Focaccia: Semantic Validation for CPU Emulators

Theofilos Augoustis
TU Munich
Munich, Germany
theofilos.augoustis@tum.de

Sebastian Reimers
TU Munich
Munich, Germany
sebastian.reimers@tum.de

Christian Krinitsin
TU Munich
Munich, Germany
christian.krinitsin@tum.de

Redha Gouicem
RWTH Aachen University
Aachen, Germany
gouicem@cs.rwth-aachen.de

Dennis Sprokholt
TU Munich
Munich, Germany
dennis.sprokholt@tum.de

Pramod Bhatotia
TU Munich
Munich, Germany
pramod.bhatotia@tum.de

Abstract

CPU emulators are widely used to execute programs across different ISAs, but their implementation complexity makes them prone to *mistranslation* errors, where guest instructions are incorrectly translated to the host. These errors subtly corrupt program state or control flow, yet often surface only much later in execution, making them difficult to detect and localize with existing validation methods.

We propose Focaccia, our novel semantic validation system that automatically detects, localizes and reproduces mistranslation errors in unmodified CPU emulators. Focaccia bridges formal methods with systems by defining a formal model of emulation correctness based on state transition systems and building a translation validator around that model. To validate an emulator, Focaccia compares emulated behavior against native semantics captured through speculative concolic tracing, while cross-ISA record-replay handles OS-level non-determinism.

We evaluate Focaccia on QEMU and Box64 across multiple ISAs, detecting and localizing 17 known mistranslation errors across six different bug categories, all of which escape existing emulator test suites. Focaccia further identified eight previously unknown mistranslations, all subsequently confirmed and fixed. Focaccia scales to full applications, including SQLite, Curl and the Lua interpreter, tracing and validating a 99,000-instruction SQLite execution in approximately 5 minutes. Generated reproducers reduce program size by up to 99%, isolating errors to single basic blocks.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; **Dynamic analysis**; *Virtual machines*; • **Theory of computation** → *Operational semantics*.

ASPLOS '27, Heraklion, Greece

© 2027 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece*, https://doi.org/10.1145/3845814.3850076.

Keywords: CPU emulation; Translation validation; Symbolic execution; Deterministic replay; Bug reproduction

ACM Reference Format:
Theofilos Augoustis, Sebastian Reimers, Christian Krinitsin, Redha Gouicem, Dennis Sprokholt, and Pramod Bhatotia. 2027. Focaccia: Semantic Validation for CPU Emulators. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27), April 11–15, 2027, Heraklion, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3845814.3850076

1 Introduction

Modern computer systems are increasingly heterogeneous, incorporating new Instruction Set Architectures (ISAs) such as Arm and RISC-V [20, 35]. These platforms promise improved performance, energy efficiency, and open hardware ecosystems, but they also raise expectations for seamless support of the vast existing software base [16].

Emulation is the primary solution that enables programs to run without source code changes or recompilation across architectures [15, 17, 21, 34, 65, 69]. Beyond compatibility, emulators are essential for dynamic analysis, instrumentation, and developer tooling [23, 36, 55].

Despite their importance, CPU emulators are difficult to implement correctly for three primary reasons: *(i)* state-of-the-art translators, such as QEMU [21], can emulate dozens of ISAs across a wide range of host architectures, where each host–guest pair introduces its own porting challenges; *(ii)* the ISAs themselves are often ambiguously specified [19, 63], leaving emulation correctness on shaky ground; and, finally, *(iii)* the compiler optimizations used by emulators are difficult to implement correctly [46, 49]. These difficulties cause developers to easily overlook edge cases, resulting in emulator implementation bugs.

Among various causes of emulator bugs, we focus on *mistranslation errors*, where an instruction from the guest architecture translates to host instructions that behave differently (*e.g.* translating an x86 `ADD` to an Arm `MUL`). Mistranslation errors are particularly critical because they silently corrupt program state and control flow, often surfacing only much later in emulated execution [3, 8, 39, 50, 51], ultimately invalidating all assumptions about a program's behavior.

To substantiate the significance of mistranslation errors in CPU emulation, we study all user-mode bugs reported in the last decade for QEMU [21] and Box64 [27]. We find that mistranslations account for 10% of user-mode bugs in Box64 and over 22% in QEMU, despite QEMU being significantly more mature and extensively tested (§ 2.2). We identify six distinct root-cause categories that dominate mistranslations.

Unfortunately, existing testing approaches [11, 28] are insufficient to detect mistranslation errors. Developer-written tests only check observable behavior, such as program output, and miss mistranslations that corrupt unexamined program state. Fuzzers [6, 50, 51] automatically generate test programs, expanding coverage beyond a static test suite, yet they too can only detect mistranslations that manifest directly in program behavior. Beyond detection, no existing approach localizes mistranslations to the specific instructions that caused them.

Existing approaches consider translation correctness ad-hoc, without a formal characterization of emulation correctness, and thus can neither reliably detect nor localize mistranslations. Moreover, even with such a characterization, validating it in practice requires solving major systems-level challenges, such as efficiently tracing program semantics at scale and ensuring deterministic cross-architecture execution despite OS-level non-determinism. To this end, we aim to answer the following research question:

> **Research question.** How can we design an automated, formally grounded system to detect, localize and reproduce mistranslation errors in state-of-the-art emulators?

Addressing this question requires a *general, comprehensive, and automatic* validation system that: *(i)* works for any source and target ISA combination, *(ii)* across all stages of an optimizing emulator, and that *(iii)* generates reproducible test cases for detected mistranslations, to guide developers when fixing bugs in emulators.

To address this research challenge, we propose to *validate emulator translation correctness using program execution modelled as a state transition system* [54]. By treating both native and emulated execution as transition systems, we can formally relate their states [60]. When native and emulated states are related prior to executing a program fragment but not after, then that fragment must have been mistranslated. This formal foundation directly drives our systems design, determining what native semantics must be captured, what emulator state must be compared and what information is needed to reproduce a detected error.

We realize our core approach in Focaccia, a grey-box translation validation system built upon our formal model, requiring only knowledge of emulated program states. Focaccia consists of four main components: a native tracer, which uses speculative concolic execution [43, 68] to efficiently capture the program's semantics on the guest architecture; an emulator tracer, which collects emulated guest states inside the

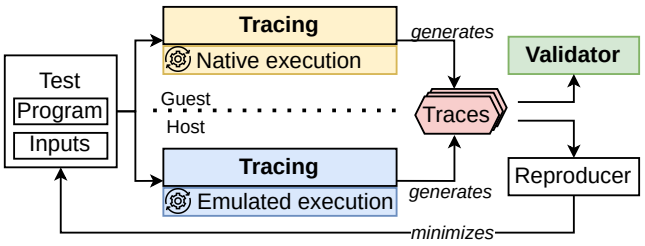


Figure 1. Overview of Focaccia's components: Tracing of native and emulated executions, validation, and reproduction

emulator, using cross-ISA record-replay to ensure deterministic execution; a translation validator, which compares the native and emulated program trace; and a reproducer generator, which generates a small program to reproduce the error. Fig. 1 shows the position of each component in the architecture.

Realizing automatic fine-grained emulator validation in Focaccia required solving five key technical challenges, each of which we address with a key idea:

(1) **Platform-independent validation:** Comparing emulated and native states is difficult because registers and memory are represented differently across platforms. Our key idea is to *represent states symbolically* [43] with logical formulae, making them independent of platform specifics.
(2) **Mismatching intermediate states:** Emulators might optimize or transform program execution, resulting in different intermediate states without causing a mistranslation error. We *select cutpoints [32] adaptively*, enabling targeted validation resilient to intermediate divergences.
(3) **Effectful programs:** Programs execute system calls, receive signals, and perform I/O, introducing non-determinism that causes their states to differ between executions. Our key idea is to *formally model these effects as actions* and enforce identical effects across executions through cross-ISA record-replay.
(4) **Tracing large programs:** Tracing every step of a large native program is time-consuming and reduces scalability. Our key idea is *speculative concolic tracing*, which decouples symbolic and concrete execution, allowing symbolic execution to run ahead along statically determined control flow while concrete execution advances only at branches and side effects.
(5) **Error reproduction:** Generating a small program to reproduce a mistranslation error requires preserving the mistranslated instructions along with the context in which the error appeared. Our key idea is to *leverage the traced states*, which contain sufficient information to recreate the error context and trigger the mistranslation.

We built Focaccia from the ground up to validate two state-of-the-art emulators, QEMU and Box64, on Linux across both x86-64 and Arm guest/host architectures. Focaccia validates emulator executions for small programs and large software applications with non-deterministic interactions.

We evaluate Focaccia on QEMU and Box64 across multiple ISAs, detecting and localizing all 17 known mistranslation errors across six bug categories, all of which escaped existing emulator test suites [11, 28]. Applying Focaccia to the open-source Arancini translator [64] further discovered eight previously unknown mistranslations, all confirmed and fixed upstream. For each error, Focaccia pinpoints the exact mistranslated instruction and the specific state values that differ, such as an incorrectly set carry flag. Focaccia validates small mistranslated programs in 8 seconds on average and scales to full applications including SQLite, Curl, and the Lua interpreter with non-deterministic I/O, tracing and validating a 99,000-instruction SQLite execution in approximately 5 minutes end-to-end. Generated reproducers reduce program size by up to 99%, isolating errors to single basic blocks.

Contributions. We make the following contributions:

(1) **Emulator bug study** – We present a study of emulator bugs and extensively analyze mistranslation bugs to identify distinct categories based on their root causes.
(2) **Formal model** – We formally define the state transition system that captures the semantics of native and emulated execution, which lays the formal foundation for our tracing and validation implementations.
(3) **The Focaccia system** – Our Focaccia system realizes our five key ideas to automatically, comprehensively, and deterministically detect mistranslation errors in state-of-the-art emulators across various architectures. Focaccia works reliably even for optimizing emulators and when emulating effectful programs.
(4) **Extensive evaluation** – We integrate Focaccia with QEMU and Box64, detecting, localizing and reproducing all tested mistranslation errors across six bug categories and three major software applications, and we discover eight previously unknown mistranslations in the Arancini hybrid binary translator [64].

