Focaccia: A Framework for Testing Binary Translators

Theofilos Augoustis
Technische Universität München

Abstract

Program emulators are widely used to execute programs on different computer architectures that are not directly supported by the program code. However, emulators must contend with many possible *correctness* issues due to the inherent complexity of existing ISAs and the major differences in the design of each ISA. Existing testing infrastructures partially alleviate this issue but lack *comprehensiveness* in their verification.

Focaccia is a user-level emulator testing and verification framework designed to augment existing testing infrastructures to enable comprehensive testing. This report makes the following contributions: a black-box program-state verifier that can detect correctness issues, a reproducer generator that produces programs that exhibit a detected issue and a framework for building additional verification tools. The design of these tools results directly from the formal foundations of emulators, which are utilized to comprehensively test program execution for a given input.

1 Introduction

The computing architecture underlying many currently deployed systems is undergoing a shift from the dominating x86 architecture to ISAs such as Arm and RISC-V [5, 13]. The key goals of this transition include better performance and energy efficiency, as well as more permissive licensing. An important point is that this transition affects both commercial deployments, as well as consumer hardware.

However, porting applications across this diverse set of architectures is not straightforward since existing applications are not compatible across different ISAs. Unfortunately, recompiling for the new target architecture is often not an option since the source code of these application may no longer be available or it may depend on architecture-specific intrinsics [3]. Similarly, even when source code is available, large-scale refactoring may be necessary to adapt a codebase to the new architecture.

Emulation is a widely deployed approach to enable existing programs to run on a new architecture, since it supports running programs without source code changes and without recompilation. For instance, an emulator underlies the execution environment for x86 programs on Apple M1 chips [18], which use an Arm-based architecture [4]. Emulators are also integrated and widely used on Linux systems through the binfmt technology [16].

Existing emulators operate directly on the application binary, employing either interpretation or binary translation to execute it on the host ISA. In particular, they translate the instructions of the application binary, taken at some granularity, to instructions of the host ISA. This translation needs to fully emulate the semantics of the guest instructions in terms of those provided by the host ISA to achieve correctness. Additionally, it must be efficient to minimize the runtime delay relative to the running time of the program on the native architecture.

A number of challenges make providing both correctness and performance difficult. One challenge is the complexity inherent in the design of modern ISAs, which may include more than a thousand instructions [14, 15]. Another challenge are the differences between architectures, such as the inclusion of architectural flags, the number of architectural registers and the level of support for different types of instructions. Finally, existing emulators tend to support multiple combinations of host and guest ISAs, which requires additional abstractions to avoid duplicating their core translation logic [6].

Considering performance, modern emulators utilize techniques developed for compilers, such as the use of an Intermediate Representation (IR) and optimization passes over that IR [10]. A constraint here is that optimization must be performed quickly to ensure that the overhead of optimizing the translation does not exceed the speedup gained by executing optimized code [21]. Another constraint is that the optimization pass must preserve the semantics of the guest program, since an incorrect translation may produce different behaviour from native execution in various edge cases.

Correctness has traditionally been verified for complex software using a well defined testing and verification suite, which can discover unexpected errors and guide development. Modern emulators include a large number of handwritten and automated tests that verify their behaviour [11]. These tests are neither complete, nor comprehensive. In particular, they do not verify that an emulated program executes semantically equivalent code at every point of its execution. They are also largely written by the same developers that may embed incorrect assumptions as part of the test, decreasing their effectiveness at verifying correctness.

This report presents the design and implementation of Focaccia, a comprehensive verification framework for emulator testing. Focaccia aims to augment existing testing mechanisms with a comprehensive verifier for completely testing the execution of an emulator for an existing set of programs. It aims to do so while remaining emulator and platform independent at its core, enabling extensibility through the implementation of backends for supporting new emulators. It also aims to simplify development by detecting errors, displaying the precise location in which they occur and providing reproducers for sharing between developers.

Background and Motivation

Emulation

Program emulation is the practice of executing a program designed for a given platform on another platform. An emulator is used for this purpose, which maps the semantics of the program to the semantics provided on the new platform. This emulator may operate at various levels, depending on the actions needed to achieve correct emulation. For instance, it may interpose calls to system calls to implement them in terms of those provided by the native system.

