Tripti Agarwal
University of Utah

Symbolic Formal Differencing of RISC-V programs

tripti.agarwal@utah.edu

November 30, 2022

Abstract: Tripti Agarwal is a second-year Ph.D. student who is dedicated to the Symbolic Formal Verification of RISC-V programs. This work is an extension of CASM-Verify [1] which can perform symbolic verification of x86 programs. We aim at verifying RISC-V open standard Instruction Set Architecture (ISA) using a similar technique as mentioned in CASM-Verify [1]. The proposed work will extend to cover safety-critical programs. Previously she has worked on lossy data compression algorithm [2] [2] and produced computational results of compressed datasets and decompressed datasets for different compiler optimization flags [3].

1 Problem Statement

Formal equivalence verification of programs is a topic of growing importance. Consider an x86 assembly program P1 and its optimized variant P2 generated by a compiler under development: it will be important to compare the behaviors of P1 and P2 for all possible inputs. Formal methods based on SMT offer a way to solve such problems symbolically and prove their satisfiability.
In this work, we will describe how we have analyzed the design and implementation of an existing tool for symbolically verifying the equivalence of highly optimized x86 implementation of cryptographic algorithms (see section 2). As a way to deepen our understanding and address new opportunities in an emerging context of high interest, we describe how we have ported this tool to handle RISC-V instructions (see section 3). Finally, in Section 4, we describe our ongoing work which consists of two tasks:

- Work on scaling our RISC-V verifier called SymDiff_RISC-V
- Investigate adaptations of our tool for verifying formal equivalence problems in domains such as PLC controllers [4] that are ubiquitous and of critical importance to safety-critical embedded and IoT systems [3].

2 Background Research

My work began with understanding CASM-Verify, a formal equivalence verification system of highly optimized x86 cryptography algorithms.

- The tool generates random inputs which get executed on two implementations of x86 codes and find out the variables that are likely to be equivalent. It then verifies whether the likely equivalent nodes are actually equivalent or not by using an SMT solver.
- It also decomposes the original query into small smaller sub-queries by using a collection of optimization for memory accesses.

The high sketch of the tool is verifying the equivalence of DSL (domain-specific language) with an x86 assembly code and x86 assembly code with an optimized version of the x86 assembly code. In both cases, the user provides precondition and postcondition that relates input and output variables respectively. The implementation of the tool is shown in the flow graph below:

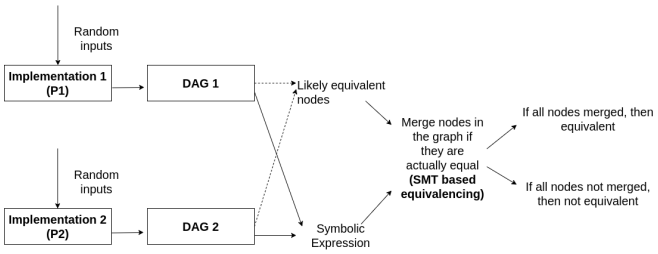

Figure 1: CASM-Verfiy Flow chart: The chart shows how two implementations in x86 are verified in CASM-Verify

3 SymDiff_RISC-V

RISC is an open chip architecture, which is free to license. Customers can add their own extensions and customize the chip for a number of applications that include artificial intelligence, and mobile and industrial applications. RISC-V is an open architecture that is gaining widespread adoption. It is designed as a modular instruction set with a very small base of fewer than 50 instructions. RISC-V's non-technical advantages include an open standard model that encourages both competition and collaboration, and which ensures long-term stability to protect investment in the software ecosystem. Based on the advantages of RISC-V we focused on developing a tool (SymDiff_RISC_V) that can perform symbolic difference verification of RISC-V programs.