2 Background and Motivation

2.1 CPU Emulators

CPU emulators virtualize an execution environment to run programs compiled for one architecture on another [51, 69]. This environment must faithfully run arbitrary programs on the host platform [69]. Dynamic Binary Translation (DBT) maps instructions from the guest ISA to the host ISA, which the host machine natively executes.

Complexity. Mapping between ISAs is intrinsically complex, since the semantics of individual ISAs are rarely well defined. Additionally, those mappings must be constructed from N input ISAs to M output ISAs. To accommodate this, emulators internally use a higher abstraction–an Intermediate Representation (IR). A typical emulator will first lift guest instructions to an IR, and then generate host instructions from it.

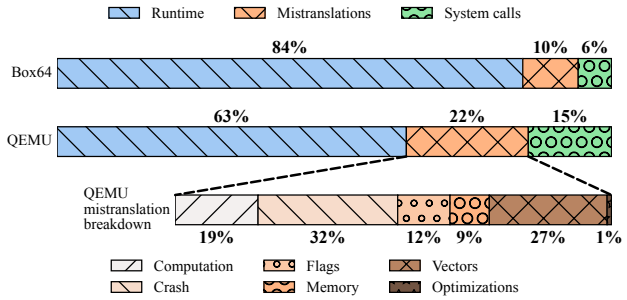


Figure 2. Analysis of user-mode errors in QEMU and Box64.

Optimizations. Modern emulators, such as QEMU, implement ad-hoc optimizations on their IR without specifying which semantics are preserved or at what granularity.

2.2 Analysis of Errors in CPU Emulators

We analyze QEMU [21] and Box64 [27] to answer two questions: *(i)* what types of errors exist in user-mode CPU emulators? *(ii)* are mistranslation errors a significant cause of bugs?

Methodology. We analyze bug reports from the Box64 GitHub tracker [7], the QEMU GitLab tracker [8], and mailing list [61] from April 2015 to May 2025, filtering for user-mode emulation and classifying into three error types. We use the gemma3:27b Large Language Model (LLM) [33] for filtering and categorization, but manually verify the results.

Dataset. During the analyzed period, QEMU and Box64 received 5,812 and 1,265 bug reports, respectively. For QEMU, we exclude system-mode bugs (the majority of reports); for both, we exclude non-execution reports (documentation, performance, build issues). The remaining user-mode bugs amount to 551 for QEMU and 1,093 for Box64. Neither consistently categorizes bug reports by root cause, requiring further analysis.

Error types. We divide the found user-mode bugs into three error types: *(i)* mistranslation errors, where guest instructions incorrectly translate to host instructions; *(ii)* system call support errors, from incorrect system call emulation or passthrough; and *(iii)* runtime errors, grouping all remaining execution failures that occur despite a correct translation.

Results. Fig. 2 shows the distribution of user-mode bug report types in QEMU and Box64. Mistranslation errors constitute a significant part of the reports, accounting for 22% of all user-mode reports in QEMU. For Box64, they account for *at least* 10% of all bugs, though most bug reports lack conclusive root-cause analysis. While mistranslations constitute a smaller fraction of all bug reports for QEMU, other reports predominantly concern system-mode device emulation, where the sheer number of emulated devices naturally generates high volume. In contrast, mistranslation errors affect the core instruction translation shared by all emulated programs, making them disproportionately impactful.

2.3 A Mistranslation Error Example

To illustrate the challenges in localizing and fixing mistranslation errors, we examine bug #1370.GL [74] from QEMU,

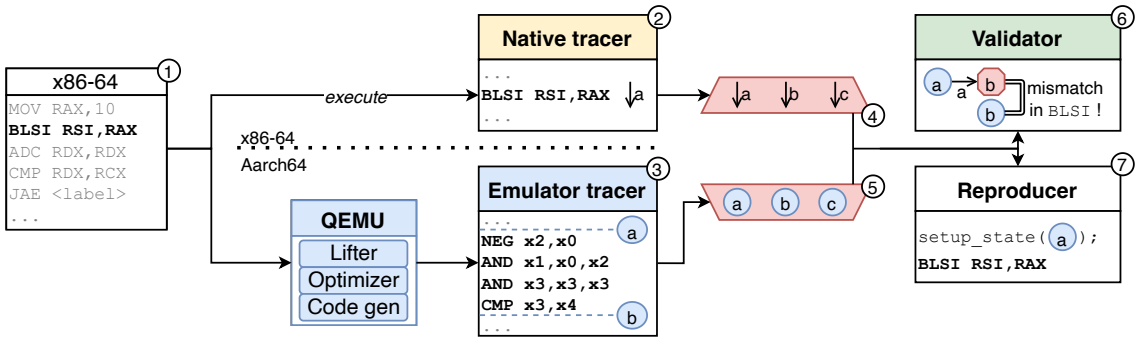


Figure 3. Example of QEMU's mistranslation of BLSI from x86-64 (guest) to Aarch64 (host), causing #1370.GL [74]

shown in Fig. 3. The mistranslation concerns the x86 BLSI instruction that extracts the lowest set bit. Crucially, it sets the x86 carry flag if-and-only-if the input is non-zero. QEMU incorrectly generates Arm code that inverts this flag.

Existing test suites failed to catch this subtle error, as the flag corruption only becomes observable when a subsequent branch depends on it, such as JAE in our example.

Furthermore, the initial fix proved insufficient, leading to another bug report for the same instruction (#2175.GL [40]). We will use this example throughout the paper to illustrate how Focaccia effectively addresses several challenges (§ 3.3).

2.4 State-of-the-Art

Testing. Existing approaches [11, 28] rely on user-written tests, limiting their ability to detect mistranslation errors for three reasons: *(i)* user-written tests exercise a fixed instruction subset, and errors outside it go undetected; *(ii)* detected mistranslation errors cannot be localized to the specific instructions that caused them; *(iii)* the expected behavior must be captured manually, which is difficult for subtle architectural behavior (*e.g.* flag values).

Differential testing [38] solves issue *(iii)*, by executing the same program on multiple emulators and comparing their outputs, identifying inconsistencies.

Fuzzing. Complementing prior testing methods, fuzzing [51] generates diverse random programs and inputs to trigger various execution paths, addressing issues *(i)* and, when combined with differential testing, *(iii)*. However, issue *(ii)* still remains. Fuzzing with crash or output oracles misses silent state corruption, and fuzzers that compare CPU state [51] observe it only within their own synthetic test case. Focaccia is complementary to fuzzers, validating real executions at the instruction level and catching silent state corruptions, such as an incorrectly set flag, that may never cause an observable failure in the fuzzed program but silently affect every program that executes the mistranslated instruction.

Symbolic execution. Symbolic execution has shown promise when applied to emulators for finding bugs [39, 50], comparing individual instruction implementations against a reference [63]. However, these approaches test instructions in isolation, not within larger applications where mistranslations can be catastrophic, nor can they handle effectful programs.

No current approach can detect bugs caused by emulator optimizations that only activate for specific instruction sequences (§ 7.2).

2.5 Problem Statement

CPU emulators are prone to mistranslation errors, and existing testing techniques often neither automatically detect nor reliably localize them. Moreover, they treat correctness ad hoc, without a formal semantic foundation. We therefore need an *automated* system, built on a *formal model defining correctness*, to detect and *localize* mistranslations in *unmodified* state-of-the-art emulators.

The resulting system should also satisfy three primary design criteria: It *(i)* works for any source and target ISA combination, *(ii)* across all stages of a complex optimizing emulator, and *(iii)* generates reproducible test cases for detected mistranslations, to guide developers in their bug fixing work.

3 Overview

We propose Focaccia, a validation system that comprehensively detects, localizes, and reproduces mistranslation errors in CPU emulators. In this section, we give a high-level overview of Focaccia's design and core approach, along with the key challenges we address to realize our design goals.

We design Focaccia to fulfill the five following design goals, which we believe are essential for a practical and effective validator for emulation correctness: *Generality* to support different pairs of guest/host ISAs and emulators; *Comprehensiveness* to detect behavioral mismatches even when deeply hidden in program executions; *Automation* to enable validation and diagnostics with minimal user guidance; *Scalability* to support large-scale programs; and *Reproducibility* to allow deterministic triggering of detected mistranslation bugs.

3.1 System Architecture

We build Focaccia around four components: a *native tracer*, an *emulator tracer*, a *translation validator*, and a *reproducer generator*. Figure 1 shows their position in the architecture.

Detection, localization, and reproduction all derive from one mechanism, the symbolic trace, which every component either produces or consumes.

Native tracer. The native tracer runs the guest program on its native platform and records execution traces capturing the correct program behavior. We use *concolic execution* [68], which combines symbolic and concrete execution, to record a platform-independent trace efficiently.

Emulator tracer. The emulator tracer runs the same guest program on the host architecture, inside the emulator under test. During emulation, it traces the *emulated guest states*, while controlling non-deterministic behaviour by replaying natively recorded side effects.

We detail the inner workings of both tracers in § 5.1 and § 5.2, respectively.

Translation validator. The translation validator compares the emulation to native execution, using our differential testing algorithm based on state transformations, presented in § 4.2. Upon detecting a mistranslation, the validator extracts diagnostics from the traces (*e.g.* the violating instructions) to help generate a reproducer. We detail the translation validator in § 5.3.

Reproducer generator. The reproducer generator uses extracted diagnostics to generate a small program that deterministically reproduces the detected mistranslation. The generated program replicates the last correct program state and then executes only the mistranslated instructions. We detail the reproducer generator in § 5.4.

3.2 Design Principle

We now present Focaccia's core approach to validate emulator translation correctness, and how all four system components interact to realize this approach in an automated workflow.

Programs as state transition systems. We view program execution as a sequence of states, each containing registers and memory, where each transition corresponds to executing one or multiple instructions.

By considering both native and emulated programs as transition systems, we can formally relate their executions. When their states are related *before* executing instructions, they should be similarly related *afterwards*. However, if the emulator mistranslates those instructions, their observed states will mismatch.

States need only be *related*—by some embedding [60]—instead of matching byte-for-byte, because the emulator may organize memory differently, or apply optimizations. Instead, we only require state transformations have the same structure between native and emulated execution.

We determine mistranslation through differential testing [53], treating the program's native execution as an oracle against its emulated execution.

3.3 System Workflow using the Example from § 2.3

We show how Focaccia finds the root cause of the example mistranslation error in QEMU, #1370.GL [74]. We have already described this bug in § 2.3, but now discuss how Focaccia detects and reproduces it.

