2025 Design, Automation & Test in Europe Conference (DATE 2025)

Accurate and Extensible Symbolic Execution of Binary Code based on Formal ISA Semantics

Sören Tempel[*], Tobias Brandt[†], Christoph Lüth[‡§], Christian Dietrich[*] and Rolf Drechsler[‡§]
[*]Institute of Operating Systems and Computer Networks, Technische Universität Braunschweig, Braunschweig, Germany
[†]Independent Researcher, Bremen, Germany
[‡]Cyber-Physical Systems, DFKI, Bremen, Germany
[§]Institute of Computer Science, University of Bremen, Bremen, Germany
tempel@ibr.cs.tu-bs.de, tobbra91@gmail.com, christoph.lueth@dfki.de, dietrich@ibr.cs.tu-bs.de, drechsler@uni-bremen.de

***Abstract*—Symbolic execution is an SMT-based software verification and testing technique. Symbolic execution requires tracking performed computations during software simulation to reason about branches in the software under test. The prevailing approach on symbolic execution of binary code tracks computations by transforming the code to be tested to an architecture-independent *intermediate representation (IR)* and then symbolically executes this IR. However, the resulting IR must be semantically equivalent to the binary code, making this process complex and error-prone. The semantics of the binary code are specified by the targeted *instruction set architecture (ISA)*, commonly given in natural language and requiring a manual implementation of the transformation to an IR. In recent years, the use of formal languages to describe ISA semantics in a machine-readable way has gained increased popularity. We investigate the utilization of such formal semantics for symbolic execution of binary code, achieving an accurate representation of instruction semantics. We present a prototype for the RISC-V ISA and conduct a case study to demonstrate that it can be easily extended to additional instructions. Furthermore, we perform an experimental comparison with prior work which resulted in the discovery of five previously unknown bugs in the ISA implementation of the popular IR-based symbolic executor *angr*.**

I. Introduction

Program analysis and software testing has the goal to validate, or even verify, that certain properties hold on the different paths through a *software-under-test (SUT)*. In comparison to unit testing, *symbolic execution (SE)* [1] allows the exploration of more, or even all, paths by interpreting the SUT in a symbolic domain. Instead of concrete values ($X = 5$), we propagate symbolic values ($X > 5$) through the SUT. This requires us to bridge a *semantic gap* and *translate* the SUT (e.g., given as C code) to symbolic expressions that manipulate symbolic values. Later on, the SE engine can formally reason about branches in the SUT by using a constraint solver to check the feasibility of symbolic branch conditions.

While many translation methodologies (see Fig. 1) were proposed [2], they all exhibit some shortcomings: (1) *Direct IR-based* translation, as employed by the popular KLEE [1] engine, first lowers the program to a compiler's IR (i.e., LLVM-IR) using the existing toolchain, before *symbolizing* each IR instruction to a symbolic expression. While this

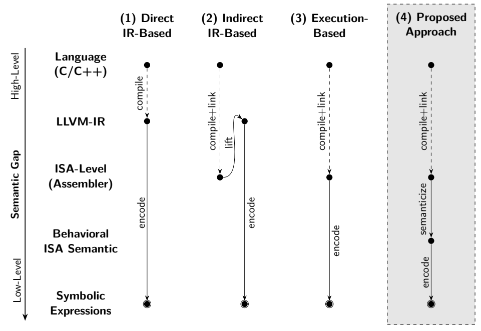


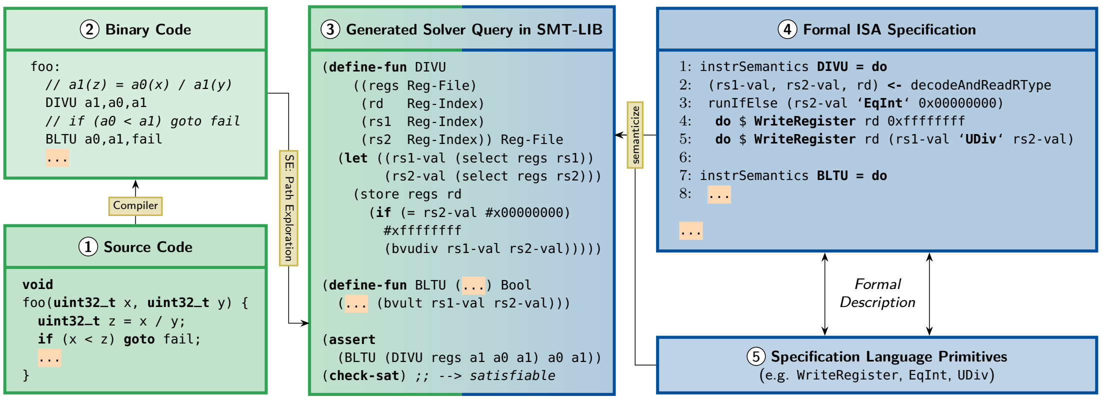
Fig. 1. Different translation methodologies for symbolic execution.