User-level cross-ISA program emulation is the emulation of a program designed for some ISA (called the guest ISA) on a different architecture (called the host ISA). For instance, QEMU can execute programs written for the x86 architecture on the Arm architecture [6]. Typical, cross-ISA emulators use either interpretation or binary translation [21].

In the case of interpretation, they iterate over the instructions of the guest program and perform the actions specified by that instruction on their internal state. For binary translation, the guest instructions are directly translated into host instructions and subsequently executed.

Binary translation can be performed either *statically* or *dynamically*. Static binary translation can fully translate the instructions of a program prior to its execution, producing a new program for the host ISA that is semantically equivalent to the original. Dynamic binary translators perform this process at runtime by translating dynamic basic blocks into semantically equivalent dynamic basic blocks for the host ISA that are executed by a runtime component. In each case, the binary translator may perform extensive optimizations on the generated instructions or even merge multiple basic blocks into a superblock.

Modern emulators use dynamic binary translation extensively, due to its superior performance over interpretation. Interpretation is also widely used, especially for binary translators focusing on complete support for host ISA instructions or those developed primarily aiming at correctness. Static binary translation is largely unused for cross-ISA emulation due to the undecidability of static disassembly of binaries for currently used ISAs that renders it fundamentally incomplete [12].

Testing and Verification

Emulators utilize multiple distinct types of testing to verify their correctness, executed as part of a testing suite. Each testing methodology verifies certain aspects of the system, while a testing suite utilizes multiple methodologies to broadly check expected behaviour.

The main techniques used for testing are the following:

- Unit Testing: tests are used to verify the correctness of a restricted part of the system, such as the process of lifting to an internal IR. Many emulators use a large number of unit tests and often require contributors to include new ones when adding new features.
- Functional Testing: tests are used to verify that emulation for a given program is performed correctly. Although modern emulators include many functional tests, they are not comprehensive and only verify a representative set of programs.
- Fuzzing: new test cases are produced by fuzzing tools, based on a seed or existing test cases. Increased interest in fuzzing exists due to its ability to discover errors not generally found by handwritten tests.

The above techniques are generally able to verify correctness for many representative use-cases and some edge cases. However, they are neither complete nor comprehensive. Specifically, they do not test whether incorrect behaviour appears at some point during the tests execution, merely that the end result is correct. They also only verify an expected state but cannot detected unexpected state changes, since the test does not check them.

Symbolic Execution

Symbolic execution is the technique of executing a program with symbolic, rather than concrete, values as input [17]. Specifically, these symbolic values are mutated according to the instructions of a program, such that they can be described via symbolic equations. Each symbolic equation may refer to multiple symbolic values and precisely describe the operations performed to yield a given mutation of a symbolic value, which is a major benefit of this approach. On the other hand, symbolic execution cannot cope with branching control flow and may lead to an explosion of possible states for large enough programs.

Symbolic execution has been applied at various different representations of programs. For instance, there exist symbolic execution engines that operate at the level of

```
// AND RSP, 0xFFFFFFFFFFFFFFF0
ZF  = (RSP & 0xFFFFFFFFFFFFFFF0) ? (0x0, 0x1)
NF  = (RSP & 0xFFFFFFFFFFFFFFF0)[63:64]
PF  = parity(RSP & 0xF0)
OF  = 0x0
CF  = 0x0
RSP = RSP & 0xFFFFFFFFFFFFFFF0
RIP = 0x401043
```

Figure 1: Symbolic equations generated for the x86 `AND RSP, 0xFFFFFFFFFFFFFFF0` instruction. Note that each affected flag is treated as an individual register and that mutations for some registers reference their current symbolic value

source code written in a high-level programming language. Symbolic execution is also possible at the level of instructions in an ISA, which requires a symbolic execution engine capable of encoding the semantics of the ISA.

The major limitation of symbolic execution is in the type of code that it can handle without an explosion in its state space. Specifically, loops may continue or terminate depending on symbolic values, such that symbolic execution engines must consider multiple possible execution paths in parallel. Given sufficiently complex loops, the number of states becomes to large to track effectively and symbolic execution becomes infeasible.

An adaptation of symbolic execution is concolic execution, which performs dynamic analysis using symbolic values and determines control flow using concrete values [20]. In such a scheme, the concrete values are updated by the program normally, while the symbolic value equations are maintained separately by the symbolic execution engine. This approach avoids the state explosion issue of pure symbolic execution, while sacrificing the exploration of different programs path.