Detecting the mistranslation with Focaccia. As depicted in Fig. 3, Focaccia first runs the program ① through the native tracer ② and the emulator tracer ③, collecting the native execution trace ④ and emulated guest states ⑤. The translation validator ⑥ then applies the native execution trace to the emulated states and checks for equivalence. When applying the `BLSI` transformation, the validator finds that the carry flag in the recorded emulated state differs from the calculated state, pinpointing the mistranslation to this transition.

Generating a reproducer. Using the translation validator's output, the reproducer generator creates a small program that deterministically reproduces the mistranslation. As shown in ⑦, the generated program sets up the state right before the mistranslation ⓐ and executes only the mistranslated `BLSI` instruction, together with Focaccia's diagnostics pointing to the carry flag forming a complete bug report.

3.4 Technical Challenges and Key Ideas

Realizing our design goals is non-trivial given the complexity of modern emulators and the comprehensive nature of our proposed validation system. We identify five core challenges; for each we present a key idea that we integrate into our approach.

Challenge #1: Platform-independent validation. Validating translation correctness requires comparing program states between native and emulated execution. Even when translation is correct, concrete values can differ between these executions. The emulator *embeds* guest memory within its host address space, so addresses and all values derived from them (pointer arithmetic, struct field accesses) differ. Environmental values such as auxiliary vector entries also differ legitimately but rarely affect correctness. Additionally, the emulator may not expose intermediate state at every point.

Key idea #1: Symbolic state abstraction. Representing states *symbolically* hides emulator-specific details while capturing their essential structure.

Focaccia represents state values as logical variables (*e.g.* α, β) and relationships between them as formulae (*e.g.* $\beta{=}\alpha{+}4$). Address-derived values like $\alpha{+}8$ remain correct regardless of the concrete base. Environmental values become distinct symbolic variables, so derived computations can match despite differing concrete values, yet any divergence they cause is traceable to the specific variable. Gaps in the emulator trace are bridged by the native symbolic state, enabling validation at cutpoints. During validation, Focaccia *instantiates* the symbolic state using the emulator's concrete values, producing an expected state that is directly compared against the actual emulator output (§ 5.1, § 5.3).

Challenge #2: Mismatching intermediate states. Emulators expose guest states at different granularities, so not every guest state has a corresponding host state, and vice versa.

Challenge #2a: Optimized programs. After optimizing an emulated program, it may go through different intermediate states than the original. Consider the program '`x=x+1; x=x+2;`' optimized to '`x=x+3;`': both terminate in the same final state, but the first has an additional intermediate state.

Challenge #2b: Coarse-grained visibility. Emulators may not grant access to the internal emulated state after any single instruction, meaning we can only observe state before and after executing *a sequence* of guest instructions.

Our key idea to address that challenge is:

> **Key idea #2: Adaptive cutpoint selection.** Exploring multiple cutpoints [32] allows identifying one where program behavior matches.

We thus allow program states to temporarily mismatch if-and-only-if they later match at the selected cutpoint [32], with all intermediate side-effects being the same. In practice, Focaccia buffers intermediate mismatches and checks subsequent states until one matches, accepting it as the cutpoint (§ 5.3).

Challenge #3: Effectful programs. Programs interact with the outside world through *effects* (*e.g.* querying the current time) that introduce non-determinism. Although multiple executions of the same program—either native or emulated—can observe different timestamps, those differences are *not* emulator bugs.

To reason about execution equivalence, we must eliminate effects as a source of state differences, for which our key idea is:

> **Key idea #3: Effects as actions.** Formally modeling effects as actions [54] enables reasoning about them, isolating them from pure computations, and thus controlling them.

We apply this key idea in our formal model (§ 4.1), recording effects during native execution and replaying them during emulation so they cannot cause mismatching states.

Challenge #4: Tracing large programs. Recording all states of a long-running execution is time consuming and unnecessary. We require a tracing strategy that minimizes overhead while preserving sufficient information to identify mismatching states, for which our key idea is:

> **Key idea #4: Targeted tracing.** Collecting limited traces at strategically selected points in the program execution significantly reduces its runtime footprint.

Focaccia realizes this key idea in two ways, together scaling validation to entire applications: *(i)* Speculative tracing records symbolic traces ahead of native execution for linear program regions (§ 5.1); *(ii)* Selective tracing limits emulated tracing only to user-defined program regions (§ 5.2), allowing Focaccia to efficiently find bugs in suspected program regions.

Challenge #5: Error reproduction. Reproducing an error first observed in a large program is difficult, yet errors must be easily reproducible; for which our key idea is:

> **Key idea #5: Transition context replication.** The symbolic state contains all information needed to localize and replicate the error context in a small program fragment.

Focaccia applies this key idea in its reproducer generator (§ 5.4), extracting the erroneous fragment and using the preceding symbolic state to replicate the error context in a minimal program.

4 Programs as State Transition Systems

To identify mistranslation, we formally define when a translation is correct, starting from program behavior.

4.1 Semantics

We model a program's execution as a Labeled Transition System with actions $\langle S,I,A,\rightarrow\rangle$ [54], where S is the set of program states; I is the set of operations, which capture pure computations; A is the set of actions, which represent effects; and '$\rightarrow$' is the transition relation. We represent any program state in S as a tuple $\langle R,M\rangle$, where R contains all ISA-specific variables (*i.e.* registers and status flags [18, 37]), while M represents main memory. Each transition we write as:

$$\langle\, \langle R, M\rangle, i \,\rangle \xrightarrow{a} \langle R', M'\rangle$$

Here, $i \in I$ is the executing operation, while $a \in A$ is an action that captures an effect.

Example: Simple. Consider a transition for the internal computation adding 42 to the `RAX` register on x86:

$$\langle\, \langle R, M\rangle, \text{`}\texttt{ADD RAX, 42}\text{'} \,\rangle \xrightarrow{\tau} \langle R', M\rangle$$

$$\text{where } R' \triangleq R \text{ with } R' = R[\texttt{RAX} \mapsto (R[\texttt{RAX}]+42) \bmod 2^{64}]$$

It sets the register `RAX` in R to the value after addition. $R[x \mapsto v]$ denotes setting *variable* x to *value* v. The instruction also updates the x86 status flags in R, which we omit for brevity. By convention, $\tau \in A$ is the internal action, indicating no external behavior.

Example: Effects. We capture non-determinism through effects, which are any interaction with the world outside the program state, *e.g.*, reading from file or accessing the current time. Such effects are not caused by mistranslations and must be modeled separately, corresponding to our key idea #3 (effects as actions). The following example illustrates that:

$$\langle\, \langle R,M\rangle, \text{`}\texttt{TIME}\text{'} \,\rangle \xrightarrow{\text{time}(v)} \langle\, R', M \,\rangle$$

$$\text{where } R' \triangleq R \text{ with } R'_t = R_t[\texttt{RAX} \mapsto v]$$

We assume for simplicity that `TIME` is a primitive operation. Crucially, its semantics depend on the *action* '$\text{time}(v)$' $\in A$, meaning executing the same operation with the same action always yields the same state, enabling formal reasoning about correctness in the presence of side effects.

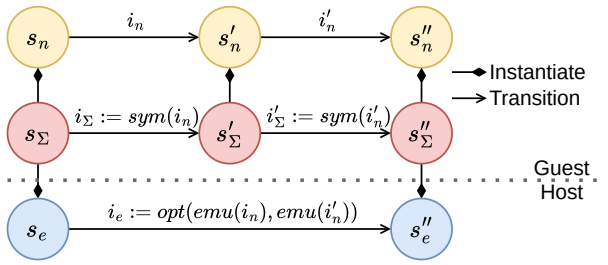


Figure 4. Analogous transitions between native, symbolic, and emulated states. The symbolic state s_Σ generalizes over the native state s_n and emulated state s_e.

4.2 Correctness

Our correctness criterion, expanded from Popek and Goldberg [60], must relate programs in two languages, stating when a program *emulates* another. The two languages share the same semantic model but instantiate it differently, with the state representation S, instructions I, and transition relation '$\rightarrow$' differing between architectures, while the set of side effects A is shared. When considering x86 and Arm, we thus have $\langle S_{x86}, I_{x86}, A, \rightarrow_{x86}\rangle$ and $\langle S_{Arm}, I_{Arm}, A, \rightarrow_{Arm}\rangle$.

We must relate the *states* between the native and emulated program. We map any native state $s_n \in S_n$ to an emulated state $s_e \in S_e$ with function '$V : S_n \rightarrow S_e$'. For instance, $V_{x86\rightarrow Arm}$ maps `RAX` to `R0`, `RBX` to `R1`, etc. However, note that the state embedding V could be more complex, for instance, by mapping guest states to a data structure internal to an existing emulator. Def. 1 states the emulation correctness generally.

> ▶ **Definition 1** (Emulation). Given operation $P_n \in I_n$ in language $\langle S_n, I_n, A, \rightarrow_n\rangle$, operation $P_e \in I_e$ in language $\langle S_e, I_e, A, \rightarrow_e\rangle$, state mapping '$V : S_n \rightarrow S_e$', and start states $\langle R_n, M_n\rangle$ and $\langle R_e, M_e\rangle$ satisfying $V(\langle R_n, M_n\rangle) \equiv \langle R_e, M_e\rangle$ and:
>
> $$\langle \langle R_n, M_n\rangle, P_n \rangle \xrightarrow{a}_n \langle R'_n, M'_n\rangle$$
>
> $$\langle \langle R_e, M_e\rangle, P_e \rangle \xrightarrow{a}_e \langle R'_e, M'_e\rangle$$
>
> Then operation P_e *emulates* P_n (w.r.t. mapping V and start state $\langle R_n, M_n\rangle$) iff $V(\langle R'_n, M'_n\rangle) \equiv \langle R'_e, M'_e\rangle$.

