemented was a simple log parser specific for Arancini's log format—frankly, just because Arancini served as the first emulator on which early prototypes of Focaccia were tested. Its log format, at that time, was severely restricted in that Arancini exclusively wrote register values to the log, but no memory content. The latter is always a challenge for static, non-interactive communication paths like log files as their size can easily blow out of proportion for even moderately sized programs when the emulator dumps *all* memory to disk at every instruction. Additionally, almost all memory is never touched by each given instruction (a typical non-vector instruction addresses 16 unique bytes of memory at most) and is thus not strictly required by the verifier.

Log files are a well decoupled solution that only requires a single parsing function to be implemented to support a new emulator. On the other hand, dumping memory to files in a large-scale fashion is very costly and the emulator has to **support** dumping memory to logs in the first place. Our major evaluation target, QEMU, does not. Another more memory-efficient approach is an online debugging interface, such as the one QEMU provides by implementing the `gdbserver` protocol. This method is less intrusive to Focaccia (in fact, it demands no modification of Focaccia at all to interface with new emulators) but moves some of the effort to the side of the emulator, at which the responsibility lies to implement said protocol.

Recording QEMU State

QEMU is an explicit evaluation target of Focaccia. It has extensive options for logging output, but it cannot write memory to the log. Additionally, even though logging can output memory mapping informations, QEMU's `gdbserver`-based remote debugging interface can not, so that there is no obvious way to know which memory addresses exactly to include in the snapshot. The result was a rather inventive first iteration of a customized snapshotting system for QEMU, for which Figure 4.8 gives a high-level visualization.

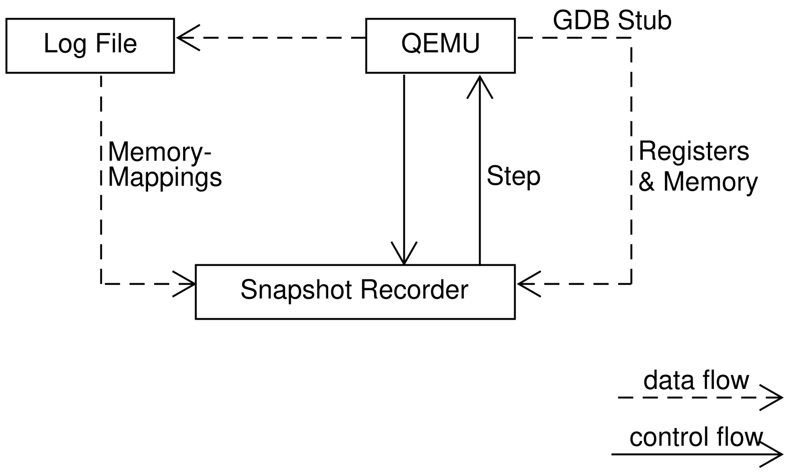


Figure 4.8: Recording QEMU Snapshots

The snapshot recorder is an algorithm that does multiple things. First, it starts an instance of QEMU, passing to it several options that enable logging and the `gdbserver` stub, which makes QEMU wait before the first instruction until a connected GDB instance instructs it to start execution. The script then connects to the GDB stub, which it uses from there on to read register- and memory values from QEMU, and also to step the emulation forward by increments of single guest instructions. To know what bytes of memory to record, it simultaneously reads from QEMU's log file, to where QEMU outputs its memory mappings every time they change (e.g., when a new page is allocated). The recorder thus keeps track of the memory pages allocated to the emulated process and records at each instruction all content of all allocated pages, in addition to all register contents, in a snapshot.

Unfortunately, this approach turned out not only to be highly memory inefficient seeing as almost all memory of a process is irrelevant for each given single instruction, but also exceedingly slow—so much so that the system was virtually unusable: For a minimal 'Hello World'-program, which, compiled with `musl libc`, results in a trace of 1106 instructions on our test system and compiler version, the memory footprint of an average snapshot is around 2.2 megabytes and the average time taken to read that amount of data over QEMU's GDB stub is around 185 milliseconds, resulting in a total 15.88 gigabytes of snapshot data recorded over 321 seconds, or 5.35 minutes, of execution time.

These results are unacceptable to any productive development setting, not to mention that the solution is entirely specific to QEMU and cannot be transferred to other emulators easily. The evolution of this system is our *minimal snapshotting* approach. It reduces the amount of data stored in each snapshot to the theoretical minimum that is required for the verifier to compute its results, while simultaneously removing the reliance on a specific log structure by moving to a purely `gdbserver`-protocol-based interface. Even this interface is highly exchangeable with any remote communication protocol as long as it supports three basic commands: Reading memory content at arbitrary addresses, reading register content, and stepping the program forward by single instructions. The following section explains all of this in detail.

Minimal snapshots

A key capability that interactive remote debugging interfaces to running emulators enable for the snapshotting engine is that it can **choose** which parts of the emulator's state it wants to record (in contrast, for example, to static log files where the emulator chooses which data to dump). By considering as additional information the symbolic representation of an instruction's semantics, it can calculate the exact subset of a program's state that is needed to verify only that instruction, and record it in a *minimal snapshot*.