Overview

System Overview

Focaccia consists of a program verifier and a reproducer generator. These tools interact with a symbolic execution engine, a native program trace collector and an emulator. The emulator interface is provided via an emulator backend, which abstracts the required instrumentation of the emulator for trace collection. A JSON-based common data format is also used to abstract the core logic of Focaccia from the specific traces collected.

A number of utility tools are provided to enable offline tracing and symbolic execution log collection, as well as conversion from collected traces to the common JSON format.

The primary interface to Focaccia is a Python script called *focaccia*, which executes programs and verifies their correctness relative to a user-provided oracle.

System Workflow

Figure 2 shows how an emulator is tested using Focaccia, a sample input binary and any inputs required for it. In the sample phase, an emulator log is collected ① that contains the guest emulated state, including both registers and the allocated memory. Additionally, the program is executed natively using an LLDB-based trace collector that samples the state of the program ②.

The sampled state of the native program is analyzed by our symbolic executor on a per-instruction basis to determine the symbolic transformations that occurred on the state ③. The results are appended to a symbolic log that denotes all the mutations that occurred in the program for each instruction, which is processed along with the emulated guest state to yield the next expected state of the emulator ④.

This processing enables Focaccia to abstract over differences in the emulated guest state, which may result from differences in the layout of the address space, defined environment variables or auxiliary vectors. The resulting states can be directly compared in the state checker for mathematical equality ⑤. Focaccia only needs to verify that the mutated values have changed as expected to determine correct execution. However, it can optionally verify all architectural registers and memory with a trade-off in terms of its runtime performance.

Finally, all errors are added to an error log ⑥. The error log is maintained to enable Focaccia to track all discovered errors so that it may compute statics and create reproducers. Focaccia by default stops on the first error but may optionally continue to discover more errors, which effectively finds parts of the program that depend on the discovered error.

Optionally, Focaccia can generate reproducer programs based on the errors discovered in the error log ⑦. The generated reproducers merely setup the program state as required and include the instruction or group of instructions that triggered an error. This allows small programs to be generated that can be directly shared with other developers or used as a seed for test case generation.

Design Challenges and Key Ideas

Focaccia operates in terms of the program state and the emulated guest state for verifying correctness. The *program state* is considered the set of architectural register values and the entirety of main memory after the execution of some instruction. A program state exists

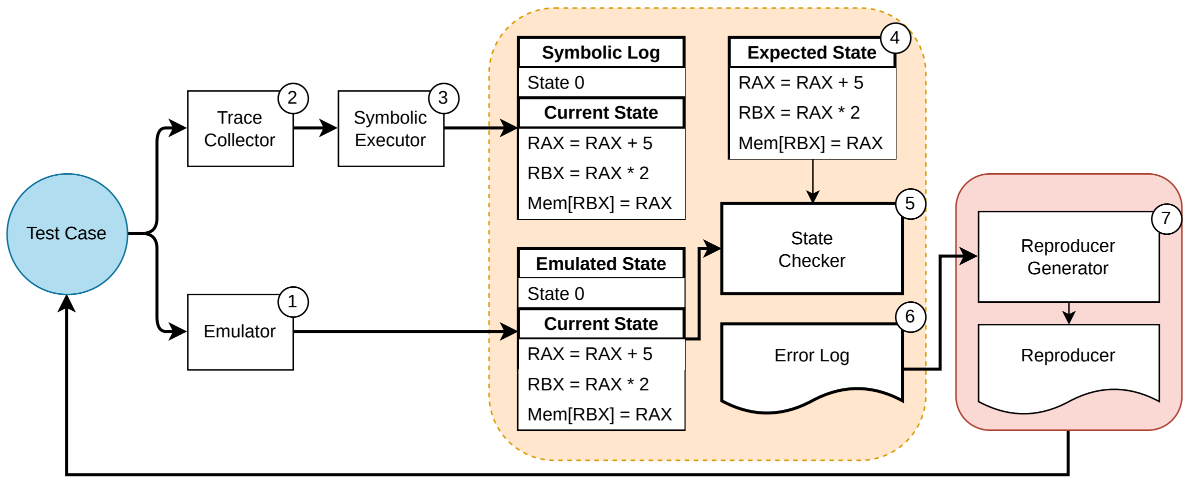

Figure 2: An overview of FOCACCIA. First, a test consisting of a program and its inputs are executed by ① an emulator and ② a LLDB-based trace collector. The native traces are converted to symbolic equations ③ and processed to yield an expected state ④ using the emulated guest state. The resulting expected state is compared to the emulated guest state for equality ⑤, resulting in a possible error log ⑥. Errors are optionally passed to the reproducer generator to create a reproducer program ⑦.

after each instruction executed natively within the hardware executing the program. Similarly, emulators also maintain such a state that is emulated to match the behaviour of the original program, called the *emulated guest state*.