approach captures compiler bugs, it is unable to handle inline assembly, compiler intrinsics, and third-party binary code. (2) The *Indirect IR-based* approach [3]–[5] tackles this by lifting the final binary code back up to the IR level before symbolizing it down again to symbolic expressions. This involves two semantically-rich translation steps: First, the lifter has to capture the ISA semantic, which is often only available as thousands of pages of informal, natural-language specification (e.g., ARM manual [6]), making lifter construction an erroneous endeavor [7], [8]. Second, compiler IRs are usually implementation-defined and, even if formal semantics exists [9], then only as a secondary artifact that is prone to deviation. (3) The *Execution-based* method [10]–[13] avoids this error-prone detour by weaving the SE engine into execution(s) of the SUT. With an ISA-level interpreter or by trap-stepping the actual machine, symbolic expressions are emitted on the fly. While this avoids the IR problem, capturing the instruction semantics *correctly* and *completely* is still challenging, especially when confronted with feature-rich[1] or rapidly-evolving extensible ISAs.[2]

About this paper: Given the benefits and shortcomings of existing methods, we argue that (a) SE engines should represent the program with a formal "intermediate representation" but (b) that it should be located "below" the ISA level and explicitly designed to capture ISA semantics (see Fig. 2). This would give us build-chain coverage and the possibility to independently verify the ISA-level "semantification" step while retaining a strict lowering hierarchy.

This work was supported by the German Federal Ministry of Education and Research (BMBF) within projects Scale4Edge under grant no. 16ME0127, ECXL under grant no. 01IW22002, and VE-HEP under grant no. 16KIS1342.

[1]The ARMv8 Base ISA alone has 472 instructions.
[2]RISC-V has 41 ratified extensions, 12 of them newly ratified in 2024 [14].



② Binary Code
Compiler
① Source Code
SE: Path Exploration
③ Generated Solver Query in SMT-LIB
semanticize
④ Formal ISA Specification
Formal Description
⑤ Specification Language Primitives
(e.g. WriteRegister, EqInt, UDiv)

```
foo:
  // a1(z) = a0(x) / a1(y)
  DIVU a1,a0,a1
  // if (a0 < a1) goto fail
  BLTU a0,a1,fail
  ...
```

```
void
foo(uint32_t x, uint32_t y) {
  uint32_t z = x / y;
  if (x < z) goto fail;
  ...
}
```

```
(define-fun DIVU
    ((regs Reg-File)
     (rd   Reg-Index)
     (rs1  Reg-Index)
     (rs2  Reg-Index)) Reg-File
  (let ((rs1-val (select regs rs1))
        (rs2-val (select regs rs2)))
    (store regs rd
      (if (= rs2-val #x00000000)
        #xffffffff
        (bvudiv rs1-val rs2-val)))))

(define-fun BLTU (...) Bool
  (... (bvult rs1-val rs2-val)))

(assert
  (BLTU (DIVU regs a1 a0 a1) a0 a1))
(check-sat) ;; --> satisfiable
```

```
instrSemantics DIVU = do
 (rs1-val, rs2-val, rd) <- decodeAndReadRType
 runIfElse (rs2-val `EqInt` 0x00000000)
  do $ WriteRegister rd 0xffffffff
  do $ WriteRegister rd (rs1-val `UDiv` rs2-val)

instrSemantics BLTU = do
 ...

...
```

Fig. 2. Generation of an SMT solver query for an exemplary branch condition in a SUT (in binary form) based on a formal ISA specification.

Fortunately, *formal ISA specifications* [10], [15], [16] emerged in recent years to tackle the problem of ISA manuals with thousands of pages of natural-language specification. In a nutshell, these projects provide a formal, machine-readable *domain-specific language (DSL)* to express ISA-level semantics and come with tooling to work with these specifications (e.g., derive theorem-prover definitions). This direction seems so promising that RISC-V [17] and ARM [18] have recognized their potential and provide official formal specifications.

For design automation, utilization of formal ISA specifications is advantageous as—once formally specified—a variety of tools can be derived from the specification, reducing development time and thereby the time-to-market. For example, we can combine them with concurrency models [19], perform fault-injection [20], verify binary code [21], prove security properties [22], and derive emulators and documentation [15].

We argue that formal ISA specifications and their DSLs are perfectly suited to make SE extensible, accurate, and (potentially) easier to verify. With this paper, we claim the following contributions:

- We present BinSym, an SE engine for RISC-V binary code that facilitates an executable formal specification.
- We conduct a case study to demonstrate the extensibility of our approach with custom instruction set extensions.
- We validate the accuracy of BinSym by an experimental comparison to existing SE engines, leading to the discovery of five previously unknown lifter bugs in prior work.

II. Formal ISA Semantics for Symbolic Execution

We give a short primer on symbolic execution and discuss the benefits of formal ISA semantics for the SE of binary code.

A. Background on Symbolic Execution

As SE is a simulation-based software verification technique, it executes the SUT to explore its state space by enumerating reachable execution paths. Unlike unit testing, which runs the program with concrete values, SE replaces selected concrete inputs with symbolic ones that represent a set of continuously constrained concrete values. The SE engine manages this process by tracking constraints and iteratively selecting the next path to explore, a process referred to as *path selection* [2, Sect. 2.2]. During execution of a selected path, the SE engine collects constraints on symbolic values up to a previously unvisited branch. The constraints are obtained by *translating* the performed operations on symbolic values to symbolic expressions, and a constraint solver checks which upcoming branch targets are feasible under the collected constraints. Usually, we use the *satisfiability modulo theories (SMT)* formalism (i.e., SMT bitvector theory) to perform constraint checking.