4 Ongoing Work

SymDiff_RISC_V tool [5] is built on top of CASM-Verify which can perform verification of optimized RISC-V programs to DSL programs or other RISC-V programs, given the precondition and postcondition. The ISA for RISC-V consists of both 32-bit and 64-bit instructions. We have

[1] https://github.com/rutgers-apl/CASM-Verify
[2] https://github.com/mmartel66/blaz
[3] https://github.com/damtharvey/pyblaz
[4] https://en.wikipedia.org/wiki/Programmable_logic_controller
[5] https://github.com/tripti-agarwal/RISC_V-Symbolic-Verification

currently developed 32-bit instructions and the code can easily be extended to a 64-bit instruction set. RISC-V has 32 integer registers in which the first register (x0) is a zero register and the rest are general purpose registers. Our current code can handle these registers and then produce results based on the opcode provided. As of now, we have implemented 20, 32-bit integer opcodes, which include basic arithmetic, logical, and branch opcodes. These include register and immediate value instructions. The instructions that can handle memory registers (to handle arrays) are yet to be implemented. The current implementation can handle binary and unary instructions, and we can easily extend the code for ternary operations. We further plan to extend our implementation so that we can perform formal equivalences of the problems in the domain of PLC controllers.

4.1 Example

This is a small example of verification of RISC-V code compared with an equivalent DSL, along with preconditions and postconditions.

P1	P2	Precondition

d = (a >>> 13) ^ (a >>> 9);
e = (a & b) ^ (a & c);
f = d + e;
	srli x4, x1, $4
	xor x5, x4, x1
	srli x6, x5, $9
	xor x7, x2, x3
	and x8, x1, x7
	add x9, x6, x8
		P1.a == P2.x1;
		P1.b == P2.x2;
		P1.c == P2.x3;
		Postcondition
		P1.f == P2.x9;

Figure 2: Simple example of DSL and RISC-V code with preconditions and postconditions.

Query Decomposition

```
[P2TempName.29 = = P2TempName.16 + P2TempName.25,
P2TempName.25 ==P2.x1.0 & P2TempName.21,
P2TempName.21 == P2.x2.0 ^ P2.x3.0,
P2TempName.16 == RotateRight(P2TempName.12, 9),
P2TempName.12 == P2TempName.7 ^ P2.x1.0,
P2TempName.7 == RotateRight(P2.x1.0, 4),
P1TempName.6 == P1TempName.2 + P1TempName.5,
P1TempName.5 == P1TempName.3 ^ P1TempName.4,
P1TempName.4 == P1.a.0 & P1.c.0,
P1TempName.3 == P1.a.0 & P1.b.0,
P1TempName.2 == P1TempName.0 ^ P1TempName.1,
P1TempName.1 == RotateRight(P1.a.0, 9),
P1TempName.0 == RotateRight(P1.a.0, 13)]
```

Nodes that are found to be equivalent

```
P2TempName.29 , P1TempName.6
P2TempName.25 , P1TempName.5
P2.x3.0 , P1.c.0
P2.x2.0 , P1.b.0
P2TempName.16 , P1TempName.2
P2.x1.0 , P1.a.0
```

Figure 3: The query decomposition, along with the temporary name given to nodes in DAG and list of nodes found to be equivalent after applying SMT-based equivalence check.

We can also perform verification of RISC-V code with another optimized RISC-V code, similar to the way as shown in 2, with P1 also being a RISC-V code.

References

[1] LIM, J. P., AND NAGARAKATTE, S. Automatic equivalence checking for assembly implementations of cryptography libraries. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2019), pp. 37–49.

[2] MARTEL, M. Compressed matrix computations. *CoRR abs/2202.13007* (2022).

[3] PERNSTEINER, S., LONCARIC, C., TORLAK, E., TATLOCK, Z., WANG, X., ERNST, M. D., AND JACKY, J. Investigating safety of a radiotherapy machine using system models with pluggable checkers. In *Computer Aided Verification* (Cham, 2016), S. Chaudhuri and A. Farzan, Eds., Springer International Publishing, pp. 23–41.