Ensuring comprehensiveness in emulator testing requires verifying that the emulator executes correctly every guest instruction in the program. This is infeasible for many emulators, since they implement optimizations that cause the generated host ISA instructions to be semantically different locally, while semantically identical globally within the execution of a program.

These design challenges are overcome utilizing three main ideas in FOCACCIA. First of all, the system must extract the semantics of program execution, such that it can check the equivalence of the emulator state with the state of native execution. Second of all, the system must check that these equivalences hold, making native execution the oracle for verifying correctness. Third of all, it must support this workflow at the smallest granularity supported by the emulator.

Key Idea #1: Concolic Execution for Equivalence Checking. Due to differences in address layouts and other emulator-dependent data, it is not possible to mathematically compare program state values to those of gathered from native execution. Instead, it is necessary to perform an equivalence check that verifies whether the semantics of the program are maintained. FOCACCIA uses concolic execution at the level of individual guest instructions on the native platform to determine those semantics.

Key Idea #2: Native Execution as Oracle. The expected semantics of a program are those exhibited by it when executed natively. This holds for the overall expected program behaviour but also for the state mutations occurring at every single point in its execution. Assuming such a view of native execution, it can be used as a perfect oracle for verification purposes.

Key Idea #3: Granular Verification. A program can be verified at the granularity of a guest instruction to determine whether its translation mutates state as expected. This is not possible when optimizations are used, since the translations for individual guest instructions may be incorrect locally but correct when considered together with other instructions in the same block. FOCACCIA makes no assumption on the verification granularity, which include individual instructions, dynamic basic blocks, superblocks or other granularities. It does, however, require the emulator backend to define the type of block that should be verified for a match.

Program Verifier

Ensuring comprehensive verification requires checking that the emulated program mutates its state in the same way as the original program. However, the emulated guest state is generally different than the state of the program executing natively, since addresses and other platform-dependent data differ. As a result, it is not possible to compare state values for equality but it is necessary to test them for semantic equivalence. The program verifier in Focaccia implements this type of *equivalence checking* by extracting program semantics through concolic execution, determining the equivalence relations between the states and then checking that they match.

Focaccia uses the Miasm symbolic execution engine to extract program semantics, while taking into account the limitations of symbolic execution and its own domain-specific restrictions. First of all, Focaccia uses concolic execution to drive program execution, while capturing symbolic equations for each instruction. This allows us to avoid the state explosion problems present in typical symbolic execution by only maintaining dependencies on symbolic state at the level of a single instruction, while executing with concrete values. Additionally, since symbolic equations are available at the granularity of an instruction, they can be combined to describe arbitrary sequences of instructions. Finally, symbolic equations describe the entire state of the program, including both registers and memory, which allows later checks to have complete semantic information and use it when corresponding concrete values from an emulator are available.

Besides semantic information, the emulated guest state is also traced at the smallest granularity supported by the emulator. This is done in an emulator-specific manner, which is encoded as part of an emulator backend for Focaccia. The design of emulator backends are further described in the emulator interface section. The collected tracing information is sufficient for the verifier to check correctness by examining whether the emulated guest state matches the corresponding symbolic equations.

The equivalence check is performed on two different emulator states at a time. The current state is taken and used as an input to the symbolic equations previously sampled. This yields the correct next state of the emulator, which is checked with the actual next state. Since the symbolic equations have been applied on the emulator state, the expected state will be in terms of the same address layout and other emulator-dependent data. As such, the actual equivalence check is only a comparison for equality on the expected state and the next state. An important optimization here is that Focaccia uses symbolic equations to only check the state values that were supposed to change, rather than the entirety of the state. Depending on the chosen granularity, this may mean only a few register and memory values, rather than the entire emulated architectural state and memory contents.