In a nutshell, the classical SE engine *repeatedly* executes program paths with symbolic values on top of an SMT solver. More information on SE is available in the survey [2].

To illustrate binary-based SE, we compile an example C program ① in Fig. 2 to binary code ②. The C function `foo()` performs division of two 32-bit unsigned integers and ensures that the result is greater than or equal to the dividend (failing otherwise). Since the control flow forks at the `BLTU` branch instruction, the engine selects the path `[DIVU, BLTU]` and translates it, including the arguments, to the SMT expression ③, which we show in the standard SMT-LIB [23] format. The engine then invokes an SMT solver to check (`check-sat`) if the expression is satisfiable resp. if it is possible to take the branch. Conceptually, an SE engine repeatedly generates such queries for each encountered branch in the SUT. The example demonstrates the significant semantic gap between ISA instructions ② and SMT expressions ③ which the SE engine has to bridge – a translation that is known to be error-prone if done with handwritten translators [7], [8].

B. ISA Semantics and SMT Translation

The core idea of our proposed SE approach is: formal ISA semantics, which capture the behavior of instructions in a machine-readable specification language, are well suited as an intermediate step between binary code and SMT expressions. In our running example (Fig. 2), the normative ISA semantics are provided in a machine-readable formal language ④, which expresses instruction behavior using a DSL ⑤ (further explained in Sect. III-A). Different DSLs have been presented in prior work to express such semantics [10], [15], [18], [24].

By taking the specific ISA semantics into account, an interesting edge case in the SUT becomes apparent that has to be accounted for: in the C programming language, division by zero is undefined behavior, and the compiler can assume that it never happens. Therefore, no check for $y \equiv 0$ is inserted before the unsigned division instruction `DIVU` by the compiler. However, the formal `DIVU` semantics ④ show that the unsigned division of our ISA—instead of trapping—returns a value with all bits set if the divisor is zero. In this case, z becomes `0xffffffff`, which is larger than most 32-bit x values, whereby the `fail` branch is actually reachable with $y = 0$. This is contrary to the intuition of programmers reading ① as division usually makes numbers smaller, not larger. This emphasizes the need for a binary-based, ISA-specific SE.

To uncover such problems, the SE engine has to operate on the binary-level and translate the binary code instruction semantics to an equivalent SMT representation. As our example suggests, the translation from ② to ③ is not straightforward as it requires us to capture (a) arithmetic and logic operations as well as (b) interactions with the hardware state (e.g., the register file). Note also how the `DIVU` representation in ② advances the system state by returning a new register file, in Fig. 2 we omit details for clarity of exposition (e.g. program counter handling). The emitted SMT must be accurate wrt. the ISA specification, otherwise SUT edge cases may be missed.

Instead of directly jumping from ISA to SMT, our approach uses the primitives of a formal specification as an abstraction layer. That is, instead of directly translating the `DIVU` instruction to SMT, we employ divide-and-conquer and translate the individual language primitives ⑤ in which the semantics are expressed. For example, the `WriteRegister` primitive is translated as a write to an SMT array (`store`). By building on these primitives, SE engines also become more extensible: as long as new instructions can be expressed in terms of existing specification primitives, our SE approach can be easily extended. In Sect. IV, we will demonstrate this capability.

Our example illustrates the benefits of binary-code analysis and affirms that even supposedly simple code is expanded to complex SMT representations. Formal specifications of ISA semantics help us to mitigate this complexity, reducing the potential for errors in the translation and enabling extensibility.

III. Application to RISC-V Binary Code

In this section, we present BinSym, a prototype implementation of our proposed approach that symbolically executes 32-bit binary code for the open standard RISC-V [25] architecture. We chose RISC-V for our prototype as, due to its openness, it has enabled a large body of research on formal ISA specifications and even provides an official golden formal specification [17]. Furthermore, RISC-V is a modular architecture, i.e., it consists of a base instruction set and optional extensions, which are implemented on top and can be combined as needed. Therefore, it benefits immensely from an extensible SE approach as the specification is constantly expanding, requiring binary analysis tools to “catch up” to it.

A. Executable Formal Specifications

A variety of different formal ISA specifications for RISC-V have emerged in recent years which target different use cases [10], [15], [16], [24]. Since software execution is the focus of our work, we make use of an *executable formal specification*. Such specifications allow for the creation of custom *modular interpreters* [26] which are responsible for interpreting the language primitives used by the specification. These interpreters are modular in the sense that the executable specification provides generic versions of essential components (e.g., the register file or memory) which can be reused by the interpreter. Different executable specifications have been presented in prior work [10], [16], [24]. For our BinSym prototype implementation, we build upon the open source LibRISCV [10] specification. Like many executable formal specifications [16], [24], LibRISCV describes the ISA using a DSL that is embedded into the functional and strongly-typed general-purpose programming language Haskell.