Minimal data to verify an instruction's correctness encompasses information from **two** program states: The input state to the instruction, i.e., the state on which the instruction acts, and its output state, i.e., the state that the instruction produces; the former to compute the expected reference destination state, the latter to compare it to the reference state. Relevant data of the input state are input values to the instruction (its *source operands*), while that of the output state encompasses the instruction's output values (its *destination operands*). We can determine both from an instruction's symbolic equations.

To create a minimal snapshot s_i from a full program state S_i^E so that it has all data required by the verifier to calculate the comparison to its respective truth state $\Delta_{S_i} = S_i - S_i^E$ and also calculate the next expected truth state $S_{i+1} = \sigma_i(S_i^E) = \sigma_i(s_i)$ from it, the following information is required:

- σ_{i-1}: A representation of the transformation that produces S_i. Determines the destination operands of x_{i-1}, the values of which are required to compare S_i^E to the expected truth state.

- σ_i: A representation of the transformation that acts on S_i^E. Determines the source operands of x_i, the values of which are used to calculate the next truth state S_{i+1} that will be compared to S^E_{i+1}.

- S^E_{i-1}: The previous program state on which producing instruction x_{i-1} acts. We need it to record s_i because symbolic addresses of memory accesses in x_{i-1}'s destination operands, which must be included in s_i, depend on it. Example: for $x_{i-1} =$ `mov [rsp],rax`, we need to know the value of `rsp` in S^E_{i-1} so that we can include memory at that address in s_i.
- S^E_i: The current emulator state that is to be reduced to s_i.

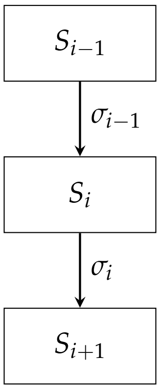


Figure 4.9: Instruction data flow

Example The instruction $x =$ `ADD [RDI+0x20], RAX` calculates `RAX + 64@[RDI+0x20]` and stores the result in `[RDI+0x20]`, also modifying some state flags in the process. We shall call the emulated program state before the instruction is executed S^E_0 and the emulated state after the instruction $S^E_1 = E(x)(S^E_0)$.

The snapshot of the state S^E_0 must contain at least the register `RAX` and 8 bytes of memory `64@[RDI+0x20]` so that the expected state $S_1 = x(S^E_0)$ can be calculated. The snapshot of the emulator state S^E_1 must contain the output values `64@[RDI+0x20]` and `RFLAGS` so that the comparison $S_1 = S^E_1$ can be performed. Additionally, because the memory address to which the instruction writes is based on `RDI`'s value in S_0, not in S_1, the snapshot of S_0 must also store `RDI` so that the following state can calculate the address `[RDI + 0x20]`. In the context of a continuous trace, the second snapshot must also contain all inputs to the next instruction, and so forth.

When we apply this technique to record snapshots from QEMU in the same testing environment as used in the measurements above, the algorithm generates a total of 121 kilobytes of snapshot data (a large part of which is overhead from the admittedly far from optimized JSON format we use to serialize snapshots) recorded over a total of 6.9 seconds. This is a reduction in space consumption of 13,123,900% and a 4,650% reduction in execution time.

However, minimal snapshots, as always, introduce a new restriction: The algorithm can now establish only a *lower bound* on emulator correctness, that is, tell whether an instruction does **at least** what it is supposed to do. With entire-memory snapshots, an upper bound, which means that an instruction does **at most** what it is supposed to do, would have been possible to establish because its calculation does not inherently suffer from the Δ_S problem—the only thing that needs to be tested is whether all values that should **not** have been touched by an instruction have remained the same, no matter their concrete value. Obviously, establishing both an upper- and a lower bound would imply certainty that an instruction does **exactly** what it is supposed to do. With minimal snapshots and all of the benefits that go along with them, however, that becomes impossible since the data required for the calculation is not present anymore.

Note that, even though this system was motivated by the attempt to interface with QEMU, it is applicable to all emulators that provide some kind of remote interface. Focaccia contains an implementation for the `gdbserver`-protocol.

5 Implementation

Focaccia is a Python application. This section details some notable points of its implementation.

5.1 Tools

5.1.1 Symbolic Execution Backend

A symbolic execution backend, in order to be useful for our specific purpose, must satisfy multiple conditions:

- Symbolic execution for assembly languages (many tools operate on high-level languages); primarily x86, but preferably more
- Python API
- Ability to execute single instructions symbolically on arbitrary states
- Low-level access to generated symbolic expressions

The first tool in consideration was the binary analysis framework *angr* [Sho+16]. angr is a mature and well-tested open source platform for control-flow graph recovery, symbolic execution, disassembly, lifting, and more [24a]. Still, angr was rejected for use as Focaccia's symbolic execution backend. Almost none of our tasks were easily accomplished with angr; the interface is so tuned to high-level cyber security tasks that our lower-level needs only resulted in battles against the API. Not only does angr bury its low-level functionality, which would have sufficed for Focaccia's purposes, under large stacks of abstraction, delegation, and opaqueness, but also turned out to be slow while doing that.

Further research has led us to adopt Miasm [Des12], a reverse engineering framework developed by CEA IT Security at the French Alternative Energies and Atomic Energy Commission. Its features include opening/modifying/generating binary files, assembling/disassembling a host of assembly languages, emulation, symbolic execution, and **representing assembly semantics using an intermediate language** [cea24]. The latter statement hints towards exactly the functionality that Focaccia requires.