Any mismatch detected during the equivalence check will result in an error being stored in the error log, along with diagnostic information printed to the user. Each mismatch may occur due to one of the reasons described below:

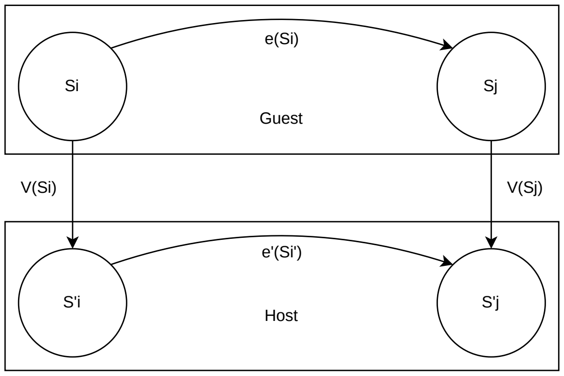

Figure 3: Emulation is defined via the construction of an *isomorphism* between the guest and the host [19, 21]

- Incorrect state values: emulator performed an incorrect translation.
- Incomplete state values: errors may exist but verifier cannot detected them.
- Symbolic execution inconsistency: symbolic equations are wrong and verifier cannot determine errors using them.
- Emulator failed unexpectedly: segmentation fault or similar.

In effect, this type of checking makes the native execution of the program an oracle for the verifier. In particular, using its extracted semantics to determine correctness ensures that we verify that the emulator matches the exact behaviour of the original program. This is equivalent to verifying the isomorphism between the two execution environments, as part of the formal definition of emulator [19, 21]. The verified isomorphism type is shown in Figure 3.

Most importantly, Focaccia can verify that this isomorphism holds at lowest supported instruction group granularity for a given program with a given input. It expects to be able to run both the oracle and sample data from the emulator as needed, while being able to verify correctness without modifying the state of the emulator. Additionally, the verifier can ensure that subsets of this isomorphism hold, such as the equivalence between register values, even when other parts cannot be verified.

This type of verification is heavily reliant on the correctness of the symbolic executor. Focaccia implements a mechanism to ensure that errors in a symbolic executor do not produce false positives during verification. In particular, the verifier applies the extracted symbolic

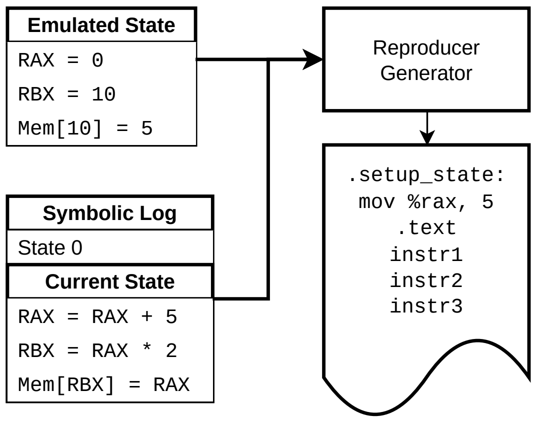

Figure 4: Reproducer workflow

equations to both the expected state and the emulated guest state, such that it can verify whether the next expected state matches those equations. A mismatch is considered an incorrect symbolic equation, which causes Focaccia to raise a warning.

Reproducer Generator

As shown in Figure 2, Focaccia includes the option to generate reproducers for detected errors. These reproducers contain significantly less code than the original program, while still reproducing a known error, such that they can be shared and run by other developers. However, reproducers also provide an avenue for generating new test cases from other complex programs, such that they can be used for further verification.

The reproducer generator is designed to extract the failing instructions from the program code and the program state prior to the error being triggered. Generating a reproducer can be achieved by setting the same state and then executing the failing instruction. In particular, the reproducer generator creates an assembly program, consisting of a preamble that sets the state followed by the failing instructions. Since the state encompasses both registers and memory, the generator utilizes information from the symbolic equations to minimize the actual state that it must set up to the values strictly needed for reproducing the error.

An important consideration is also that since verification is done at the smallest granularity supported by the emulator by default, the reproducer generator may only generate reproducers at the same granularity. This means that the reproducer generator can pinpoint the exact failing instruction or a group of failing instructions.