Example: Emulation. To illustrate Def. 1, consider again the x86 program '`ADD RAX, 42`'. When $V_{x86\rightarrow Arm}$ maps `RAX` to `R0`, we can emulate the instruction on Arm with '`ADD R0, R0, 42`'. Given a concrete initial state $\langle R_{x86}, M\rangle$ where $R_{x86}[\texttt{RAX}] = 3$, which $V_{x86\rightarrow Arm}$ maps to $\langle R_{Arm}, M\rangle$ with $R_{Arm}[\texttt{R0}] = 3$, where $V_{x86\rightarrow Arm}(\langle R_{x86}, M\rangle) \equiv \langle R_{Arm}, M\rangle$ beforehand. Then that instruction's *emulation is correct* because:

$$\langle \langle R_{x86}, M\rangle, \text{'}\texttt{ADD RAX, 42}\text{'} \rangle \xrightarrow{\tau}_{x86} \langle R'_{x86}, M\rangle$$
where $R_{x86}[\texttt{RAX}] = 3$ and $R'_{x86}[\texttt{RAX}] = 45$, and

$$\langle \langle R_{Arm}, M\rangle, \text{'}\texttt{ADD R0, R0, 42}\text{'} \rangle \xrightarrow{\tau}_{Arm} \langle R'_{Arm}, M\rangle$$
where $R_{Arm}[\texttt{R0}] = 3$ and $R'_{Arm}[\texttt{R0}] = 45$,

which satisfies $V_{x86\rightarrow Arm}(\langle R'_{x86}, M\rangle) \equiv \langle R'_{Arm}, M\rangle$.

5 Design

Focaccia is built around four components, shown in Fig. 5: a native tracer, emulator tracer, translation validator, and a reproducer generator; which we explain below.

5.1 Native Tracer

Our native tracer executes the guest program P_{guest} on the guest-architecture, recording the executed instructions I_{guest} and actions for comparison against emulated execution (Def. 1). Focaccia obtains these traces with concolic execution [68], combining *concrete* execution with *symbolic* [43] representation of program semantics.

Symbolic tracing. Focaccia interprets parts of the program into logic formulae through symbolic execution [43]. By representing values symbolically—with a variable in a logic formula—the resulting symbolic state s_Σ represents a set of concrete states. Encoding states symbolically embodies our key idea #1 (symbolic state abstraction), as it encodes states and their transformations independent from specific platform values, such as memory addresses. We can instantiate s_Σ to both the native state s_n and emulated state s_e, as depicted in Fig. 4.

Concolic tracing. Symbolic execution alone is often impractical for two reasons: *(i)* large-scale analysis may suffer from state explosion [68]; and *(ii)* comparing side effects requires concrete data such as memory contents. Focaccia therefore employs concolic execution [68], restricting symbolic execution to a single concrete path.

We realize our key idea #3 (effects as actions) by capturing interactions with the outside world as a sequence of *actions* (from Def. 1). For instance, a `read` system call generates an action capturing the updates to register `RAX` and memory. Together, the symbolic trace of executed instructions and the log of actions fully capture a program's execution.

Speculative symbolic execution. Focaccia realizes key idea #4 (targeted tracing) by decoupling symbolic and concrete execution. Starting from the current program counter, Focaccia symbolically executes instructions until reaching a conditional branch or side effect whose outcome cannot be statically determined. Concrete execution then advances to that point via a hardware breakpoint, resolving the outcome, after which symbolic execution resumes. This avoids single-stepping every instruction through the debugger interface, significantly reducing the concrete execution cost of native tracing (§ 7.4). For linear code regions, which constitute the majority of executed instructions, symbolic execution proceeds at disassembly speed with no concrete execution cost.

5.2 Emulator Tracer

In contrast to native tracing, where Focaccia executes program P_{guest} directly on the guest-machine, we also trace the program P_{host}, which emulates P_{guest} *inside* the emulator; recall that the ultimate objective is to determine whether P_{host} correctly emulates P_{guest} (Def. 1). In either case, we obtain the

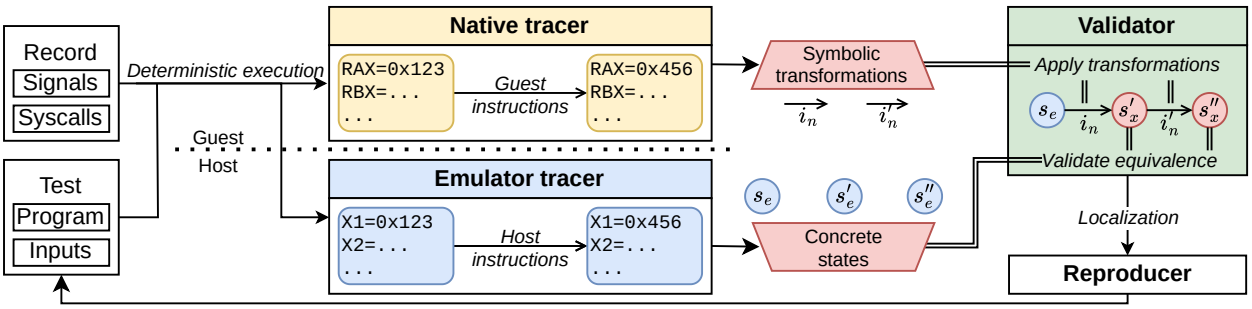


Figure 5. The components in Focaccia: native tracer (§ 5.1), emulator tracer (§ 5.2), validator (§ 5.3), and reproducer generator (§ 5.4).

executed sequence of operations and actions, but the mechanisms to obtain these differ.

Emulator states. The emulator embeds the native state $s_n \in \mathrm{S}_n$ within the emulated state $s_e \in \mathrm{S}_e$; that embedding we formalized with '$V : \mathrm{S}_n \to \mathrm{S}_e$' (§ 4.2). However, in practice, we want to extract the native state s_n from the emulated state s_e. We thus consider '$V^{-1} : \mathrm{S}_e \to \mathrm{S}_n$', which is defined for any $s_e \in \mathrm{range}(V)$. In practice, we assume the emulator preserves that membership; regardless, Focaccia extracts the native state s_n with an emulator-specific layer.

Tracing granularity. Emulators vary in the frequency at which the emulated state s_e corresponds to the native state s_n. Emulator internals, such as optimizations, affect the granularity across which we want states to match, corresponding to weak bisimulation [54]. For example, Fig. 4 shows a weak bisimulation after an optimization *opt()*. The initial and final states match even when the intermediate state matches no target state. To accommodate this, each emulator backend can either select cutpoints statically—for instance, per-instruction or per-basic-block—or rely on dynamic cutpoint selection in the validator (§ 5.3), corresponding to our key idea #2 (adaptive cutpoint selection).

Deterministic non-determinism. Non-determinism arises from four primary sources: *(i)* system calls with non-deterministic results, *(ii)* I/O with non-deterministic data sources and other processes, *(iii)* signal delivery, and *(iv)* multiple thread interleavings [2, 57, 58]. These can cause native and emulation executions to diverge even when the underlying translation is correct.

Def. 1 only holds when both executions observe identical non-determinism. Focaccia records the first three sources as actions (§ 4) and uses those to deterministically replay them during emulation, corresponding to key idea #3 (effects as actions). We do not currently support concurrency, as it requires modifying an emulator to control its thread scheduler.

Selective tracing. When misbehavior is observed during emulation, validating the entire execution is unnecessary. Focaccia allows the user to specify an approximate program region of interest, restricting validation to that region. The user need not know the precise location of the bug; Focaccia validates all instructions within the region and reports any mistranslation found, even if its root cause lies before the specified start point. This is another realization of our key idea #4 (targeted tracing).

5.3 Translation Validator

Focaccia validates that emulated execution (§ 5.2) correctly emulates (Def. 1) the native execution (§ 5.1). In practice, Focaccia checks that the symbolic state s_Σ, computed analogous to s_n during native execution, instantiates to the emulated state s_e, as illustrated in Fig. 4.

Emulator validation. When in guest state s_n^0, natively executing guest operation $i_n \in \mathsf{I}_n$ with arbitrary action a, we produce another state s_n^1. Simultaneously, we execute that same guest operation i_n *symbolically* on the corresponding symbolic state s_Σ^0. Finally, the emulator executes its analogous host operation $i_e \in \mathsf{I}_e$ on its host state s_e^0, producing:

$$\langle\, s_n^0, i_n \,\rangle \xrightarrow{a}_n s_n^1 \qquad \langle\, s_\Sigma^0, i_n \,\rangle \xrightarrow{a}_\Sigma s_\Sigma^1 \qquad \langle\, s_e^0, i_e \,\rangle \xrightarrow{a}_e s_e^1$$

By Def. 1, we must check whether s_n^1 corresponds to s_e^1 under some mapping V. However, V is not known explicitly, since address layouts vary between executions and no fixed mapping from native to emulated states exists. Instead, we instantiate the symbolic state s_Σ^1, co-created with s_n^1 by the same operations, by substituting the emulator's concrete values for the symbolic variables, producing an expected emulated state $s_e^{1\prime}$ that we check against s_e^1. A mismatch indicates i_e does not correctly emulate i_n (Def. 1), another application of key idea #1 (symbolic state abstraction).

Mismatching granularities. Although a mismatch between $s_e^{1\prime}$ and s_e^1 indicates i_e *does not* emulate i_n (*i.e.* it violates Def. 1), that does not immediately indicate a translation error. Such mismatches arise when the emulator exposes s_e at a finer granularity than it maintains $V(s_n) \equiv s_e$—for instance, reporting s_e per instruction while only preserving correspondence at basic block boundaries. In such cases, Focaccia buffers the mismatch and continues validating. If a later state s_e^k matches $s_e^{k\prime}$, all buffered mismatches are resolved and s^k is accepted as a cutpoint [32]. Otherwise, the earliest unresolved mismatch is reported as a translation error. This realizes our key idea #2 (adaptive cutpoint selection). Conversely, when Focaccia

reports no mismatch, Def. 1 holds at every cutpoint and both executions produced identical emulator-visible effects.

Alignment cost. State alignment costs time and memory linear in the executed instructions. Focaccia aligns states by guest program counter in a single forward pass, merging consecutive symbolic transforms until their end address matches the next emulator-visible state, then performs one equivalence check there. Cutpoint selection thus does not depend on basic block boundaries, as Focaccia finds cutpoints adaptively wherever native and emulated states match, and a coarser-grained emulator only yields coarser cutpoints. Recurring program counters, as in loops, are paired in execution order, so every iteration is checked independently.

Cross-validation. The instruction semantics used to generate s^1_Σ may themselves be incorrect. To validate them, we instantiate s^1_Σ to a native state $s^{1\prime}_n$ instead and check it against the hardware-observed s^1_n. A mismatch indicates erroneous semantics, as illustrated by Fig. 4's dual instantiation.