The evaluation has shown that the backend's completeness really is a major pain point of Focaccia. As a proposal for future improvement, it is entirely possible to build a custom abstraction layer over the symbolic execution backend as a means to have multiple interchangeable backends for feature- and implementation completeness of all desired architectures. We have discarded this idea initially for its likelihood to incur unnecessary effort.

5.1.2 Concrete Execution Tracer

We need to follow a program's native execution on a per-instruction basis and inspect the program's state at each trace point (see section 4.3). A debugger is a canonical choice for this task; we use LLDB because it has a Python API (one can start and manipulate debugger instances conveniently in Python code) [Tea24].

5.2 Algorithms

5.2.1 Generating Symbolic Expressions

The goal for transforming instructions to symbolic expressions is to generate equations that are, despite being guided to a certain extent by the native execution's concrete reference state, as symbolic as possible. Ideally, the concrete state's only purpose would be to indicate for which instructions symbolic expressions are to be generated, which is indeed possible for the most part. For example, an `ADD` instruction always performs the same conceptual operation for each configuration of operand types (memory, register, or immediate), no matter the concrete values of the program state on which it is executed. Conditional instructions like `CMOVcc`, on the other hand, modify their behaviour based on the condition operand's concrete value. The first iteration of the implementation inspected the concrete state to resolve to which location the instruction writes: `CMOVNZ RDX, RAX` would generate the equation `RDX = RAX` if `ZF = 0` in the concrete reference state, and `RDX = RDX` otherwise [Int23]. However, the formula can be improved to be less reliant on concrete state, thereby increasing the number of test cases it can verify correctly, if our symbolic language has support for conditional expressions. This is the case with Miasm. An improved version of the algorithm generates for the same instruction the equation `RBX = (ZF == 0) ? RAX : RBX`. No concrete state is considered at all—the equation is perfectly symbolic and captures all possible scenarios.

The perfectly abstract approach does not work for all instructions: Repeated string instructions like `REP STOS`, where the number of loop iterations depends on the initial state, can not be captured in a finite formula because they do not necessarily terminate for all inputs—at least not in a realistic amount of time and with realistic space consumption: an equation for a possible loop count of $2^{64} - 1$ on 64-bit architectures is theoretically but not practically representable or even computable. For these instructions, the concrete reference execution is used as a termination condition: the generated equation will always have a number of iterations matching the expected number for its specific initial state, even though an emulated execution may execute a different number of iterations if its initial state differs (see section 4.3.2). This is a severe limitation on its verification power, though not circumventable. Listing 5.1 shows an example.

```
REP STOS es:[RDI] EAX
```

```
RDI = RDI + 0x130
RCX = RCX + 0xFFFFFFFFFFFFFFB4
@32[RDI] = RCX ? (RAX[0:32], @32[RDI])
@32[RDI + 0x4] = (RCX == 0x1) ? (@32[RDI + 0x4], RAX[0:32])
@32[RDI + 0x8] = (RCX == 0x2) ? (@32[RDI + 0x8], RAX[0:32])
@32[RDI + 0xC] = (RCX == 0x3) ? (@32[RDI + 0xC], RAX[0:32])
@32[RDI + 0x10] = (RCX == 0x4) ? (@32[RDI + 0x10], RAX[0:32])
...
```

Figure 5.1: Symbolic equations for `REP` instruction

The Miasm backend lifts parsed instructions to an Intermediate Representation (IR) on which the symbolic execution engine operates. For semantically complex instructions like the latter two described above, it generates multiple IR blocks that are chained together, forming an IR control flow graph; Figures 5.3 and 5.4 show examples of these IR graphs for instructions `CMOVNZ RDX, RAX` and `REP STOS es:[RDI], EAX`, respectively.

For our purposes, we want to collapse the entire graph into a unified collection of assignment statements, such that Figure 5.4 produces the symbolic equations shown in Figure 5.1. We accomplish this with a small symbolic interpreter that traverses the graph (i.e., executes the graph purely symbolically) and collects the visited nodes' modifications to the program state. If a branch is encountered, instead of deciding on a path to take, both destinations are considered and combined into a single conditional assignment, thus avoiding the need to consider concrete state to decide the branch condition. As noted before, this works because a single instruction's internal branching behaviour is usually simple enough not to induce explosive path growth. Figure 5.2 shows this transformation for the `CMOVNZ RDX, RAX` example. The left side shows the instruction's semantics as pseudocode, the right side shows the resulting conditional assignment. As a reference, Figure 5.3 visualizes the instruction's IR graph as generated by the Miasm backend.

```
if not ZF:
    RDX = RAX
```