Figure 4 shows the workflow for generating a reproducer. The reproducer generator iterates over the error log and attempts to produce a reproducer for each. It is passed the symbolic log and the emulator state, such that it can utilize them directly. For each given error, the reproducer generator uses the symbolic log to determine the symbolic values that the erroneous sequence of instructions is dependent on.

Data Format

One of the main design goals of Focaccia is to use its core verification logic independently of the tested emulator. As such, an emulator interface must be implemented to facilitate the collection of required data for verification. This interface is expected to be emulator dependent, but it must not be invasive or require significant modification to the logic of the emulator.

Focaccia uses a JSON-based common data format as the interface to its core verification logic. It translates the traces collected from the emulator to a JSON document with a minimal schema designed to remain independent of any particular emulator. The JSON documents includes a sequence of architectural register values and memory contents for each state, collected at the smallest possible granularity. The instruction at the given state is not recorded, since it can be determined directly via the instruction pointer. The document may also have only partial values for a given state, it may contain only the precise memory contents needed and may reference a BLOB for the memory image, as needed. This flexibility is necessary to support a wide variety of emulators, each providing different tracing mechanisms.

A major concern is the handling of very large memory dumps, which are necessary for verifying memory operations. Focaccia supports verification using the entire allocated memory image of the guest within the emulator, which may be infeasible to represent and use. There exist two solutions for handling memory data: *(i)* storing each memory snapshot in a file referenced by the JSON document or *(ii)* extracting only those memory contents referenced within a given sequence of instructions.

An example of a log with a sample state dump is shown in Listing 1. The log consists of a preamble defining the architecture and the environment in which the states were sampled. The environment includes both the binary name and a SHA256 hash computed over its contents, which are used to ensure that the same versions are compared during verification. Different versions of the same binary are likely to differ in their trace captures, such that verification becomes meaningless. Additionally, the program arguments and the system environment variables are stored to allow the verifier to execute the program natively with the same environment and invocation parameters. Finally, a sequence of snapshots is provided that contains the actual register state and the memory state or a reference to a BLOB containing it.

```
{
    "architecture": "x86_64",
    "env": {
        "binary_name": "test_program",
        "binary_hash": "",
        "argv": [],
        "envp": []
    },
    "snapshots": [
        {
            "registers": {"RAX": 0, "R10": 0,
                ↪ "RIP": 4198450, "RDX": 0, ...},
            "memory": ""
        }
    ],
}
```

Listing 1: Example of a state dump in the JSON common format for a single instruction at a given snapshot. captionpos

Another major concern is the granularity at which emulators expose their emulated guest state. An emulator utilizing interpretation may expose this state at the level of a single instruction, while emulators utilizing binary translation usually expose it at the granularity of a basic block or superblock. FOCACCIA handles this uniformly by identifying each state with its address without imposing any particular granularity. The emulator backend invoked implements an interface that enables the verifier to determine the granularity of a given state.

Emulator Interface

The emulator interface is handled by an emulator backend within FOCACCIA. The emulator backend may instrument the emulator in any available way but is expected to not make invasive changes to the emulator codebase. Furthermore, the emulator backend is responsible for converting dumped guest states from the emulator into the JSON common data format. The backends for two different emulators are described, QEMU [6] and Arancini. Note that both systems operate at the level of a dynamic basic block and log collection is handled at that level.

In the case of QEMU, log collection uses the existing GDB stub support in QEMU to gather register values and memory [9]. Since GDB directly supports dumping register values, the log collector only needs to invoke its API to get them and store them in the trace log [8]. Memory is handled differently due to the large overhead of dumping the entire process image at each basic block. FOCACCIA already has access to the symbolic log, representing the expected symbolic transformations of the program, during log collection from QEMU. As such, it uses the gathered symbolic transformations, along with the register state dumped by QEMU, to access only memory values that are modified as part of a basic block.

A different approach is taken for Arancini, the experimental emulator developed at TUM. Since Arancini does not provide a GDB stub, log collection relies on invoking Arancini with its logging facilities enabled and then extracting useful data. In particular, Arancini supports a compile-time and runtime configurable logger that can dump the state of register values, among other debug output at a basic block granularity. The Arancini backend within FOCACCIA collects these logs, extracts the dumped register values and stores them to the JSON log. The resulting log in this case is much more limited, since it provides no memory values and requires the verifier to operate only on registers.