Aliasing. Coincidentally equal concrete values can mask a mistranslation on a particular input, as with any dynamic validation technique. Focaccia therefore validates only the observed execution, not other inputs or untaken paths. Checking every emulator-visible state limits masking when an incorrect value remains live and later influences observable computation, but an error whose effect is overwritten or remains observationally equivalent throughout the trace escapes detection.

State mismatches. When a mismatch occurs, Focaccia records detailed diagnostics in an error log. This log includes *(i)* the state values that differ, *(ii)* the symbolic equations they violate, *(iii)* the prior guest state, and *(iv)* the mistranslated guest instructions. This information supports debugging and enables Focaccia to generate reproducers. In the event of a crash, the triggering state and instructions are similarly captured for reproducer generation.

5.4 Reproducer Generator

Our *reproducer generator* produces small programs for which an error identified in a larger program reappears.

Localizing bugs. To generate those small programs, Focaccia first localizes the source of the bug within the program. We identify program fragments reaching mismatching states, and since we know exactly which state variables (*e.g.* registers, flags, or memory) mismatch, we can accurately pinpoint which instructions were mistranslated.

Context reproduction. We realize our key idea #5 (transition context replication) by using the symbolic state recorded before the erroneous fragment to generate setup instructions that recreate the exact pre-error state. For instance, when the state requires `RAX=3`, we emit a '`MOV RAX, 3`' instruction. Together with the mistranslated fragment, this forms a small program that faithfully reproduces the original bug.

Reproducer growth. Reproducers are significantly smaller than the original program because *(i)* only the basic block containing the mistranslation need be included, and *(ii)* setup instructions need only satisfy the symbolic constraints relevant to that block, not the full execution history, keeping reproducers small regardless of the original program's size.

6 Implementation

We implement Focaccia targeting the QEMU and Box64 emulators on two distinct Linux environments, x86-64 and Aarch64. As part of our implementation, we use the Miasm symbolic executor and RR program record-replay system [57].

Concolic executor. At its core, Focaccia uses Miasm to produce symbolic equations for the x86-64 and Aarch64 instructions. However, Focaccia also extends Miasm with 15 additional x86-64 instructions, primarily for the Intel BMI and AVX instruction set extensions [37], as well as Aarch64 atomics.

QEMU interface. The Focaccia backend for QEMU implements its target by exposing read-only program state through QEMU's GDB remote interface. We use this interface to fetch guest state and drive execution, as well as the base for our deterministic replayer logic.

Box64 interface. Unlike QEMU, Box64 does not support a debugger interface that permits accessing arbitrary guest state [12]. Instead, Focaccia partially validates Box64 by parsing its debug log, which includes register values per basic block. This results in partial validation, sufficient for catching some mistranslation errors.

Deterministic replay. We implement our replay facilities atop the RR record of non-deterministic execution [57]. In particular, Focaccia's native tracer uses the GDB stub exposed by the RR replayer, while our emulator validator parses the RR log to extract deterministic values for non-deterministic events and directly replays them atop QEMU's GDB interface.

7 Evaluation

We evaluate Focaccia around seven dimensions: generality (§ 7.1), validation effectiveness (§ 7.2), discovery of unknown bugs (§ 7.3), tracing efficiency (§ 7.4), support for non-deterministic execution (§ 7.5), support for application-level deployment (§ 7.6) and the effectiveness of generated reproducers (§ 7.7).

Experimental testbed. We evaluate Focaccia on an Intel Xeon Platinum 8562Y+ and an Ampere-1a processor for x86-64 and Aarch64 respectively. All input programs are compiled using GCC 14.2.1 and linked statically against Musl 1.2.5 [4].

Benchmarks. We validate QEMU and Box64 using programs known to trigger mistranslations from each error category (see § 2.2). We also inject these same mistranslations in three large applications, the Lua interpreter, Curl and SQLite.

7.1 Generality

RQ1. *Does Focaccia support validating different emulators across multiple guest/host architectures?*

Focaccia supports all architectures that can be deterministically replayed using RR, namely x86-64 and Aarch64, as

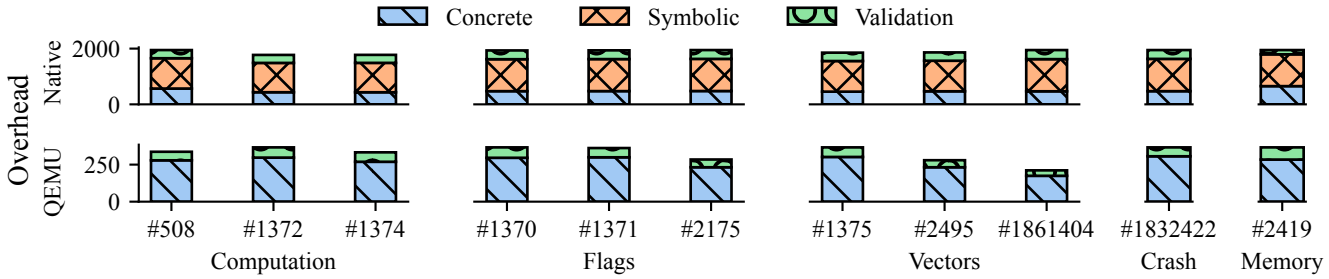


Figure 6. End-to-end overhead breakdown for native tracing and translation validation for QEMU (× native execution time).

Guest	Host	Emulator	Deterministic
x86-64	x86-64	QEMU	Only from Intel Nehalem [56]
x86-64	Aarch64	QEMU	–"–
Aarch64	Aarch64	QEMU	Only on Arm Neoverse [56]
Aarch64	x86-64	QEMU	–"–
x86-64	Aarch64	Box64	Unsupported

Table 1. Focaccia architecture and emulator coverage.

shown in Tbl. 1. Although QEMU supports additional ISAs, we exclude them because Focaccia inherits RR's limitation and requires Miasm to support the guest ISA. For Box64, Focaccia supports x86-64→Aarch64; deterministic replay is unavailable due to its limited debugger interface.

Focaccia validates single-threaded executions and thereby detects mistranslations whose manifestation does not depend on thread interleaving, including every bug in our study. Concurrency-specific mistranslations are out of scope because detecting incorrect memory ordering mappings requires enforcing recorded interleavings in the emulator's scheduler, which is not broadly supported by emulator debuggers.

7.2 Effectiveness

RQ2. *Can Focaccia detect and localize the cause of known mistranslation errors in emulators?*

We evaluate Focaccia's effectiveness at detecting mistranslations using a set of known bugs from each error category (see § 2.2), summarized in Tbl. 2. We selected the bugs by predetermined criteria, spanning all six root-cause categories, both emulators, and both ISAs, favoring breadth over count. Since reproduction requires a triggering program and a build of the buggy emulator version, we limited candidates to well-documented reports with trigger code. Crucially, each stems from a mistranslation that escaped the extensive test suites used by these emulators and was discovered in the wild before being fixed.

Mistranslated programs. Each bug is exhibited in a small C program that reliably triggers it under emulation. Each program was either sourced directly from a bug report or crafted based on the bug description. The programs span 3 to 32 lines of C and execute roughly 950 to 36,000 dynamic user-mode instructions, as each is validated end-to-end including libc initialization without any region specification.

Optimization bug. Of particular interest is bug #2248.GL [59], caused by a QEMU optimization that only fires when a specific sequence of instructions appears together in a basic block. Per-instruction tools (§ 2.4) cannot trigger this bug because the optimization requires the surrounding instruction context to activate. Fuzzers face a similar challenge, as they must both generate the precise sequence that triggers the optimization and detect the resulting silent state corruption. Focaccia detects it because it validates entire instruction sequences and checks state equivalence at every step.

Focaccia can detect all known mistranslation errors shown in Tbl. 2, across multiple mistranslation error categories, architectures and emulators. The bugs concentrate on computation and flag errors for x86-64 guests, mirroring the category distribution of our study (§ 2.2). Additionally, Focaccia precisely localizes each mistranslation to the instruction that is the root cause, the specific state values that are incorrect and each violated symbolic equation.

Comparison to prior tools. Prior tools cannot detect most of these bugs. EmuFuzzer [51], PokeEMU [50], and Examiner [39] test instructions in isolation and none of them executes x86-64 guest code, while Fleece [38] and Mishegos [6] test only instruction decoding. None of these tools validates an effectful full-program execution, localizes a divergence observed in that execution, or derives a reproducer from it.

False positives. False positives occur but are managed. Cut-points resolve granularity mismatches, and cross-validation against hardware catches incorrect reference semantics, exposing real Miasm bugs. Remaining mismatches stem from incomplete state visibility, such as Box64's partial log, and Focaccia grades them as unconfirmed. Reports carry the violated equation and states for triage.

7.3 Discovering Unknown Bugs

RQ3. *Can Focaccia discover previously unknown mistranslation errors?*

We apply Focaccia to Arancini [64], an open-source hybrid binary translator from x86-64 to Aarch64, through the

Category	Bug ID	Guest	Host	Emulator	Instruction	Description
Computation	#364.GL [47]	Aarch64	any	QEMU-5.2	LDSMAX	Incorrect result
	#508.GL [45]	x86-64	any	QEMU-6.1	CMPXCHG	Incorrect zero extension
	#508.Box [10]	x86-64	any	Box64-0.3.8	CMPXCHG	Incorrect zero extension
	#1372.GL [76]	x86-64	any	QEMU-7.2	BEXTR	Incorrect result
	#1374.GL [77]	x86-64	any	QEMU-7.2	BZHI	Incorrect result and sign flag
	#1828867.LP [30]	x86-64	any	QEMU-4.0	LAHF	Incorrect result
Flag	#1370.GL [74]	x86-64	any	QEMU-7.2	BLSI/BLSR	Incorrect carry flag
	#1371.GL [75]	x86-64	any	QEMU-7.2	BLSMSK	Incorrect carry flag
	#2175.GL [40]	x86-64	any	QEMU-8.2	BLSI	Incorrect carry flag
Vector	#1375.GL [78]	x86-64	any	QEMU-7.2	ADDSUBPS	Incorrect NaN handling
	#2495.GL [41]	x86-64	any	QEMU-9.0	MOVQ	Incorrect result
	#1861404.LP [44]	x86-64	any	QEMU-4.2	VMOVDQU	Incorrect result
Crash	#1376.GL [79]	x86-64	any	QEMU-7.2	LSL	Incorrectly raised SIGSEGV
	#1377.GL [80]	x86-64	any	QEMU-8.0	CVTPS2PD	Incorrectly raised SIGSEGV
	#1832422.LP [31]	x86-64	any	QEMU-4.0	CMPPD	Incorrectly raised SIGILL
Opt.	#2248.GL [59]	Aarch64	any	QEMU-8.2.1	LSR	Incorrect result
Mem.	#2419.GL [66]	Aarch64	any	QEMU-8.1.3	LDAPUR	Incorrect result

Table 2. Known QEMU and Box64 mistranslation errors validated with Focaccia.

same log-based interface as Box64. We validate guest programs under Arancini without prior knowledge of any bug and inspect each reported mismatch.