(a) Branch

```
RDX = ZF ? RDX : RAX
```

(b) Combined Conditional Assignment

Figure 5.2: Transforming Branches to Conditional Assignments

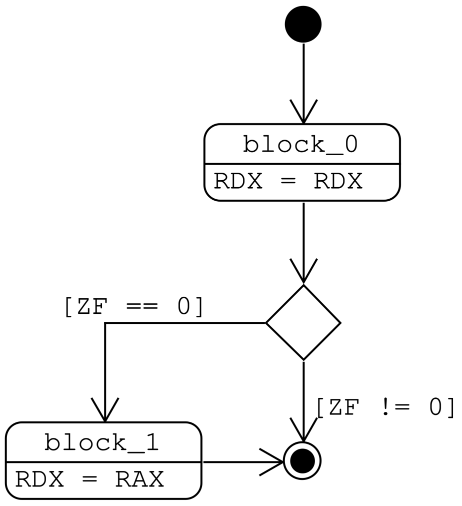


Figure 5.3: IR graph for `CMOVNZ RDX, RAX`

To handle the complex case of loops, the branch destinations of branches that are encountered repeatedly are concretized to a single IR block by taking into account the concrete guidance state. This decides possibly unhalting loop termination conditions and is the only case where concrete state is considered during symbolic execution.

5.2.2 Calculating Truth States

The process of computing truth states by applying symbolic expressions to initial states is comparatively straightforward. For any symbolic assignment statement (symbolic representations of instructions consist only of assignments at the top level, either to registers or to memory locations), a small interpreter walks through the right hand side's syntax tree and replaces all symbolic references to registers or memory locations

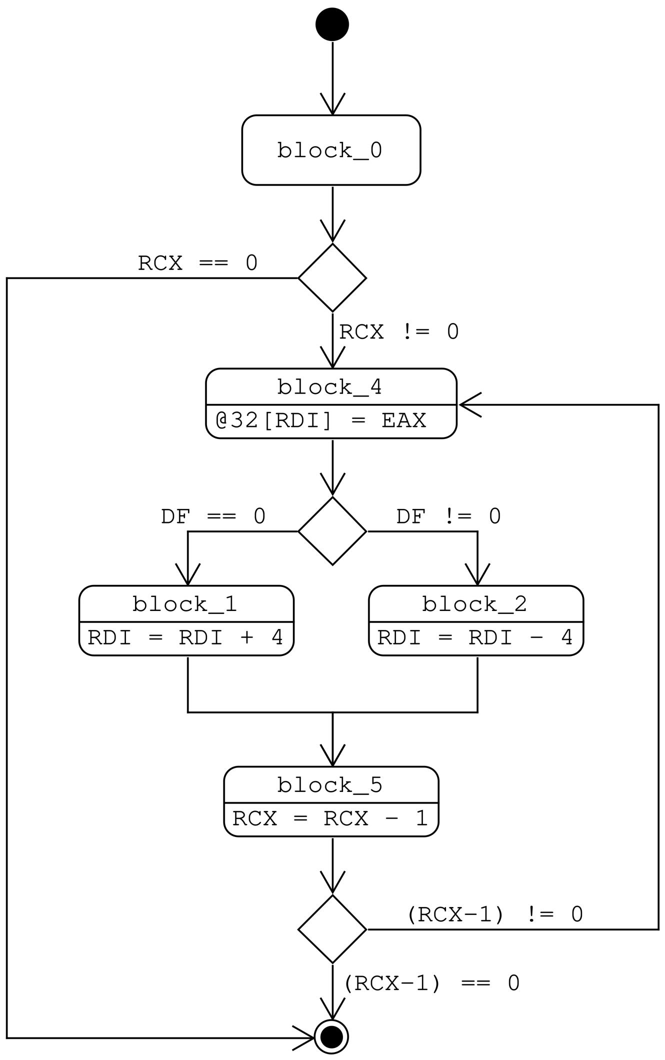


Figure 5.4: IR graph for `REP STOSD`

(the addresses of which are resolved to concrete values the same way) with concrete values from the initial program state to which the symbolic expression is applied. Once the expression is devoid of symbols, an expression simplifier provided by the backend can collapse complex arithmetic expressions or bitvector-accesses to single values. The assignment destination in the updated state, determined by the assignment's left hand side to which the same process is applied in case it is a complex expression, is updated with this value.

5.2.3 Verifying a Program State's Correctness

The comparison of states is similarly uncomplicated. For any pair of emulator states (S_i^E, S_{i+1}^E) and a symbolic transformation $\sigma_i : S_i^E \mapsto S'_{i+1}$ we use that transformation to calculate the expected state S'_{i+1}, then test whether all values that were *modified* by σ_i are equal to the respective values in the emulator state S_{i+1}^E.

Errors of different severity levels are generated depending on the type of error found: A common warning that is not a definite error in the emulator is insufficient data (that is, any of the emulator states don't contain data required by the transformation or the comparison), which may happen if the emulator's log is incomplete or other complications occur in the communication of the emulator's states.

5.3 Details

5.3.1 Environmental Differences: The Auxiliary Vector

A curious source of difference in branching behaviour (as discussed in section 4.3.2) between concrete- and emulator execution is the auxiliary vector on Linux systems, which is "a mechanism that the kernel's ELF binary loader uses to pass certain information to user space when a program is executed" [24b]. Libc's initialization code (at least glibc as well as musl libc [24c]) iterates over that array to bring it into a representation that is indexable by entry names, thereby depending its branching behaviour on the array's size. It turns out that QEMU, at least on some systems, passes an auxiliary vector to the application that is different from the native auxiliary vector on the same system; Figure 5.5 shows an example. While a customized launcher that starts both the tested emulator and the symbolic trace recorder could in theory ensure equal environment arrays, the same is not possible for the kernel-provided auxiliary vector.