Although these backends provide very different logs to the verifier, in terms of completeness, they are both usable for determining execution errors. In particular, the QEMU log is complete for the verifier to comprehensively test a program, while the Arancini log raises a large number of warnings but can still detect incorrect register values, as long as they do not depend on memory data.

Implementation

FOCACCIA is implemented as a Python program and includes other Python components also. It relies on a number of libraries, which are critical to its implementation. Specifically, the trace collection logic relies on the LLDB debugger API for collecting data from the QEMU GDB stub [9]. We also rely on the Miasm symbolic execution engine for implementing core part of the verification logic [7].

The use of Miasm was motivated by its wide support for the x86 architecture and other architectures. Another important factor was its ease of use and existing Python interface, which enabled a relatively simple integration into FOCACCIA. Given the use of a symbolic log to determine correctness, it is imperative that the symbolic log correctly correspond to the behaviour of the oracle. Although Miasm is capable of constructing such a log for a large number of x86 instructions, it is notably incomplete and required changes to support all instructions of interest in our tests.

Evaluation

The evaluation is centered around two main components: the verifier and the reproducer generator, which are presented in order.

Verifier Evaluation

The verifier is evaluated in terms of its ability to correctly determine execution errors in single-threaded programs executed under QEMU. The main programs considered are programs that are known to cause execution errors for particular versions of QEMU, taken directly from the QEMU mailing list [1].

Specifically, we verify that the program in Listing 2 that executes incorrectly on QEMU version 6.1.0-rc1 [2] can be used to reproduce an error with Focaccia.

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

Listing 2: This program executes an x86 `CMPXCHG` instruction that should set `EAX` to 0 but QEMU sign extends it and sets `RAX` to 0 instead.

The following sequence of commands is necessary for Focaccia to reproduce the error:

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

Listing 3: Command sequence to verify QEMU using Focaccia

This results in the error log from Listing 4 being produced.

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

[Symbols]
Expected transformation: 0x40116a -> 0x40116e:
ZF = (-@32[RBP + -20]) == (-RAX[0:32])
RAX = ((-@32[RBP + -20]) == (-RAX[0:32]))
    ? (RAX, {@32[RBP + -20] 0 32, 0 32 64})
RIP = 0x40116E
@32[RBP + -20] = ((-@32[RBP + -20]) == (-RAX[0:32]))
             ? (RDI[0:32], @32[RBP + -20])

[Instructions]
CMPXCHG DWORD PTR [RBP + -20], EDI

Actual difference (snapshot of x86_64 state):
{'RBP': '0', 'RAX': '0xedcba98800000000', 'RIP': '0x4'}
```

Listing 4: Verifier error log for `CMPXCHG` behaviour

The produced error log shows that the expected transformation does not match, since the snapshot of the state has different values than expected. Besides the different values shown, the error log also clearly denotes the relations that need to hold between those values, which are violated by the emulator.

Reproducer Generator Evaluation

The reproducer generator is evaluated in terms of the size reduction for the generated programs, relative to the original program. The same evaluation programs are used for the reproducer generator as for the program verifier, since the reproducer generator depends on errors discovered by the verifier.

Considering the program described in Listing 2, it generates a binary of roughly 30 KiB. A large proportion of the size is the C standard library, which is not directly needed for reproducing the error. The verifier reproducer can achieve a reduction to 5 KiB by eliminating the C standard library code and only setting the required state to reproduce the issue.

Executing the verifier yields a log of the form shown in Listing 5, which shows that the error is still properly detected.

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

[Symbols]
Expected transformation: 0x40116a -> 0x40116e:
ZF = (-@32[RBP + -20]) == (-RAX[0:32])
RAX = ((-@32[RBP + -20]) == (-RAX[0:32]))
    ? (RAX, {@32[RBP + -20] 0 32, 0 32 64})
RIP = 0x40116E
@32[RBP + -20] = ((-@32[RBP + -20]) == (-RAX[0:32]))
             ? (RDI[0:32], @32[RBP + -20])

[Instructions]
CMPXCHG DWORD PTR [RBP + -20], EDI

Actual difference (snapshot of x86_64 state):
{'RBP': '0', 'RAX': '0xedcba98800000000', 'RIP': '0x4'}
```

Listing 5: Verifier error log for `CMPXCHG` behaviour but executing from the reproducer

The resulting error log is notable due to the decrease in runtime to reproduce the error, as well as the overall smaller binary size. Furthermore, the generated reproducer is notable because the setup code can be abstracted, such that a developer can reason simply about the particular section of the program code that is mistranslated.

