Symbolic Execution of x86 assembly in Isabelle/HOL

Freek Verbeek
Virginia Tech, USA
freek@vt.edu

Abhijith Bharadwaj
Virginia Tech, USA

Joshua A. Bockenek
Virginia Tech, USA

Ian Roessle
Joint Artificial Intelligence Center, Washington DC, US

Binoy Ravindran
Virginia Tech, USA

Abstract

In this short paper we present progress on a symbolic execution engine for x86 assembly in the Isabelle/HOL theorem prover. We discuss the two main challenges tackled: 1.) how to leverage reliable machine-learned semantics of x86 assembly instructions, and 2.) how to generate preconditions that allow deterministic symbolic execution of basic blocks. We end with a discussion on how we intend to use our symbolic execution engine.

2012 ACM Subject Classification Software and its engineering → Formal methods

Keywords and phrases Symbolic Execution, x86-64 assembly, Theorem Proving

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements This work is supported in part by ONR under grant N00014-17-1-2297 and NAVSEA/NEEC grant N00174-16-C-0018.

1 Introduction

Symbolic execution is a powerful technique in program verification and analysis [7, 2]. It can be used to explore an overapproximation of all possible paths. In case of assembly code, it can also be used to *summarize* state changes induced by sequences of individual assembly instructions. In assembly, one will typically find series of instructions whose net effect can be described much more succinctly than by using the semantics of the individual instructions. As example, consider the x86 assembly sequence `push rbp; pop rbp`. The net effect is only a single write into memory (register `rbp` is written to the top of the stack frame). The succinct output of symbolic execution can be the base for further for formal verification.

This short paper describes our progress in building a formal symbolic execution engine in Isabelle/HOL [5] for x86-64. Our symbolic execution engine targets basic blocks, i.e., blocks without unconditional jumps. We specifically deal with the following two challenges:

- The semantics of x86 are typically highly complicated and its CISC nature requires formal semantics for many instructions. The Intel manuals provide documentation, but translating these into a formal model is error-prone and requires human interpretation. We use *Strata* [4] to embed highly trustworthy machine-learned instruction semantics into Isabelle/HOL. The challenge is that, since these semantics are not manually written but machine-learned, they are typically not in a form suitable for formal verification. We thus provide manually written *presimplified semantics* and prove equivalence between our manually written semantics and the machine-learned version.

- Once each instruction in a block has been given semantics, symbolic execution amounts to aggregating these individual state changes. The objective is that one basic block has a deterministic aggregated state change, since each individual instruction is deterministic. However, since all values are symbolic, addresses are typically symbolic as well. This leads to the *memory aliasing* problem: if two values are written to memory to symbolic addresses a_0 and a_1, it is possible that they overwrite each other, overlap each other, or are separate. We generate preconditions under which symbolic execution becomes deterministic for basic blocks.

2 Using machine-learned semantics

Strata uses a stochastic search methodology to derive instruction semantics from an x86-64 machine. The search space used to learn instructions consists of 62 hard-coded base instructions. These base instructions cover bit-vector operations such as integer arithmetic, bitwise operations, data movement, floating point operations, splitting and combining of registers, and setting and clearing of status flags. The base set covers fundamental operations, serving as building blocks for the more complex instructions. Ultimately, semantics are learned as assignments of bit-vector formula's to state parts.

In [6], we describe a methodology for generalizing the output of Strata, and lifting it into the Isabelle/HOL theorem prover. As an example, we consider the instruction variant `sub r32 m32`, which subtracts the value stored in the 32-bit memory location from the value stored in the 32-bit register. Note that in x86-64, a 32-bit register is actually the lower part of a 64-bit register. This instruction thus actually reads from and writes to a 64-bit register. We also show two of the flags: the zero flag and the carry flag.