In order to illustrate this DSL, we discuss the formal description of the RISC-V `DIVU` instruction as provided by LibRISCV. We have already seen this formal description in top-right corner ④ of Fig. 2 and in the following, we describe it in greater detail. The formal `DIVU` description in Fig. 2 starts off by specifying the operands of the instruction in Line 2. As mandated by the RISC-V specification, `DIVU` is an R-Type instruction with three register operands (`rs1`, `rs2`, and `rd`). The instruction semantics are described in terms of these operands (Line 3 - Line 5). If the divisor (`rs2`) is zero (Line 3), then the destination register (`rd`) has all bits set (Line 4). Otherwise, standard unsigned division is performed on the dividend (`rs1`) in Line 5. This description is entirely abstract and does *not assume* a specific representation of operands.

B. BinSym: A Modular Symbolic Interpreter

The `DIVU` example serves to illustrate that LibRISCV abstractly describes instruction semantics in terms of several language primitives: (1) stateful ones and (2) arithmetic/logic primitives. Since LibRISCV is an executable formal specification, it takes RISC-V binary code (in the ELF format) as an input and converts the instruction stream to a continuous, lazily evaluated sequence of these primitives. The aforementioned modular interpreter is then responsible for processing this sequence. As an example, prior work has presented a concrete interpreter and an interpreter performing dynamic information flow tracking [10]. For BinSym, we implemented a symbolic modular interpreter that utilizes the language primitives as an abstraction layer for the implementation of a binary-level SE engine. In Fig. 1, BinSym implements the *semanticize* step.

For SE, instruction operands are symbolic values (e.g., the value of the register $x1$ may be symbolic). As such, we need variants of the register file and memory that are capable of operating on such symbolic values. Due to the utilization of an executable formal model, we were able to reuse existing components from LibRISCV for this purpose, such as a generic implementation of a register file which is parameterized over a value type. This significantly reduced the effort required for our BinSym prototype implementation and is one major benefit of an executable formal specification. The symbolic representation of the hardware state enables us to symbolically interpret stateful LIBRISCV language primitives such as `WriteRegister`. Additionally, we had to map arithmetic and logic primitives to operations of an SMT solver. We make use of the Z3 [27] SMT solver in BINSYM, translating arithmetic and logic operations of LIBRISCV to Z3 SMT solver operations. This is the *encode* step from Fig. 1.

The outlined translation enables us to propagate and track symbolic values throughout program execution. Based on the propagated values, we can symbolically reason about branch points during execution of the SUT. That is, every time we encounter a branch (denoted via the `runIfElse` language primitive) that depends on a symbolic value, we can use Z3 to check if both the true and the false case are satisfiable and if so, explore both in parallel. For example, if a SUT executes a RISC-V `DIVU` instruction with a symbolic divisor operand, we construct an SMT query to check if it is possible for the divisor to be zero/non-zero under the current constraints. On the technical side, BINSYM implements a so-called offline executor, which continuously restarts execution of the SUT with input values obtained for branch points from the solver [2, Sect. 2.4]. Specifically, it implements dynamic symbolic execution [2, Sect. 2.1] with depth-first search path selection [2, Sect. 2.2] and address concretization [2, Sect. 3.2]. These are well-established SE algorithms (details in the references).

Our BINSYM prototype implements the entire SE engine for RISC-V binary code in only 1000 LOC in Haskell with 1500 LOC of LIBRISCV specification. This reduced complexity can be attributed to the utilization of an executable formal specification and its implementation as a modular interpreter. Please note that while our prototype focuses on RISC-V, it is by no means limited to this architecture. Formal semantics are available for many ISAs and prior work has demonstrated that it is feasible to capture the semantics of different ISAs using a common set of language primitives [15].

IV. CASE STUDY: SUPPORTING A CUSTOM INSTRUCTION

With the advent of RISC-V, custom instructions are becoming increasingly popular for power savings in embedded systems or increased throughput in high-performance processors [28, Sect. 2]. Utilization of formal ISA semantics significantly eases supporting—and experimenting with—custom instructions during design space exploration. Once formally described, documentation, simulators, fault-injection tooling, et cetera can be derived [15], [20], [29]. Thereby, formal semantics significantly reduce the effort required to support custom instructions throughout the development process, contributing to a reduced time-to-market. In the following, we conduct a case study with an exemplary custom instruction to demonstrate that—once formally specified—custom instructions can be easily supported in our proposed SE approach.

For our case study, we define a new non-standard `MADD` instruction which takes three register operands (`rs1`, `rs2`, `rs3`) and combines multiplication and addition into a single instruction computing: $(rs1 \times rs2) + rs3$. In order to support this custom instruction in our SE engine, we first need to specify how it is encoded. For this purpose, LIBRISCV

```
1 madd:
2   encoding: '-----01------------------1000011'
3   extension: [rv_zimadd]
4   mask: '0x600007f'
5   match: '0x2000043'
6   variable_fields: [rd, rs1, rs2, rs3]
```

Fig. 3. YAML `riscv-opcodes` description of a custom `MADD` instruction.

utilizes the existing `riscv-opcodes`[3] instruction format provided by the RISC-V Foundation. Fig. 3 provides the description of the `MADD` encoding in this format. Essentially, the format defines two bitmasks (Line 4 and Line 5) which can be used to uniquely identify the instruction's opcode. Further, it specifies which well-known instruction operand fields are used by the instruction (Line 6). The existing LIBRISCV tooling automatically generates the decoding code from this description [10, Sect. 4.1].