Results. Focaccia discovered eight previously unknown mistranslations in Arancini's dynamic Aarch64 backend, localizing each to the mistranslated instruction. Six fall in categories from our study (§ 2.2), three computation, two flag, and one memory bug, while the remaining two stem from incorrect register allocation, a class that our study did not surface. All eight were confirmed and fixed upstream.

Takeaway. Focaccia's detection extends beyond reproducing known bugs, as it uncovers new mistranslations even in emulators that expose only a log-based interface.

7.4 Validation Overhead

RQ4. *What is the overhead of validating translation correctness end-to-end with Focaccia?*

We evaluate Focaccia's overhead on a subset of the mistranslated programs, measuring runtime relative to native and emulated execution. Because these programs use deterministic inputs, we omit deterministic replay. Fig. 6 summarizes the overheads for native tracing and QEMU translation validation.

Native tracing. Symbolic execution dominates the tracing cost, as it must generate symbolic equations for each executed instruction. Concrete execution contributes less, and can be effectively further reduced through speculative tracing.

Emulator tracing. Most of the cost comes from concrete execution, with validation contributing only a small portion. Single-stepping guest instructions via the debugger interface dominates runtime and cannot be optimized. Despite these per-component overheads, end-to-end validation remains practical, averaging just 8 seconds including native tracing.

7.5 Non-deterministic Execution

RQ5. *Can Focaccia validate emulated effectful programs?*

Application	Instructions	Injection Point	Bug ID	Replayed
SQLite	99,348	`local_getline`	#508.GL	File-I/O
Curl	417,848	`Curl_sendrecv`	#2175.GL	Socket-I/O
Lua	401,983	`laction`	#2495.GL	Signals

Table 3. Applications with injected bugs, injection points, and replayed OS interfaces.

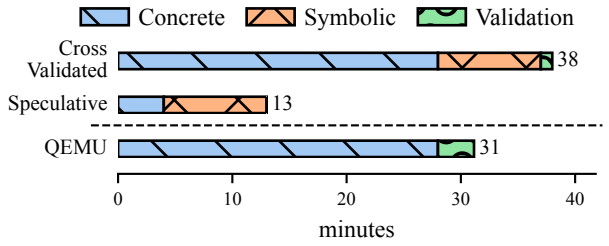


Figure 7. End-to-end runtime comparison for cross-validated, speculative native tracing and QEMU validation for Curl.

To assess Focaccia's capability to handle effectful programs, we inject known mistranslation bugs into three larger applications that require deterministic replay to validate (Tbl. 3).

Injection points. We chose the three applications to cover distinct non-determinism classes and place injection points only along control paths reached under nondeterministic conditions. SQLite exercises file I/O (`open`, `read`, `close`). Curl exercises socket I/O (`socket`, `connect`, `recvfrom`). Lua exercises signal handling (`rt_sigaction`, `rt_sigreturn`). Tbl. 3 reports dynamic instruction counts from `main` to each point, ranging from 99,348 to 417,848 instructions, which places each injection point deep in the application's execution.

Deterministic replay. Focaccia deterministically replays system calls and signals to reach each injection point, substituting recorded return codes, replaying memory writes from the RR log, and entering and exiting signal handlers while

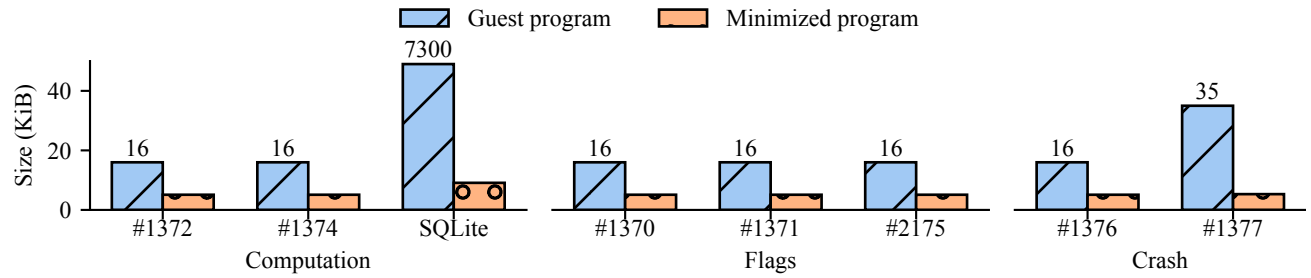


Figure 8. Code size comparison of the guest mistranslated program and the minimized reproducer generated by Focaccia.

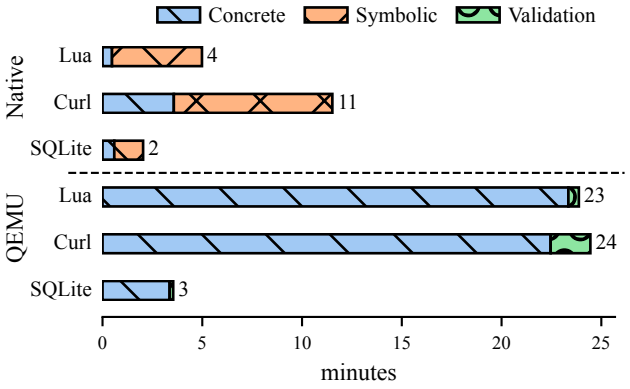


Figure 9. Runtime breakdown for Lua, Curl and SQLite for speculative and selective tracing, and for QEMU validation.

restoring the recorded register state. Using this replay, Focaccia detects and localizes the injected bug in each application.

7.6 Application-level Deployment

RQ6. *Can Focaccia validate entire software applications?*

We answer this question by validating Curl end-to-end as a representative large program, and then assessing selective tracing across three x86-64 applications (Lua, Curl, and SQLite) emulated on Arm.

Full-application validation. Focaccia supports end-to-end validation of large applications using either cross-validated native tracing, which validates symbolic equations against native execution, or speculative native tracing, which reduces concrete execution costs.

End-to-end runtime. Fig. 7 shows the end-to-end runtime of cross-validated, speculative native tracing, and emulator validation, for a Curl `GET` request for a 5 KiB file. Speculative tracing obtains significantly reduced time by eliminating the cross-validation cost and reducing concrete execution time.

Selective tracing. Full-execution validation with no user input is the default, and selective tracing optionally reduces validation time. Application-level validation benefits from starting at application code and stopping once the suspected bug is reached, skipping initialization code assumed to be correct. In particular, Focaccia treats the `_start` → `main()` path as correct a priori, though it still replays all non-deterministic inputs from `_start` to prevent divergence. If the selected region excludes the root cause, widening it and rerunning restores exact localization. To assess scalability, we measure its runtime from application start to the injection point using the benchmarks in Tbl. 3.

Fig. 9 shows the runtime of speculative and selective native tracing and of emulator validation from `main()` to bug detection. Selective tracing significantly reduces overhead, making Focaccia practical even when mistranslation errors occur deep in a program's execution.

Deployment. Focaccia primarily targets continuous integration and triage workflows. In continuous integration, emulator developers run test suites and representative workloads under validation after changes, analogous to compiler translation validation. In triage, a user who observes misbehavior under emulation applies selective tracing to localize the fault and generate a reproducer. Neither deployment approach requires knowing bug locations in advance.

7.7 Reproducer Effectiveness

RQ7. *Are the reproducers generated by Focaccia effective for reproducing mistranslation errors with a smaller code size?*

We assess the effectiveness of the minimized programs by comparing their code size to that of the original mistranslated programs in Tbl. 2 and in SQLite with the injected bug.

Effectiveness. Fig. 8 shows that Focaccia produces substantially smaller programs that still reproduce the mistranslation. For small programs, savings come from eliminating libc initialization; for SQLite, nearly all application code before the mistranslated instruction is removed, yielding an input-free, fully deterministic reproducer.

Scalability. As discussed in § 5.4, minimization targets only the instructions leading to the mistranslation, so reproducer size grows marginally even for large guests such as SQLite. The resulting reproducers are compact, deterministic artifacts suitable for inclusion in bug reports.

8 Related Work

Symbolic execution. Systematic testing via symbolic and concolic execution, along with related optimizations has been explored extensively [24, 25, 48, 62, 68, 70, 82]. Other work uses symbolic execution for bug detection and reproduction [24], as well as necessary optimizations to minimize symbolically traced paths and thus overhead [25, 70, 82]. Another

line of work combines concrete tracing with symbolic exploration to systematically explore program paths [68]. Focaccia builds on both lines of work to build a concolic executor that uses optimized tracing for a single program path, resulting in program semantics represented by symbolic equations.

Emulator testing. Prior work in emulator testing spans component-level testing, fuzzing-based differential testing, and differential testing based on symbolic execution. Component-level testing includes Fleece [38], Mishegos [6], and Kim et al. [42]. Fuzzing-based differential testing includes EmuFuzzer [51] and Box64 [11, 13]. Symbolic-execution-based differential testing includes PokeEMU [50], Examiner [39], and Arm ASL [63]. We discuss their mechanisms, limitations and relation to Focaccia in § 2.4.

Record-replay. Record-replay systems have been designed for recording and replaying execution of user-space programs [2, 5, 14, 22, 57, 58, 67, 84]. Among these, only a subset support recording and replaying multi-threaded programs [22, 29, 52, 57, 58, 71] without hardware modification. Focaccia uniquely combines symbolic execution with native replay, while also being unique in transferring a native record log to an emulator and replaying on a different ISA.