This shows that branch-based trace mismatches are not a rare phenomenon.

```
AT_BASE : 0x790a3a4f3000
AT_CLKTCK : 0x7ffd4e2f21b9
AT_EGID : 0x3e8
AT_ENTRY : 0x5d141499c050
AT_EUID : 0x3e8
AT_EXECFN : 0x7ffd4e2f2fec
AT_FLAGS : 0x0
AT_GID : 0x3e8
AT_HWCAP : 0x64
AT_HWCAP2 : 0x2
AT_MINSIGSTKSZ : 0x7f0
AT_PAGESZ : 0x1000
AT_PHDR : 0x5d141499b040
AT_PHENT : 0x38
AT_PHNUM : 0xd
AT_PLATFORM : 0xbfebfbff
AT_RANDOM : 0x7ffd4e2f21a9
AT_RSEQ_ALIGN : 0x20
AT_RSEQ_FEATURE_SIZE : 0x1c
AT_SECURE : 0x0
AT_SYSINFO_EHDR : 0x7ffd4e349000
AT_UID : 0x3e8
```

(a) Native AUXV

```
AT_BASE : 0x2aaaab2ac000
AT_CLKTCK : 0x2aaaab2ab7f9
AT_EGID : 0x3e8
AT_ENTRY : 0x555555557050
AT_EUID : 0x3e8
AT_EXECFN : 0x2aaaab2abfd1
AT_FLAGS : 0x0
AT_GID : 0x3e8
AT_HWCAP : 0x64
AT_PAGESZ : 0x1000
AT_PHDR : 0x555555556040
AT_PHENT : 0x38
AT_PHNUM : 0xd
AT_PLATFORM : 0xfcbfbfd
AT_RANDOM : 0x2aaaab2ab7e0
AT_SECURE : 0x0
AT_SYSINFO_EHDR : 0x2aaaab2e3000
AT_UID : 0x3e8
```

(b) QEMU's AUXV

Figure 5.5: Comparison of auxiliary vectors in native execution and QEMU

5.3.2 Extending to Other Architectures

As Focaccia deals in **program states** rather than instructions (which must be supported by the symbolic execution backend, but bear no significance for Focaccia itself), the only thing necessary to support more architectures than the currently supported x86 is to define a list of an architecture's registers. Focaccia has a built-in mechanism to define subsets of registers (e.g. `EAX = RAX[0:32]`) as well as state flags. Peripheral tools such as communication with `LLDB` or `GDB` may need to be implemented for specific architectures as well, depending on the consistency of those tools' interfaces.

6 Evaluation

The primary evaluation target for Focaccia is QEMU with x86 as the guest architecture. Ideally, Focaccia would also be tested with Arancini, which was the initial motivation for this project. Unfortunately, support for most of the targets Arancini aims to support is not polished enough to be able to execute full programs at the time of writing. Additionally, its logging capabilities are severely limited and restrict the calculations that Focaccia is able to perform to a significant degree.

To test Focaccia's functionality, we attempt to reproduce and detect bugs that were already reported, confirmed, and fixed so that we have a verifiable measure for success. A search on the gitlab issue tracker yields a total of 235 bug reports with a `target:i386` tag, nine of which are instruction semantics related bugs. It is noteworthy how rare this specific type of bug on which Focaccia specializes is in mature implementations.

6.1 Setup

For each bug, we compile the newest version of QEMU that still contained the bug. We try to reproduce the bug with a minimal program, which is usually included in the bug report. If reproduction succeeds, we run Focaccia on the emulator with the same program and see if it detects the bug automatically.

Not much is to be said about the concrete testing process because using Focaccia does not incur a lot of effort: We generate a symbolic trace, then start a QEMU instance with the GDB-stub active to provide an interface for Focaccia, and finally execute the verifier script that interfaces with a GDB server, feeding it the pre-computed symbolic oracle. Figure 6.1 shows an example, where `bug.out` is the minimal reproducing program. Both Python programs that are being invoked are utilities provided by Focaccia.

```
python capture_transforms.py -o oracle.trace bug.out
qemu-x86_64 -g 12345 bug.out &
python verify_qemu.py --symb-trace oracle.trace localhost 12345
```

Figure 6.1: Testing Process

6.2 Results

By performing the process described above for all nine bugs, we reach the following results:

- 6 bugs were not detectable because the faulty instruction is not implemented by the symbolic execution backend. In this case, Focaccia prints a warning and skips the instruction.
- 1 bug was not detectable because the faulty instruction's malfunction caused QEMU to crash before Focaccia was able to read its state.
- 1 bug was detected successfully.
- We were unable to reproduce one of the bugs entirely.

These results showcase the impact that reliance on a correct and complete symbolic execution backend has in a real-world scenario. The unimplemented instructions are so rare that errors in their implementation were only found and reported in 2021, many years into both QEMU's and Miasm's widespread production use. However, a ratio of $\frac{1}{8}$ reproducible bugs detected is still lower than expected and indicates that the assumption of higher implementation correctness of symbolic execution tools than emulators might be unreasonable—at least if one aims to attain a deeper than superficial attestation of correctness. Possible actions are discussed in chapter 9.