In addition to the instruction encoding, we also need to specify the instruction semantics in terms of the language primitives provided by LIBRISCV. The formal description of our custom `MADD` instruction is shown in Fig. 4. The instruction is decoded as an R4-Type instruction (Line 2), then sign-extension and multiplication of `rs1` and `rs2` is performed (Line 4 - Line 5), the lower 32-bit are incremented by `rs3` and stored in the destination register (Line 6). The semantics of `MADD` can be expressed entirely in terms of existing LIBRISCV language primitives. As our BINSYM prototype already maps all of these primitives to SMT semantics, no modifications of BINSYM are needed in order to support this instruction. In total, we only had to integrate the 7 lines of YAML from Fig. 3 and the 7 lines of Haskell code from Fig. 4 into the formal ISA specification to support SE with `MADD`.

Naturally, the formal description of the `MADD` instruction (as shown in Fig. 4) can not only be used for symbolic semantics but also for other design automation tooling. This serves to illustrate that our approach can be easily extended to support additional instructions and, moreover, integrates well with a design flow centered around formal ISA semantics.

V. EVALUATION

We evaluate our approach on five programs: three real-world modules from the RIOT operating system [30] (`base64-encode`, `clif-parser` and `uri-parser`) and two synthetic benchmark applications (`bubble-sort` and `insertion-sort`). The latter benchmark applications have also been used in prior work for evaluation purposes [31, Sect. 4.2]. All programs have been compiled for the 32-bit RISC-V architecture; we can therefore only compare against prior SE work that supports this architecture. We are presently

[3]https://github.com/riscv/riscv-opcodes

```
1 instrSemantics MADD = do
2   (rs1, rs2, rs3, rd) <- decodeAndReadR4Type
3   let
4     multResult = (sext rs1) `Mul` (sext rs2)
5     multTrunc  = extract32 0 multRes
6   WriteRegister rd $ (multTrunc `Add` rs3)
```

Fig. 4. Description of the custom `MADD` semantics in LIBRISCV.

TABLE I
AMOUNT OF EXECUTION PATHS FOUND BY DIFFERENT SE ENGINES.

Benchmark	angr	BINSEC	SYMEX-VP	BINSYM
`base64-encode`	125 †	6250	6250	6250
`bubble-sort`	720	720	720	720
`clif-parser`	11424	11424	11424	11424
`insertion-sort`	5040	5040	5040	5040
`uri-parser`	8194 †	8240	8240	8240

aware of the following SE engines that fulfill this criterion: angr [5], BINSEC [4], and SYMEX-VP [32].

We are interested in two evaluation aspects: (a) does our work discover the same amount of execution paths as prior work and (b) does our work achieve competitive SE performance? Therefore, we symbolically executed the programs with a fixed-size input of symbolic values. All benchmarks are fully explorable, i.e., all engines can discover the same amount of execution paths on all benchmarks. Additionally, all tested SE engines have been configured to use the same version of Z3 [27] to avoid benchmarking the solver. We focus on path coverage here, not the detection of bugs in the SUT.

A. Exploration Results

We compare the results of the symbolic exploration in Table I. This table contains one row for each tested program and compares the execution paths found by the aforementioned SE engines. As evident by the gathered data, angr fails to discover all execution paths in real-world modules from the RIOT operating system. Specifically, angr misses 6125 paths in the `base64-encode` program and 46 paths in the `uri-parser` program (see the cells marked with a † in Table I). These execution paths were found by all other tested engines, including our BINSYM prototype. Further debugging of angr revealed that these execution paths are missed due to implementation errors in the RISC-V lifter provided by angr. In total, *we found five previously unknown bugs* which have been reported, acknowledged, and fixed by angr developers:[4]

1) The arithmetic shift operation (e.g., as used in the `SRA` instruction) was modeled incorrectly in the lifter.
2) The R-Type shift instructions used the lower bits of the register index, not the register value, as a shift amount.
3) The lifted load instructions did not correctly zero- and sign-extend the resulting register value.
4) The shift amount for I-type shift instructions was treated as a signed, instead of unsigned, integer value.
5) The signed comparison instructions compared for unsigned instead of signed integer equality.

All inaccuracies are caused by programming errors in the manual implementation of the natural language ISA specification within the lifter provided by angr. These programming errors can result in both false-positives and false-negatives during automated SE-based software testing. As an example, consider the C code in Fig. 5 which causes SUT analysis using angr to result in both a false-negative and false-positive.

```
1 void parse_word(uint32_t x) {
2   uint32_t mask = x << 31;
3   if (x == 1)
4     assert(mask == 0x80000000); // FALSE-POSITIVE
5   else
6     assert(mask != 0x80000000); // FALSE-NEGATIVE
7 }
```

Fig. 5. Example code which results in false-positive and false-negative when analyzed using the existing angr SE engine due to a bug in the RISC-V lifter.