$$\begin{array}{lll} \mathit{r64} & := & \underset{32}{\underline{0}} \smile \langle 31,0\rangle(\underset{1}{\underline{0}} \smile \neg \mathit{m32} + \underset{33}{\underline{1}} + \underset{1}{\underline{0}} \smile \langle 31,0\rangle(\mathit{r64})) \\ \mathit{ZF} & := & \langle 31,0\rangle(\underset{1}{\underline{0}} \smile \neg \mathit{m32} + \underset{33}{\underline{1}} + \underset{1}{\underline{0}} \smile \langle 31,0\rangle(\mathit{r64})) == \underset{32}{\underline{0}} \\ \mathit{CF} & := & \langle 32,32\rangle(\underset{1}{\underline{0}} \smile \neg \mathit{m32} + \underset{33}{\underline{1}} + \underset{1}{\underline{0}} \smile \langle 31,0\rangle(\mathit{r64})) == \underset{32}{\underline{1}} \end{array}$$

It can be seen that the semantics are expressed in base instructions such as concatenation ($\smile$), taking a sub-bit-vector ($\langle 31,0\rangle$), negation, addition, constants ($\underset{33}{\underline{1}}$ means "the constant 1 in 33-bit mode) and equality. These semantics, however, also seem overly complicated. In order to express the semantics of the zero flag, for example, the input values are extended to 33-bit mode, after which a two's complement subtraction happens. Then the lower 32 bits of 33 are compared to 0. Humanly defined semantics would simply state `r32 == m32`, i.e., the zero flag after subtraction is set when its inputs are equal. We thus defined the following manually written presimplified semantics:

$$\begin{array}{lll} \mathit{r64} & := & \text{zextend}(\langle 31,0\rangle(\mathit{r64}) - \mathit{m32}) \\ \mathit{ZF} & := & \langle 31,0\rangle \mathit{r64} = \mathit{m32} \\ \mathit{CF} & := & \langle 31,0\rangle \mathit{r64} < \mathit{m32} \end{array}$$

These two semantics are formally proven to be equivalent. We have presimplified semantics for 84 instruction mnemonics, where each mnemonic has several variants. For example, `sub` is a mnemonic with 8, 16, 32 and 64 bit variants, and each of these variants has further variation in whether its operands are memory or registers. Not all semantics come from Strata, e.g., the shift instructions have been defined manually. More details can be found in [6].

3 Determinizing Symbolic Execution

Consider the following x86 assembly sequence:

```
mov QWORD PTR [rsp-16], 1
mov DWORD PTR [rsp-24], 2
mov rax, QWORD PTR [rsp-16]
```

The first instruction moves the quad (8 byte) word `1` to memory location `[rsp-16]`. The second moves the 4 byte word `2` to memory location `[rsp-24]`. The third moves 8 bytes from memory location `[rsp-16]` into the `RAX` register. Symbolic execution should produce the following:

$$s' = s\llparenthesis[\texttt{rsp} - 16] := \underbrace{1}_{64}, [\texttt{rsp} - 24] := \underbrace{2}_{32}, \texttt{RAX} := \underbrace{1}_{64}\rrparenthesis$$

That is, the new state s' is the result of three state changes with respect to the input state s. For sake of presentation, the instruction pointer is omitted. We illustrate that in order to get this seemingly trivial result, we require both extra preconditions and solving of linear equations.

Symbolic execution starts in state s and sequentially applies the presimplified semantics of the current instruction. After execution of the first instruction, the current symbolic state is:

$$s' = s\llparenthesis[\texttt{rsp} - 16] := \underbrace{1}_{64}\rrparenthesis$$

Now, in order to execute the second instruction, it needs to be established that the two regions written to are separate. If they are separate, the next symbolic state is equal to:

$$s' = s\llparenthesis[\texttt{rsp} - 16] := \underbrace{1}_{64}, [\texttt{rsp} - 24] := \underbrace{2}_{32}\rrparenthesis$$

However, were they to overlap, then a different symbolic state would be produced.