Failure reproduction. Failure reconstruction [26, 72, 73, 81, 84] and deterministic simulation testing [1, 9, 83] aim to reproduce failures with reduced tracing or controlled execution. Failure reconstruction only exposes observable errors, while deterministic simulation testing requires heavy simulator engineering. Neither is suitable for the subtle mistranslation errors Focaccia targets.

9 Conclusion

In this paper, we present Focaccia, a system to detect, localize, and reproduce mistranslation errors in user-mode CPU emulators, built on a formal definition of translation correctness. We also present a systematic study of real CPU-emulator bugs, categorizing their root causes and identifying subcategories for mistranslation errors to show their prevalence.

Artifact Availability

Focaccia is publicly available together with the full experimental setup and bug database.

The artifact appendix describes the experimental setup and reproduction workflow.

Acknowledgments

This work was supported in part by an ERC Starting Grant (ID: 101077577) and the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY), the Intel Trustworthy Data Center of the Future (TDCoF), and Google Research Grants.

References

[1] [n. d.]. Deterministic Simulation Testing (DST) - Debugging with perfect reproducibility. https://antithesis.com/resources/deterministic_simulation_testing/.

[2] 1999. MEC. https://lwn.net/1999/0121/a/mec.html. Accessed on: 2025-08-12.

[3] 2004-2025. QEMU Launchpad bug tracker. https://launchpad.net/qemu. Accessed: August 07, 2025.

[4] 2005-2021. musl libc. https://musl.libc.org/. Accessed: April 17, 2023.

[5] 2008. Reverse Debugging with the TotalView Debugger. https://cug.org/5-publications/proceedings_attendee_lists/2008CD/S08_Proceedings/pages/Authors/16-19Thursday/Gottbrath-Thursday18B/Gottbrath-Thursday18B-paper/index.html. Accessed on: 2025-08-12.

[6] 2019. Destroying x86_64 instruction decoders with differential fuzzing. https://blog.trailofbits.com/2019/10/31/destroying-x86_64-instruction-decoders-with-differential-fuzzing/. Accessed on: 2025-08-11.

[7] 2020-2025. Box64 GitHub Issue tracker. https://github.com/ptitSeb/box64/issues. Accessed: August 07, 2025.

[8] 2020-2025. QEMU GitLab Issue tracker. https://gitlab.com/qemu-project/qemu. Accessed: August 07, 2025.

[9] 2023. Deterministic Simulation: A New Era of Distributed System Testing (Part 1 of 2). https://risingwave.com/blog/deterministic-simulation-a-new-era-of-distributed-system-testing/.

[10] 2025. Box64 Commit log: [DYNAREC][INTERP] Fixed an edge case of CMPXHG. https://github.com/ptitSeb/box64/commit/70ab16a6d6c54c83a6f9f678c5bbe1c104c6a672. Accessed on: 2025-12-05.

[11] 2025. Box64 dynamic recompilation test documentation. https://github.com/ptitSeb/box64/blob/df2e4f941f616a775388004138e605518d9cdb8d/docs/USAGE.md?plain=1#L610. Accessed on: 2025-08-11.

[12] 2025. Box64 dynamic recompilation test issue. https://github.com/ptitSeb/box64/blob/cb26e7f3fe9354c016a3c46f14dafef49b7ed3dd/docs/USAGE.md?plain=1#L558. Accessed on: 2025-08-11.

[13] 2025. Box64 dynamic recompilation test issue. https://github.com/ptitSeb/box64/issues/634. Accessed on: 2025-08-11.

[14] 2025. UndoDB: Time Travel Debugging for C/C++ and Java. https://undo.io/. Accessed: 2025-08-12.

[15] Keith Adams and Ole Agesen. 2006. A Comparison of Software and Hardware Techniques for X86 Virtualization. *SIGOPS Oper. Syst. Rev.* 40, 5 (oct 2006), 2–13. doi:10.1145/1168917.1168860

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

[17] Apple-Silicon. 2020. Addressing Architectural Differences in Your macOS Code. https://developer.apple.com/documentation/apple-silicon/addressing-architectural-differences-in-your-macos-code.

[18] Arm Ltd. 2007. *Arm Architecture Reference Manual for A-profile architecture.* Arm Ltd. https://developer.arm.com/documentation/ddi0487/latest/

[19] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (Jan. 2019), 31 pages. doi:10.1145/3290384

[20] Amazon AWS. [n. d.]. AWS Graviton Processor. https://aws.amazon.com/ec2/graviton.

[21] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41. https://www.usenix.org/legacy/publications/library/proceedings/usenix05/tech/freenix/bellard.html

[22] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. iDNA: Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE '06)*. Association for Computing Machinery, New York, NY, USA, 154–163. doi:10.1145/1134760.1220164

[23] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph. D. Dissertation. Massachusetts Institute of Technology, USA. AAI0807735. https://hdl.handle.net/1721.1/30160

[24] Ning Chen and Sunghun Kim. 2015. STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution. *IEEE Transactions on Software Engineering* 41, 2 (Feb. 2015), 198–220. doi:10.1109/TSE.2014.2363469

[25] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 265–278. doi:10.1145/1950365.1950396

[26] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 17–32. https://www.usenix.org/conference/osdi18/presentation/weidong

[27] The Box64 Developers. 2025. Box64 GitHub. https://github.com/ptitSeb/box64/issues/. Accessed on: 2025-07-30.

[28] The QEMU Project Developers. 2026. Testing in QEMU. https://www.qemu.org/docs/master/devel/testing/main.html.

[29] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. 2008. SMP-ReVirt: Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '08)*. Association for Computing Machinery, New York, NY, USA, 121–130. doi:10.1145/1346256.1346273

[30] FEX-Emu. 2019. QEMU bug report #1828867. https://bugs.launchpad.net/qemu/+bug/1828867. Accessed on: 2025-07-30.

[31] FEX-Emu. 2019. QEMU bug report #1832422. https://bugs.launchpad.net/qemu/+bug/1832422. Accessed on: 2025-07-30.

[32] Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*. Springer, 65–81. doi:10.1007/978-94-011-1793-7_4

[33] Google. 2026. Gemma (gemma3:27b). https://ollama.com/library/gemma3.

[34] Redha Gouicem, Dennis Sprokholt, Jasper Ruehl, Rodrigo C. O. Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 107–122. doi:10.1145/3567955.3567962

[35] Agam Shah -- The Register. 2021. 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/.

[36] Intel. 2025. Pin: Pin 3.21 User Guide. https://software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html. Accessed on: 2025-07-09.

[37] Intel Corporation 2007. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2*. Intel Corporation. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[38] Nathan Jay and Barton P. Miller. 2018. Structured random differential testing of instruction decoders. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 84–94. doi:10.1109/SANER.2018.8330199

[39] Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, and Kui Ren. 2022. EXAMINER: automatically locating inconsistent instructions between real devices and CPU emulators for ARM. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 846–858. doi:10.1145/3503222.3507736

[40] Soomin Kim. 2024. QEMU bug report #2175. https://gitlab.com/qemu-project/qemu/-/issues/2175. Accessed on: 2025-07-30.

[41] Soomin Kim. 2024. QEMU bug report #2495. https://gitlab.com/qemu-project/qemu/-/issues/2495. Accessed on: 2025-07-30.

[42] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungll Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 353–364. doi:10.1109/ASE.2017.8115648

[43] James C. King. 1975. A new approach to program testing. *SIGPLAN Not.* 10, 6 (April 1975), 228–233. doi:10.1145/390016.808444

[44] Stevie Lavern. 2020. QEMU bug report #1861404. https://bugs.launchpad.net/qemu/+bug/1861404. Accessed on: 2025-07-30.

[45] Ilya Leoshkevich. 2021. QEMU bug report #508. https://gitlab.com/qemu-project/qemu/-/issues/508. Accessed on: 2025-07-30.

[46] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. doi:10.1145/1538788.1538814

[47] Yves Lhuillier. 2021. QEMU bug report #364. https://gitlab.com/qemu-project/qemu/-/issues/364. Accessed on: 2025-07-30.

[48] Changming Liu, Alejandro Mera, Engin Kirda, Meng Xu, and Long Lu. 2024. CO3: Concolic Co-execution for Firmware. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 5591–5608. https://www.usenix.org/conference/usenixsecurity24/presentation/liu-changming

[49] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. doi:10.1145/3453483.3454030

[50] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration lifting: hi-fi tests for lo-fi emulators. *SIGPLAN Not.* 47, 4 (March 2012), 337–348. doi:10.1145/2248487.2151012

[51] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU emulators. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 261–272. doi:10.1145/1572272.1572303

[52] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 693–708. doi:10.1145/3037697.3037751

[53] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107. https://www.cs.dartmouth.edu/~mckeeman/references/DifferentialTestingForSoftware.pdf

[54] Robin Milner. 1989. *Communication and concurrency*. Prentice-Hall, Inc.

[55] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100. doi:10.1145/1250734.1250746

[56] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2025. RR debugger: System requirements. https://github.com/rr-debugger/rr?tab=readme-ov-file#system-requirements.

[57] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 377–389. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[58] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (CGO'10)*. Association for Computing Machinery, New York, NY, USA, 2–11. doi:10.1145/1772954.1772958

[59] pjy. 2024. QEMU bug report #2248. https://gitlab.com/qemu-project/qemu/-/issues/2248. Accessed on: 2025-07-30.

[60] Gerald J. Popek and Robert P. Goldberg. 1974. Formal requirements for virtualizable third generation architectures. *Commun. ACM* 17, 7 (July 1974), 412–421. doi:10.1145/361011.361073

[61] The QEMU Developers. 2025. QEMU Mailing List. https://lists.nongnu.org/archive/html/qemu-devel/. Accessed on: 2025-07-30.

[62] Zhenxiao Qi, Jie Hu, Zhaoqi Xiao, and Heng Yin. 2024. SymFit: Making the Common (Concrete) Case Fast for Binary-Code Concolic Execution. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 415–432. https://www.usenix.org/conference/usenixsecurity24/presentation/qi

[63] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design* (Mountain View, California) *(FMCAD '16)*. FMCAD Inc, Austin, Texas, 161–168. doi:10.1109/FMCAD.2016.7886675

[64] Sebastian Reimers, Dennis Sprokholt, Martin Fink, Theofilos Augoustis, Simon Kammermeier, Rodrigo C. O. Rocha, Tom Spink, Redha Gouicem, Soham Chakraborty, and Pramod Bhatotia. 2026. Arancini: A Hybrid Binary Translator for Weak Memory Model Architectures. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 157–174. doi:10.1145/3779212.3790127

[65] Rodrigo C. O. Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. 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)*. Association for Computing Machinery, New York, NY, USA, 888–902. doi:10.1145/3519939.3523719

[66] Eliran Roffe. 2024. QEMU bug report #2419. https://gitlab.com/qemu-project/qemu/-/issues/2419. Accessed on: 2025-07-30.