This code calculates a bitmask based on a function parameter `x`, if `x == 1` it expects the bitmask to be `0x80000000` and ensures that this is the case using an assertion (Line 4). The code is compiled to use an I-type shift instruction, for such instructions angr incorrectly treats the 5-bit shift amount immediate as a signed integer in two's complement. Therefore, it ends up shifting by `-1` instead of `31`. This results in a spurious assertion failure as the bitmask is not equal to `0x80000000` if `x == 1` (i.e., angr finds a false-positive in the SUT). Similarly, the code contains an additional assertion which (incorrectly) assumes that `x == 1` is the only input to generate the bitmask value `0x80000000`. However, this assertion is incorrect as there are other values for `x` which result in such a bitmask value (e.g., `0xffffffff`). Unfortunately, due to the issue outlined above, angr fails to find such an input, resulting in a false-negative. In real-world code, large amounts of false-positives and false-negatives can significantly hinder adaption of SE [33] and must therefore be avoided. With our approach, we can—conceptually—avoid errors in the implementation of the ISA semantics by building on a normative and therefore authoritative formal ISA specification.

B. Performance Comparison

Utilizing the fixed version of angr, we experimentally compare SE performance. To this end, we executed all programs five times with each engine on an Intel Xeon Gold 6240 Linux system. The arithmetic mean over all five executions is visualized as a grouped bar chart in Fig. 6. In this figure, the absolute execution time—as the arithmetic mean over all five executions—is given logarithmically in seconds on the y-axis while the x-axis lists the benchmarks. For each benchmark, four bar charts are given which correspond to the aforementioned SE engines; from left to right: BINSEC (purple), BINSYM (green), SYMEX-VP (brown) and angr (teal). Maximum standard deviation across all executions is 5 %, which we consider negligible. The results can be reproduced using the provided evaluation artifacts [34] and are consistent across all benchmark applications: BINSEC is the fastest engine, followed by our BINSYM prototype implementation and prior work on SYMEX-VP. The slowest engine across all benchmarks is angr. This is expected and congruent with prior work which saw similar results for angr and attributes its "lower execution rate to the fact that its symbolic reasoning is implemented in Python" [35, Sect. 5.4]. Similarly, SYMEX-VP executes software in a SystemC [36] simulation environment, which enables it to support interactions with hardware peripherals modeled in SystemC [36] but

[4]https://github.com/angr/angr-platforms/pull/64

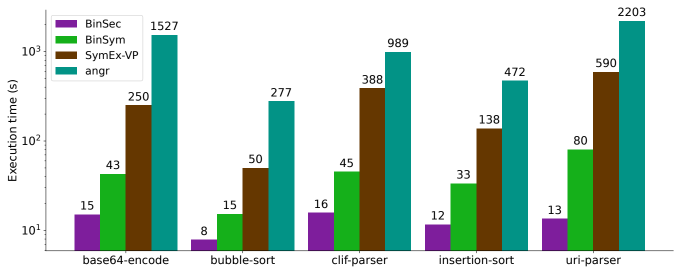


Fig. 6. Total execution time as an arithmetic mean over five executions per benchmark. The y-axis is logarithmic as results for angr constitute an outlier.

incurs a performance penalty [32, Sect. 3.2]. BinSec is not subject to these limitations and is one of the most mature and optimized SE engines for binary code. Hence, it achieves better simulation performance than BinSym, which is a prototype of our translation approach, and subsequently lacks advanced optimizations in the exploration and solver components.

Nonetheless, overall, the results from Fig. 6 show that BinSym offers competitive execution time performance. That is, the results indicate that the technologies and techniques used in BinSym (i.e., executable formal ISA semantics) do not negatively impact performance in a way that utilization for SE becomes infeasible. Unfortunately, as pointed out by prior work, it remains challenging to isolate which design decisions contribute to overall SE performance as part of an empirical comparison [35]. Since we now know that the overall performance is competitive, we plan to expand on the evaluation in future work by specifically investigating the impact of formal ISA semantics on SMT query complexity.

VI. Related Work

Symbolic Execution is an active research area on which Baldoni *et al.* [2] give a recent overview. As Sect. I already categorizes different translation methodologies to SMT, we only discuss the further SE-related work here. In contrast to *source-based* SE, of which KLEE [1] is most popular, *binary-based* methods, like our BinSym, do not require access to the source code. While initial binary-based tools [31], [37], [38] built upon KLEE, and thus LLVM-IR, also other high-level IRs, such as Valgrind VEX [5], [39] or DBA [4], [40] were utilized. These methods, which all rely on binary lifting [35], use the IR as a semantically-rich "meta"-ISA to unify the different ISAs. We propose to use formal ISA specifications, which were designed with verifiability in mind, to replace the lifting process with a straightforward per-instruction lowering.

Translational Correctness requires us to show the equivalence between binary code and symbolic expressions. If we use a lifter in the process, we have to show two equivalences: binary$\leftrightarrow$IR and IR$\leftrightarrow$symbolic expressions. Previous validation efforts focused on the former [7], [8], [41] and revealed many bugs in existing tools. This is in line with our findings (see Sect. V) that manual lifter design is an error-prone process. With our proposed approach, the ISA semantics come in the form of an independently test- and verifiable artifact to the process. The aforementioned prior work also acknowledges that there are "few existing approaches to testing the correctness of binary lifters" [8]. Unsurprisingly, there is even less work that goes beyond lifter correctness and also validates the symbolic semantics [42]–[44]. Presently, such work operates on toy programming languages and, to the best of our knowledge, there is no prior work which concerns itself with the correctness of symbolic semantics for real-world binary code. The formal representation of ISA semantics in BinSym is an ideal springboard to formally prove the equivalence between the ISA semantics and the SMT representation in future work.