To know whether they are separate, the following linear equation must be solved:

$$\texttt{rsp} - 16 + 8 \leq \texttt{rsp} - 24 \vee \texttt{rsp} - 24 + 4 \leq \texttt{rsp} - 16$$

This seems a trivial linear equation, since $\texttt{rsp} - 20 \leq \texttt{rsp} - 16$. However, the addresses are computed in 64-bit mode, i.e., the address computations are modulo 2^{64}. Thus, the equation is not true: if for example $\texttt{rsp} = 16$, then $\texttt{rsp} - 20 > \texttt{rsp} - 16$. When the extra precondition $\texttt{rsp} \geq 20$ is assumed, the linear equation can be solved and we can complete symbolic execution deterministically.

Our solution to this problem is as follows:

1. For each basic block, identify the accessed regions;
2. For each region, generate the preconditions necessary to prevent under- and overflow;
3. For each pair of two regions, precompute whether they are separate, and whether they are enclosed in each other;
4. For each basic block, generate a lemma in Isabelle/HOL with as assumptions the generated preconditions and the precomputed relations.

Step 3 uses the Z3 theorem prover [3]: for each pair of regions, linear equations are generated that model separation and enclosure. This also prevents a vacuous truth: since the assumptions are generated, we need to make sure that they are internally consistent. Z3 ensures that we cannot add an assumption such as "$[\texttt{rsp} - 16, 8]$ is separate from $[\texttt{rsp} - 12, 8]$".

Note that in the given example, the basic block *is* deterministic. In general, that is not necessarily the case, e.g., in case of aliasing. Consider the following example:

```
mov QWORD PTR [rdi], 1
mov DWORD PTR [rsi], 2
mov rax, QWORD PTR [rdi]
```

Symbolic execution cannot produce a deterministic value for register `RAX`, since it depends on the values of registers `RDI` and `RSI`. In this case, the Isabelle symbolic execution will fail, and the user manually needs to insert as assumption that, e.g., [`RDI`, 8] is separate from [`RSI`, 4]. More details can be found in [1].

4 Use Cases of Formal Symbolic Execution

As conclusion, we discuss some use cases of formal symbolic execution.

Combine with CFG extraction As discussed, we do symbolic execution per basic block. The CFG dictates how these basic blocks are tied together. We aim to combine a formally proven correct CFG extraction tool with our symbolic execution engine to get a summarized – but correct – representation of the binary in the theorem prover.

Formal Proofs of Memory Usage In [1], we use the symbolic execution engine to reason over the memory read from and written to by functions in binaries. We generate Floyd invariants, that allow reasoning per basic block to be used to reason over a function as a whole. We have applied this to 71 functions of the binary of HermitCore, and dealt with functions with loops, recursion, and pointer arguments. The methodology requires interactive theorem proving, and we aim to make this methodology more automatable to achieve better scalability.

Formal Proofs of Soundness of Randomizers Binary randomization is a technique used to prevent return-oriented-programming attacks. A randomizer rewrites basic blocks to eliminate so-called gadgets, i.e., byte-sequences that can be interpreted as a `ret` instruction. Analysis of these randomizers typically focuses on security properties, and less on soundness. Using formal symbolic execution, we intend to compare the semantics of a binary with the semantics of its randomized version, and thereby prove soundness.

References

1 Joshua A. Bockenek, Freek Verbeek, Peter Lammich, and Binoy Ravindran. Formal verification of memory preservation of x86-64 binaries. In *International Conference on Computer Safety, Reliability and Security (SAFECOMP'19)*, 2019. To be published.

2 Eric Cheng. *Binary Analysis and Symbolic Execution with angr*. PhD thesis, The MITRE Corporation, 2016.

3 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

4 Stefan Heule, Eric Schkufza, Rahul Sharma, and Alex Aiken. Stratified synthesis: automatically learning the x86-64 instruction set. In *ACM SIGPLAN Notices*, volume 51, pages 237–250. ACM, 2016.

5 Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

6 Ian Roessle, Freek Verbeek, and Binoy Ravindran. Formally verified big step semantics out of x86-64 binaries. In *Proceedings of CPP'19*, pages 181–195. ACM, 2019.

7 Teodor Stoenescu, Alin Stefanescu, Sorina Predut, and Florentin Ipate. River: A binary analysis framework using symbolic execution and reversible x86 instructions. In *International Symposium on Formal Methods*, pages 779–785. Springer, 2016.