Details

The unimplemented instructions are `ADDSUBPS` (SSE), `BZHI`, `BEXTR`, `BLSMSK`, `BLSI/BLSR`, and `VPSHUFB` (AVX, which is entirely unsupported by the SE backend).

The crashed case was the instruction `ADOX`. Note that the crash only happened when QEMU was executed with the `-g <port>` flag that enables its `gdbserver` interface, which is required for verification with Focaccia. This seems to have been another bug in QEMU that is not directly related to emulation.

The bug that was successfully found by Focaccia is in the implementation of `CMPXCHG`, which was reported in gitlab issue #508. QEMU zero-extends `EAX` to `RAX`, even though it should not if the compared values are equal: In the test case, `RAX` should contain the unmodified value `0x1234567812345678` after `CMPXCHG` is executed, but QEMU sets it to `0x0000000012345678`. Focaccia's output to that instruction is shown in Listing 6.2 (wrapped to fit on the page).

```
----------------------------------------------------------------------
For PC=0x40116a
----------------------------------------------------------------------
  1. [ERROR] Content of register RAX is false.
     Expected value: 0x1234567812345678,
     actual value in the translation: 0x12345678.

Expected transformation: Symbolic state transformation 0x40116a -> 0x40116e:
  [Symbols]
    ZF = (-@32[RBP + 0xFFFFFFFFFFFFFFEC]) == (-RAX[0:32])
    RAX = ((-@32[RBP + 0xFFFFFFFFFFFFFFEC]) == (-RAX[0:32]))
            ?(RAX,{@32[RBP + 0xFFFFFFFFFFFFFFEC] 0 32, 0x0 32 64})
    RIP = 0x40116E
    @32[RBP + 0xFFFFFFFFFFFFFFEC]
          = ((-@32[RBP + 0xFFFFFFFFFFFFFFEC]) == (-RAX[0:32]))
            ?(RDI[0:32],@32[RBP + 0xFFFFFFFFFFFFFFEC])
  [Instructions]
    CMPXCHG DWORD PTR [RBP + 0xFFFFFFFFFFFFFFEC], EDI
Actual difference: Snapshot (x86_64):
                 {'RBP': '0x0', 'RAX': '0xedcba98800000000', 'RIP': '0x4'}
```

Figure 6.2: Verifier Output for Faulty CMPXCHG Instruction

7 Related Work

Software testing in general naturally encompasses a giant body of research [DF14; FR19; GM16] and has already been mentioned in chapter 3. Focaccia, however, focuses specifically on testing emulators, which is a much smaller field of research.

Verification of Emulators

A 2009 work dubbed *EmuFuzzer* [Mar+09] explores a similar approach to that of Focaccia, but uses execution on a physical CPU as the oracle. It combines this approach with an automated, systematic fuzzing-based test case generator.

Ma, Forin, and Liu describe an approach where technical ISA and CPU specifications are mined to generate test cases automatically, which are then run against a hardware oracle [MFL10].

PokeEMU is a framework for systematic high-coverage testing and cross-validation of processor emulators [Mar+12]. It uses symbolic execution to extract semantical representations of instruction behaviour from a 'high-fidelity' emulator (one that models real CPU behaviour faithfully on a low level) to verify 'low-fidelity' emulators (ones that employ complex strategies to model instructions on a high level, but not the CPU itself).

Binary Translation in General

A subset of any binary translation process is to lift assembly to an intermediate representation that represents the lifted instructions' semantics accurately. A 2017 paper implements *MeanDiff*, a systematic testing method for binary lifters [Kim+17]. It detects bugs by collecting IRs from multiple independent binary lifters and formally testing for syntactical equivalence among them.

Another formal approach called a *Compositional Lifter* composes validated IR sequences to prove binary lifting correctness for entire programs [Das+20].

8 Conclusion

This thesis detailed the design and implementation of *Focaccia*, an automatic emulator verification system. It tests the program state of an emulator against corresponding oracle-generated states to verify whether instructions act on it correctly. Doing this, it verifies the execution of entire programs.

Focaccia is easy to use and integrate into an emulator's development process. It has several safeguards against real-world environment imperfections and data corruption scenarios to prevent false positives, maximizing its reliability. Furthermore, it scales the quality of its calculations with the quality of its inputs.

The downside to our symbolic execution-based oracle is that Focaccia's ability to detect errors in the tested emulator depends exclusively on the quality of the symbolic execution backend—which, as the evaluation shows, may be unreliable. We have effectively **moved** the point of responsibility for correctness into the symbolic execution tool. This has merit: A single, isolated source of definitions for instruction semantics must be implemented once correctly and reliably to validate any other binary translator. Formal verification methods may be able to accomplish this, after which Focaccia can apply their results to any other binary translation tool.

A clear deficiency of our method is that it is not systematic with respect to program inputs: It works on specific instantiations of test cases, i.e., executions of programs. For each run, each executed instruction is tested for exactly one concrete operand value. Generation of test cases must be systematic to maximize the verification's exhaustiveness.

Focaccia's source code resides at `https://github.com/TUM-DSE/focaccia`.