Formal Specifications of programming semantics have already been used for SE in prior work [45]. However, formal specifications of ISA semantics have only gained increased relevance in recent years with Sail [15] being the most extensive work and selected in 2019 as the official formal RISC-V specification [17]. Utilization of such formal ISA specifications for SE of binary code is presently limited. Goel *et al.* [46] use a partial ISA model to perform automated proofs on x86 binaries through SE based on *binary decision diagrams (BDDs)* instead of SMT, limiting its applicability to general programs. Prior work on retargetable tooling (e.g., TSL [47] and CDT [48]) generates code for binary analysis tooling from an informal, non-executable architecture description. Unfortunately, maintaining a diverse set of code generators for different analysis tasks is a laborious undertaking [16, Sect. 1]. In contrast, our BinSym prototype is based on an executable specification in the lineage of prior work on modular interpreters [26], enabling reuse of existing components and significantly reducing the SE engine's complexity.

VII. Conclusion

We present BinSym, an approach for symbolic execution of binary code that is accurate wrt. formal ISA specifications. Such specifications are easily extensible and increasingly adopted by modern ISAs. The novelty of our approach is the utilization of an *executable* formal specification: we derive the SE engine as a modular interpreter for this specification, which reduces its complexity and helps to avoid bugs during the translation from ISA instructions to SMT expressions.

With the BinSym prototype, we show that our approach allows to implement a complete SE engine for RISC-V in 1000 lines of code, excluding the formal ISA specification. We illustrate the extensibility with a custom instruction and experimentally compare BinSym to three existing SE engines. Thereby, we discovered five previously unknown bugs in angr, a popular IR-based symbolic executor, which can result in missing or incorrect SE results. These bugs originate from the manually-written binary-to-IR lifter in angr, highlighting the importance of deriving all parts of the testing toolchain from a single authoritative ISA specification. Further, our experiments indicate that BinSym achieves competitive SE simulation performance in comparison to existing prior work. To stimulate further research on binary-based SE with formal ISA specifications, BinSym is available as open source software.[5]

[5]https://github.com/agra-uni-bremen/binsym

References

[1] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.

[2] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018. DOI: 10.1145/3182657.

[3] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *CAV*, 2011. DOI: 10.1007/978-3-642-22110-1_37.

[4] A. Djoudi and S. Bardin, "BINSEC: Binary code analysis with low-level regions," in *TACAS*, 2015. DOI: 10.1007/978-3-662-46681-0_17.

[5] Y. Shoshitaishvili, R. Wang, C. Salls, *et al.*, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE S&P*, 2016. DOI: 10.1109/SP.2016.17.

[6] ARM Limited, *ARM architecture reference manual. ARMv8, for ARMv8-A architecture profile*, v8.2 Beta, 2017.

[7] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher, "Scalable validation of binary lifters," in *PLDI*, 2020. DOI: 10.1145/3385412.3385964.

[8] S. Kim, M. Faerevaag, M. Jung, *et al.*, "Testing intermediate representations for binary analysis," in *ASE*, 2017. DOI: 10.1109/ASE.2017.8115648.

[9] Y. Zakowski, C. Beck, I. Yoon, I. Zaichuk, V. Zaliva, and S. Zdancewic, "Modular, compositional, and executable formal semantics for LLVM IR," *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, 2021. DOI: 10.1145/3473572.

[10] S. Tempel, T. Brandt, and C. Lüth, "Versatile and flexible modelling of the RISC-V instruction set architecture," in *Trends in Functional Programming*, 2023. DOI: 10.1007/978-3-031-38938-2_2.

[11] I. Yun, S. Lee, M. Xu, *et al.*, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium*, 2018.

[12] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *DAC*, 2019. DOI: 10.1145/3316781.3317807.

[13] D. W. Currie, A. J. Hu, and S. Rajan, "Automatic formal verification of DSP software," in *DAC*, 2000. DOI: 10.1145/337292.337339.

[14] RISC-V Foundation. "Ratified extensions." (2024), [Online]. Available: https://wiki.riscv.org/display/HOME/Ratified+Extensions.

[15] A. Armstrong, T. Bauereiss, B. Campbell, *et al.*, "ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 2019. DOI: 10.1145/3290384.

[16] T. Bourgeat, I. Clester, A. Erbsen, *et al.*, "Flexible instruction-set semantics via abstract monads (experience report)," *Proc. ACM Program. Lang.*, vol. 7, no. ICFP, 2023. DOI: 10.1145/3607833.

[17] RISC-V Foundation, *ISA Formal Spec Public Review*. [Online]. Available: https://github.com/riscv/ISA_Formal_Spec_Public_Review.

[18] A. Reid, "Trustworthy specifications of ARM® v8-A and v8-M system level architecture," in *FMCAD*, 2016. DOI: 10.1109/FMCAD.2016.7886675.

[19] A. Armstrong, B. Campbell, *et al.*, "Isla: Integrating full-scale ISA semantics and axiomatic concurrency models," in *Lecture Notes in Computer Science*, 2021. DOI: 10.1007/978-3-030-81685-8_14.