The generated reproducer is shown in the Appendix, under Listing 6.

Future Developments

Although Focaccia is capable of detecting correctness issues in a large variety of single-programs, it is limited in its support of multithreading and randomness. It also does not verify that the emulator correctly interfaces the program with the rest of the system, meaning that program side-effects such as writing a file could simply not happen and FOCACCIA would not detect an error.

One goal for the future is to expand support for handling randomness in programs. Existing research has proposed a number of approaches for verifying distributed systems, such as simulating RNGs, emulating the clock and so on [22]. Applying these ideas to FOCACCIA would yield a resilient debugging tool, even for programs using random numbers, communicating over sockets or via the filesystem.

Another goal would be to support multithreaded program verification, which is currently not possible in FOCACCIA. This would be relatively difficult to achieve with the current architecture, due to the expectation that guest programs match the oracle. This expectation clearly fails when guest programs may have threads that are scheduled in a different order to the ones running in the oracle. It is yet unclear how support may be expanded to verifying multiple threads at a time.

Availability

The source code for Focaccia is open-source and available on GitHub.

References

[1] QEMU: Contribute/MailingLists, March 2024.

[2] QEMU Issue Tracker: x86_64 cmpxchg behavior in qemu tcg does not match the real CPU, March 2024.

[3] Apple. Porting Your macOS Apps to Apple Silicon. https://developer.apple.com/documentation/apple-silicon/porting-your-macos-apps-to-apple-silicon, 2024.

[4] Apple-Silicon. Addressing architectural differences in your macos code, 2020. Available at https://developer.apple.com/documentation/apple-silicon/addressing-architectural-differences-in-your-macos-code.

[5] Amazon AWS. Aws graviton processor. https://aws.amazon.com/ec2/graviton.

[6] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[7] Fabrice Desclaux. Miasm: Framework de reverse engineering. *Actes du sstic. sstic*, 2012.

[8] The GDB Developers. Gdb: The gnu project debugger. Available at https://www.sourceware.org/gdb/documentation.

[9] The QEMU Project Developers. Gdb usage. Available at https://qemu-project.gitlab.io/qemu/system/gdb.html.

[10] The QEMU Project Developers. Tcg intermediate representation. Available at https://www.qemu.org/docs/master/devel/tcg-ops.html.

[11] The QEMU Project Developers. Testing in qemu. Available at https://www.qemu.org/docs/master/devel/testing/main.htmhttps://www.qemu.org/docs/master/devel/testing/main.htmll.

[12] R. N. Horspool and N. Marovac. An Approach to the Problem of Detranslation of Computer Programs. *The Computer Journal*, 23(3):223–229, 08 1980.

[13] Agam Shah -- The Register. We're closing the gap with arm and x86, claims sifive: New risc-v cpu core for pcs, servers, mobile incoming. https://www.theregister.com/2021/10/21/sifive_riscv_cpu/, 10 2021.

[14] Intel. X86 encoder decoder (xed), 2022.

[15] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2*, 2007.

[16] The kernel development community. Kernel support for miscellaneous binary formats. Available at https://www.kernel.org/doc/html/latest/admin-guide/binfmt-misc.html.

[17] James C. King. A new approach to program testing. *SIGPLAN Not.*, 10(6):228–233, April 1975.

[18] Koh M. Nakagawa. Project Champollion: Reverse engineering Rosetta 2, 2021.

[19] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[20] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 571–572, New York, NY, USA, 2007. Association for Computing Machinery.

[21] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes.* The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[22] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2653–2666, New York, NY, USA, 2021. Association for Computing Machinery.

Appendix

```
.section .text
.global _start
_start:
_stack:
mov ax, 0x1234
push ax
mov ax, 0x5678
push ax
mov ax, 0x0000
push ax
mov ax, 0x0000
push ax
sub rsp, 0
_setup_regs:
mov rdi, 0x77777777
mov rax, 0x1234567812345678
_bb:
cmpxchg dword ptr [rsp + 0x4], edi
_exit:
mov rax, 60
mov rdi, 0
syscall
```

Listing 6: Generated reproducer for the `CMPXCHG` error described in Listing 2