9 Future Work

9.1 The Oracle Problem

Focaccia chooses symbolic execution as a tool to implement an oracle that predicts the correct behaviour of instructions. The evaluation has shown that this approach is theoretically servicable, though susceptible to the same problems as those that it tries to solve: Implementing correct representations of instruction semantics is difficult and prone to mistake.

A possible solution is to look for a more complete symbolic execution backend. Unfortunately, we decided early on in the development process that an abstraction layer for multiple interchangeable symbolic execution backends was not necessary. This is a misjudgement in hindsight and will incur some extra work to implement the mechanism now, though it is certainly still possible to do so.

On the other hand, one could look into a replacement strategy for symbolic execution, as that is currently Focaccia's dominant weakness—a theoretically more sound approach to the oracle may be required. Section 4.1.4 hints towards a possible alternative: generate a small program that sets up processor and memory state, runs an instruction on it, and then reads back the result. It uses a hardware implementation as the oracle, which undergoes much more rigorous testing than a software emulator, or may even be set forth as the **definition** of correctness, depending on context. One may be able to re-use the existing reproducer's mechanism (implemented by Alp Berkman on top of Focaccia), which performs pretty much exactly this task.

9.2 Environments

Another one of Focaccia's struggles is environment difference between the process that guides symbolic execution and the tested emulator. Advanced strategies either to ensure equal environments for both executions or to minimize the difference's impact can improve the quality of verification results.

9.3 Systematic Test Case Generation

Focaccia is highly automatic, though still works on specific test cases, meaning the approach is not much more systematic than traditional unit testing—it merely tests the program on a different level. Any technique that generates inputs to it systematically can drastically raise the tests' exhaustiveness. The reproducer mentioned above could again find use here: It can be extended to generate not only one minimal reproducing program for each bug that Focaccia finds (which is its current purpose), but systematically explore the range of inputs to the faulty instruction and generate exhaustive test cases/programs for it.

Abbreviations

ISA	Instruction Set Architecture
JIT	Just-In-Time
IR	Intermediate Representation

List of Figures

2.1 Symbolic execution example 3

4.1 A naive verification algorithm 12
4.2 Overview of the Algorithm 14
4.3 Symbolic equations for `MOV` instruction 16
4.5 Verification of the Symbolic Expression Generator 17
4.6 Nondeterministic branching 19
4.7 Different Program Trace Granularities 20
4.8 Recording QEMU Snapshots 23
4.9 Instruction data flow 25

5.2 Transforming Branches to Conditional Assignments 30
5.3 IR graph for `CMOVNZ RDX, RAX` 30
5.4 IR graph for `REP STOSD` 31
5.5 Comparison of auxiliary vectors in native execution and QEMU 33

6.1 Testing Process 35
6.2 Verifier Output for Faulty `CMPXCHG` Instruction 37

Bibliography

[24a] *angr*. [Online; accessed 25. Mar. 2024]. Mar. 2024.

[24b] *getauxval(3) - Linux manual page*. [Online; accessed 1. Apr. 2024]. Mar. 2024.

[24c] *musl libc*. [Online; accessed 1. Apr. 2024]. Feb. 2024.

[24d] *Remote Protocol (Debugging with GDB)*. [Online; accessed 8. Apr. 2024]. Apr. 2024.

[AKS00] E. Altman, D. Kaeli, and Y. Sheffer. “Welcome to the opportunities of binary translation.” In: *Computer* 33.3 (2000), pp. 40–45. DOI: `10.1109/2.825694`.

[Bal+18] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi. “A Survey of Symbolic Execution Techniques.” In: *ACM Comput. Surv.* 51.3 (May 2018). ISSN: 0360-0300. DOI: `10.1145/3182657`.

[BEL75] R. S. Boyer, B. Elspas, and K. N. Levitt. “SELECT—a formal system for testing and debugging programs by symbolic execution.” In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 234–245. ISSN: 0362-1340. DOI: `10.1145/390016.808445`.

[cea24] cea-sec. *miasm*. [Online; accessed 25. Mar. 2024]. Mar. 2024.

[CM] Cifuentes and Malhotra. “Binary translation: static, dynamic, retargetable?” In: *1996 Proceedings of International Conference on Software Maintenance*, pp. 340–349. DOI: `10.1109/ICSM.1996.565037`.

[Das+20] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher. “Scalable validation of binary lifters.” In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 655–671. ISBN: 9781450376136. DOI: `10.1145/3385412.3385964`.

[Des12] F. Desclaux. “Miasm: Framework de reverse engineering.” In: *Actes du sstic. sstic* (2012).

[DF14] E. Daka and G. Fraser. “A Survey on Unit Testing Practices and Problems.” In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 2014, pp. 201–211. DOI: `10.1109/ISSRE.2014.11`.

[FR19] G. Fraser and J. M. Rojas. "Software Testing." In: *Handbook of Software Engineering*. Ed. by S. Cha, R. N. Taylor, and K. Kang. Cham: Springer International Publishing, 2019, pp. 123–192. ISBN: 978-3-030-00262-6. DOI: 10.1007/978-3-030-00262-6_4.