[20] C. Dietrich, M. Bargholz, Y. Loeck, M. Budoj, L. Nedaskowskij, and D. Lohmann, "SailFAIL: Model-derived simulation-assisted ISA-level fault-injection platforms," in *SAFECOMP*, 2022. DOI: 10.1007/978-3-031-14835-4_14.

[21] M. Sammler, A. Hammond, R. Lepigre, *et al.*, "Islaris: Verification of machine code against authoritative ISA semantics," in *PLDI*, 2022.

[22] T. Bauereiss, B. Campbell, T. Sewell, *et al.*, "Verified security for the morello capability-enhanced prototype ARM architecture," in *ESOP*, 2022.

[23] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB standard version 2.6," Standard, May 12, 2021. [Online]. Available: https://smt-lib.org/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.

[24] B. Selfridge, "GRIFT: A richly-typed, deeply-embedded RISC-V semantics written in Haskell," in *SpISA 2019: Workshop on Instruction Set Architecture Specification*, 2019. [Online]. Available: https://www.cl.cam.ac.uk/~jrh13/spisa19/paper_10.pdf.

[25] RISC-V Foundation, *The RISC-V instruction set manual, volume I: User-level ISA*, Document Version 20191213, 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[26] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *POPL*, 1995. DOI: 10.1145/199448.199528.

[27] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008. DOI: 10.1007/978-3-540-78800-3_24.

[28] E. Cui, T. Li, and Q. Wei, "RISC-V instruction set architecture extensions: A survey," *IEEE Access*, vol. 11, 2023. DOI: 10.1109/ACCESS.2023.3246491.

[29] S. Tempel, T. Brandt, C. Lüth, and R. Drechsler, "Minimally invasive generation of RISC-V instruction set simulators from formal ISA models," in *FDL*, 2023. DOI: 10.1109/FDL59689.2023.10272224.

[30] E. Baccelli, C. Gündoğan, O. Hahm, *et al.*, "RIOT: An open source operating system for low-end embedded devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, 2018. DOI: 10.1109/JIOT.2018.2815038.

[31] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *27th USENIX Security Symposium*, 2018.

[32] S. Tempel, V. Herdt, and R. Drechsler, "SymEx-VP: An open source virtual prototype for OS-agnostic concolic testing of IoT firmware," *JSA*, 2022. DOI: 10.1016/j.sysarc.2022.102456.

[33] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Inf. Softw. Technol.*, vol. 53, no. 4, 2011. DOI: 10.1016/j.infsof.2010.12.007.

[34] S. Tempel, T. Brandt, C. Lüth, and R. Drechsler, *Benchmarks for comparing binary-level symbolic execution speed*, 2024. DOI: 10.5281/zenodo.12599534.

[35] S. Poeplau and A. Francillon, "Systematic comparison of symbolic execution systems: Intermediate representation and its generation," in *ACSAC*, 2019. DOI: 10.1145/3359789.3359796.

[36] System C Standardization Working Group, *IEEE Standard for Standard SystemC Language Reference Manual*, 2012. DOI: 10.1109/SBAC-PAD.2004.8.

[37] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *ASPLOS XVI*, 2011. DOI: 10.1145/1950365.1950396.

[38] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *22nd USENIX Security Symposium*, 2013.

[39] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007. DOI: 10.1145/1250734.1250746.

[40] S. Bardin, P. Herrmann, J. Leroux, R. Tabary, and A. Vincent, "The BINCOA framework for binary code analysis," in *CAV*, 2011. DOI: 10.1007/978-3-642-22110-1_13.

[41] J. Hendrix, G. Wei, and S. Winwood, "Towards verified binary raising," in *SpISA 2019: Workshop on Instruction Set Architecture Specification*, 2019. [Online]. Available: https://www.cl.cam.ac.uk/~jrh13/spisa19/paper_05.pdf.

[42] A. Correnson and D. Steinhöfel, "Engineering a formally verified automated bug finder," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023. DOI: 10.1145/3611643.3616290.

[43] A. W. Appel, "Verismall: Verified smallfoot shape analysis," in *Certified Programs and Proofs*, 2011. DOI: 10.1007/978-3-642-25379-9_18.

[44] S. Keuchel, S. Huyghebaert, G. Lukyanov, and D. Devriese, "Verified symbolic execution with kripke specification monads (and no metaprogramming)," *Proc. ACM Program. Lang.*, vol. 6, no. ICFP, 2022. DOI: 10.1145/3547628.

[45] D. Lucanu, V. Rusu, and A. Arusoaie, "A generic framework for symbolic execution: A coinductive approach," *Journal of Symbolic Computation*, vol. 80, 2017. DOI: 10.1016/j.jsc.2016.07.012.

[46] S. Goel and W. A. Hunt, "Automated code proofs on a formal model of the x86," in *Verified Software: Theories, Tools, Experiments*, 2014. DOI: 10.1007/978-3-642-54108-7_12.

[47] J. Lim and T. Reps, "TSL: A system for generating abstract interpreters and its application to machine-code analysis," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 1, 2013. DOI: 10.1145/2450136.2450139.

[48] A. Ibing, "Architecture description language based retargetable symbolic execution," in *DATE*, 2015. DOI: 10.7873/DATE.2015.0750.