[67] Yasushi Saito. 2005. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging (AADEBUG'05)*. Association for Computing Machinery, New York, NY, USA, 69–76. doi:10.1145/1085130.1085139

[68] Koushik Sen. 2007. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 571–572. doi:10.1145/1321631.1321746

[69] James E. Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. doi:10.1016/B978-1-55860-910-5.X5000-9

[70] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 350–360. doi:10.1145/3180155.3180251

[71] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. DoublePlay: Parallelizing Sequential Logging and Replay. *ACM Trans. Comput. Syst.* 30, 1 (Feb. 2012), 3:1–3:24. doi:10.1145/2110356.2110359

[72] Peipei Wang, Hiep Nguyen, Xiaohui Gu, and Shan Lu. 2016. RDE: Replay DEbugging for Diagnosing Production Site Failures. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*. 327–336. ISSN: 1060-9857. doi:10.1109/SRDS.2016.050

[73] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 17–32. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xu-jun

[74] Lee Yong-woo. 2022. QEMU bug report #1370. https://gitlab.com/qemu-project/qemu/-/issues/1370. Accessed on: 2025-07-30.

[75] Lee Yong-woo. 2022. QEMU bug report #1371. https://gitlab.com/qemu-project/qemu/-/issues/1371. Accessed on: 2025-07-30.

[76] Lee Yong-woo. 2022. QEMU bug report #1372. https://gitlab.com/qemu-project/qemu/-/issues/1372. Accessed on: 2025-07-30.

[77] Lee Yong-woo. 2022. QEMU bug report #1374. https://gitlab.com/qemu-project/qemu/-/issues/1374. Accessed on: 2025-07-30.

[78] Lee Yong-woo. 2022. QEMU bug report #1375. https://gitlab.com/qemu-project/qemu/-/issues/1375. Accessed on: 2025-07-30.

[79] Lee Yong-woo. 2022. QEMU bug report #1376. https://gitlab.com/qemu-project/qemu/-/issues/1376. Accessed on: 2025-07-30.

[80] Lee Yong-woo. 2022. QEMU bug report #1377. https://gitlab.com/qemu-project/qemu/-/issues/1377. Accessed on: 2025-07-30.

[81] Cristian Zamfir and George Candea. 2010. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems (EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 321–334. doi:10.1145/1755913.1755946

[82] Yufeng Zhang, Zhenbang Chen, and Ji Wang. 2012. Speculative Symbolic Execution. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 101–110. ISSN: 2332-6549. doi:10.1109/ISSRE.2012.8

[83] 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. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD'21)*. Association for Computing Machinery, New York, NY, USA, 2653–2666. doi:10.1145/3448016.3457559

[84] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution reconstruction: harnessing failure reoccurrences for failure reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1155–1170. doi:10.1145/3453483.3454101

A Artifact Appendix

A.1 Abstract

The artifact is organized around four parts of the paper's evaluation. *(1)* The 17 trigger programs in Tbl. 2 exercise detection and localization of historical QEMU and Box64 mistranslations. *(2)* The SQLite, Curl, and Lua workloads in Tbl. 3 exercise deterministic replay during application validation. *(3)* Timing profiles and generated reproducers provide the measurements in Figs. 6–9. *(4)* The emulator bug-study repository provides the data and analysis behind Fig. 2 for inspection. The evaluation repository pins Focaccia, RR, the historical emulator builds, the toolchain, and the workload sources. Reproducing native traces requires matching x86-64 and AArch64 Linux hosts. The multi-architecture Docker image provides the runtime dependencies.

A.2 Artifact check-list (meta-information)

- **Algorithms and model:** the state-transition model, symbolic state abstraction, adaptive cutpoint selection, effects as actions, targeted tracing, and transition context replication.
- **Programs and binaries:** the 17 trigger programs from Tbl. 2, the three application workloads from Tbl. 3, RR, and historical QEMU and Box64 builds.
- **Data and output:** validation reports and traces, timing profiles, generated reproducers, and the five paper plots.
- **Bug-study material:** the data and analysis behind Fig. 2, provided for inspection.
- **Environment and hardware:** Docker on native x86-64 and AArch64 Linux hosts.
- **Experiments and metrics:** trigger and application validation, native-tracing and emulator-validation time, and reproducer executable size.
- **Disk space:** approximately 5 GB per host to download. Allow 25 GB per host for images and results.
- **Preparation time:** several minutes to download the image.
- **Experiment time:** a few minutes for smoke tests and several hours for a full campaign.
- **Publicly available:** evaluator, validator, bug study, and Docker image.
- **License:** BSD 3-Clause. Bundled components retain upstream licenses.
- **Archived:** https://doi.org/10.5281/zenodo.23211000.

A.3 Description

A.3.1 How to access Clone the evaluation repository from GitHub: github.com/TUM-DSE/focaccia-eval. The validator source is available at github.com/TUM-DSE/focaccia, and the bug-study source at github.com/TUM-DSE/emulator-bug-study. The evaluation lock file pins the validator, toolchain, application sources, and historical emulators. The evaluation repository also contains the trigger sources, workloads, plotting code, and deterministic fixtures.

A.3.2 Hardware dependencies A complete run uses two machines. The x86-64 host produces x86-64 oracles and validates AArch64 cases under QEMU. The AArch64 host produces AArch64 oracles and validates x86-64 cases under QEMU and Box64. All generated evidence is written below the host-side runs/ directory. Both machines must use the same directory, either through a shared filesystem or by copying the complete directory unchanged between phases.

Native capture requires privileged debugger and perf access. On x86-64, RR application capture also requires a compatible PMU. Native AArch64 recording requires a supported processor such as Arm Neoverse. Emulator validation and plotting do not require these privileges. The evaluator checks the physical ISA and does not permit an emulator under test to produce its own oracle.

A.3.3 Software dependencies The packaged workflow requires Docker on 64-bit Linux. The multi-architecture Docker image provides x86-64 and AArch64 variants, and Docker selects the matching image for the host.

A.3.4 Data sets The evaluation repository contains all trigger and workload sources. The runs/ directory contains the generated traces, replay records, reports, profiles, reproducers, and figures. The metadata records a SHA-256 hash for each consumed artifact.

A.4 Installation

Pull the image, then open Bash in `/artifacts`. The evaluator commands are in `PATH`.

```
export IMAGE=taugoust/focaccia-artifact
docker pull "$IMAGE"
mkdir -p runs
docker run --rm -it \
  -v "$PWD/runs:/artifacts" "$IMAGE"

# Inside the container
evaluate-native --help
evaluate-emulator --help
plot-evaluation --help
```

The x86-64 image also includes the full-Curl native stage. The AArch64 image includes the Box64, full-Curl QEMU, and reproducer stages.

A.5 Experiment workflow

The commands mount runs/ at /artifacts in the container.

Smoke test. Capture one oracle on each native host:

```
# x86-64 host
docker run --rm --privileged \
  --security-opt seccomp=unconfined \
  -v "$PWD/runs:/artifacts" \
  "$IMAGE" \
  evaluate
     -native --output /artifacts/smoke --case 1372

# AArch64 host: use the same command with --case 2248
```

After both captures pass, consume each oracle on the opposite host:

```
# AArch64 host
docker run --rm -v "$PWD/runs:/artifacts" \
  "$IMAGE" evaluate-qemu-1372 --input /artifacts/smoke

# x86-64 host: replace 1372 with 2248
```

Both commands should exit zero after detecting and localizing the expected mismatch.

Full evaluation. Run native collection once on each host, with the same privileged Docker options used above:

```
docker run --rm --privileged \
  --security-opt seccomp=unconfined \
  -v "$PWD/runs:/artifacts" \
  "$IMAGE" \
  evaluate-native --output /artifacts/evaluation-001
```

Then run the full-Curl native command on x86-64.

```
docker run --rm --privileged \
  --security-opt seccomp=unconfined \
  -v "$PWD/runs:/artifacts" \
  "$IMAGE" \
  evaluate-native-curl-full \
  --output /artifacts/evaluation-001
```

After both native phases finish, run the following command once on each host:

```
docker run --rm -v "$PWD/runs:/artifacts" \
  "$IMAGE" evaluate
     -emulator --input /artifacts/evaluation-001
```

The command selects all stages for the host. If a stage fails, the remaining independent stages continue and the aggregate command exits nonzero. An interruption stops the command immediately.

Generate the figures from either image. The reproducer-size file is under evaluation-001/reproducers/x86_64-linux/.

```
docker run --rm -v "$PWD/runs:/artifacts" \
  "$IMAGE" plot-evaluation \
  --input /artifacts/evaluation-001 \
  --reproducer-sizes /artifacts/evaluation-001/
     reproducers/x86_64-linux/reproducer-sizes.json
```

A.6 Evaluation and expected results

For each emulator case, Focaccia writes a human-readable validation.log and a structured validation.json report. A historical bug case should identify the configured mismatch, signal, or fault and localize it to the expected transition. A passing evaluator status means that this expected error was found, not that the mistranslated execution was accepted. The evaluator fails on unexpected crashes, replay failures, and incomplete traces.

Plotting writes five PDFs under evaluation-001/figures: trigger overhead, full-Curl tracing, application overhead, reproducer size, and the combined emulator bug study. The plotting command generates the first four only when every required measurement is present. The combined bug-study plot uses the percentages reported in Fig. 2. The evaluator does not rerun the bug-report classification; the underlying data and analysis are available for inspection in the bug-study repository.

A.7 Experiment customization

Select individual cases with the native or QEMU case commands. Use `-iterations N` for repeated measurements. The aggregate emulator command accepts only an input directory and an iteration count.

A.8 Methodology

Submission, reviewing, and badging methodology:

- ACM artifact review and badging policy
- cTuning artifact-evaluation methodology