[GM16] V. Garousi and M. V. Mäntylä. "A systematic literature review of literature reviews in software testing." In: *Information and Software Technology* 80 (2016), pp. 195–216. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.09.002.

[Gua+10] H. Guan, H. Yang, Z. Qi, Y. Yang, and B. Liu. "The Optimizations in Dynamic Binary Translation." In: *2010 Proceedings of the 5th International Conference on Ubiquitous Information Technologies and Applications*. 2010, pp. 1–6. DOI: 10.1109/ICUT.2010.5677870.

[Haw+15] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao. "Optimizing binary translation of dynamically generated code." In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2015, pp. 68–78. DOI: 10.1109/CGO.2015.7054188.

[How77] W. Howden. "Symbolic Testing and the DISSECT Symbolic Evaluation System." In: *IEEE Transactions on Software Engineering* SE-3.4 (1977), pp. 266–278. DOI: 10.1109/TSE.1977.231144.

[Int23] Intel. *The Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. 2023.

[Jam+16] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad. "Software Testing Techniques: A Literature Review." In: *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. 2016, pp. 177–182. DOI: 10.1109/ICT4M.2016.045.

[Jan+16] S. R. Jan, S. T. U. Shah, Z. U. Johar, Y. Shah, and F. Khan. "An innovative approach to investigate various software testing techniques and strategies." In: *Int. J. Sci. Res. Sci. Eng. Technol* 2.2 (2016), pp. 682–689.

[KB13] P. Kedia and S. Bansal. "Fast dynamic binary translation for the kernel." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 101–115. ISBN: 9781450323888. DOI: 10.1145/2517349.2522718.

[Kim+17] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha. "Testing intermediate representations for binary analysis." In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017, pp. 353–364. DOI: `10.1109/ASE.2017.8115648`.

[Kin75] J. C. King. "A new approach to program testing." In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 228–233. ISSN: 0362-1340. DOI: `10.1145/390016.808444`.

[Kin76] J. C. King. "Symbolic execution and program testing." In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: `10.1145/360248.360252`.

[LGR11] G. Li, I. Ghosh, and S. P. Rajan. "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs." In: *Computer Aided Verification*. Berlin, Germany: Springer, 2011, pp. 609–615. ISBN: 978-3-642-22110-1. DOI: `10.1007/978-3-642-22110-1_49`.

[Mar+09] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. "Testing CPU emulators." In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA '09. Chicago, IL, USA: Association for Computing Machinery, 2009, pp. 261–272. ISBN: 9781605583389. DOI: `10.1145/1572272.1572303`.

[Mar+12] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. "Path-exploration lifting: hi-fi tests for lo-fi emulators." In: *SIGARCH Comput. Archit. News* 40.1 (Mar. 2012), pp. 337–348. ISSN: 0163-5964. DOI: `10.1145/2189750.2151012`.

[MFL10] W. Ma, A. Forin, and J.-C. Liu. "Rapid prototyping and compact testing of CPU emulators." In: *Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping*. 2010, pp. 1–7. DOI: `10.1109/RSP.2010.5656339`.

[Pro02] M. Probst. "Dynamic binary translation." In: *UKUUG Linux Developer's Conference*. Vol. 2002. 2002.

[Roc+22] R. C. O. 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*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 888–902. ISBN: 9781450392655. DOI: `10.1145/3519939.3523719`.

[SBC12] A. A. Sawant, P. H. Bari, and P. Chawan. "Software testing techniques and strategies." In: *International Journal of Engineering Research and Applications (IJERA)* 2.3 (2012), pp. 980–986.

[Sen07] K. Sen. "Concolic testing." In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, pp. 571–572. ISBN: 9781595938824. DOI: `10.1145/1321631.1321746`.

[Sho+16] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis." In: *IEEE Symposium on Security and Privacy*. 2016.

[Sit+93] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. "Binary translation." In: *Commun. ACM* 36.2 (Feb. 1993), pp. 69–81. ISSN: 0001-0782. DOI: `10.1145/151220.151227`.

[SMA05] K. Sen, D. Marinov, and G. Agha. "CUTE: a concolic unit testing engine for C." In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 263–272. ISSN: 0163-5948. DOI: `10.1145/1095430.1081750`.

[Sun+23] L. Sun, Y. Wu, L. Li, C. Zhang, and J. Tang. "A Dynamic and Static Binary Translation Method Based on Branch Prediction." In: *Electronics* 12.14 (2023). ISSN: 2079-9292. DOI: `10.3390/electronics12143025`.

[Tea24] T. L. Team. *xn–jo8h LLDB*. [Online; accessed 12. Apr. 2024]. Apr. 2024.

[YCW11] H. Yu, C. Y. Chung, and K. P. Wong. "Robust Transmission Network Expansion Planning Method With Taguchi's Orthogonal Array Testing." In: *IEEE Transactions on Power Systems* 26.3 (2011), pp. 1573–1580. DOI: `10.1109/TPWRS.2010.2082576`.

[Z3p24] Z3prover. *z3*. [Online; accessed 11. Mar. 2024]. Mar. 2024.

[Zhu+22] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang. "Fuzzing: A Survey for Roadmap." In: *ACM Comput. Surv.* 54.11s (Sept. 2022). ISSN: 0360-0300. DOI: `10.1145/3512345`.