# Selfie: Towards Minimal Symbolic Execution

Alireza S. Abyaneh
Simon Bauer
Christoph M. Kirsch
Philipp Mayer
Christian Mösl
Clément Poncelet
Sara Seidl
Ana Sokolova
Manuel Widmoser
Department of Computer Sciences
University of Salzburg
Austria
firstname.lastname@cs.uni-salzburg.at

## ABSTRACT

Selfie[1] is a fully self-contained 64-bit implementation of (1) a self-compiling compiler written in a tiny subset of C called C* targeting a tiny subset of 64-bit RISC-V called RISC-U, (2) a self-executing RISC-U emulator, and (3) a self-hosting hypervisor that virtualizes the emulated RISC-U machine. Selfie is implemented in a single 10k-line file and can compile, execute, and virtualize itself any number of times in a single invocation of the system given adequate resources. There is also a simple in-memory linker, a RISC-U disassembler and debugger with replay, and a profiler. Selfie has originally been developed just for educational purposes but has recently become a research platform as well. C* supports only two data types, `uint64_t` and `uint64_t*`, and RISC-U features just 14 instructions, in particular for unsigned arithmetic only, which significantly simplifies reasoning about correctness. In this paper, we report on an ongoing effort in designing and implementing a symbolic execution engine for RISC-U within selfie that is supposed to explore non-trivial parts of the system including itself. The idea is to identify a minimal set of ingredients that are necessary to do this such as the data structures for representing traces, path conditions, and symbolic states as well as algorithms for SAT and SMT solving. The key difference to related projects is that we are interested in reasoning just about selfie, for now, and are able to change selfie if necessary, as reasoning target but also as integrated platform for compilation, (symbolic) execution, and virtualization. Since selfie generates unoptimized code we are also exploring ways to leverage our symbolic execution engine in RISC-U code optimization.

## CCS CONCEPTS

• **Applied computing** → **Education**; • **Software and its engineering** → *Compilers*; *Interpreters*; *Virtual machines*;

[1]http://selfie.cs.uni-salzburg.at

## KEYWORDS

Symbolic Execution, RISC-V, Self-Referentiality

**ACM Reference Format:**
Alireza S. Abyaneh, Simon Bauer, Christoph M. Kirsch, Philipp Mayer, Christian Mösl, Clément Poncelet, Sara Seidl, Ana Sokolova, and Manuel Widmoser. 2018. Selfie: Towards Minimal Symbolic Execution. In *Workshop on Modern Language Runtimes, Ecosystems, and VMs (MoreVMs'18).*

## 1 PROBLEM

We are interested in identifying a minimal set of ingredients necessary for performing fast symbolic execution of systems code. Our target is selfie, a *minimal* but still *realistic* implementation of a compiler [7], emulator, and hypervisor [6]. The key observation is that systems code may not require many reasoning capabilities usually supported by state-of-the-art symbolic execution engines. There are at least two sides to our project, creating an adequate platform for symbolic execution (next section) and developing the actual execution engine (last section).

## 2 INFRASTRUCTURE

The original version of selfie was 32-bit, used signed integers and pointers to signed integers, and targeted a MIPS subset with up to 64MB of memory per machine instance [4]. C* even in that version has already been shown to support efficient implementations of non-trivial C programs such as the C* port of a state-of-the-art SAT solver written in C [1]. Yet to facilitate symbolic execution, we decided to port selfie to 64-bit, replace signed integers with unsigned integers, support RISC-V rather than MIPS, and increase memory per machine instance to 4GB. All that happened in three steps. First we went from `int` and `int*` to `uint64_t` and `uint64_t*`, and from MIPS32 to MIPS64. The advantage is twofold. Unsigned integers are significantly easier to reason about, and actually lead to simpler and cleaner code, and 64-bit helps with scaling up memory. We took advantage of that in the second step going up from 64MB to 4GB. This can also be done with 32-bit of course but, if necessary, we are now easily able to go up even further. The third step was to move from MIPS to RISC-V. While RISC-V instruction encoding and decoding is more difficult, everything else is simpler. In particular, the machine state is smaller (no hi and lo registers) and there are fewer instructions, just 14, down from 17 instructions. Moreover, all RISC-U instructions but system calls may only have a side effect (other than on the program counter) on at most one 64-bit machine word in either a register or in memory. The five system calls necessary for bootstrapping selfie (`malloc`, `exit`, `open`, `read`, and `write`) only required minor modifications.

The current version of selfie[2] generates proper 64-bit RISC-V ELF binaries that are compatible with the official RISC-V toolchain. However, only an earlier 32-bit RISC-V prototype of selfie actually executes on the official RISC-V spike emulator and pk kernel[3]. We are working on getting this done for the current 64-bit version which mostly requires interfacing memory allocation properly.

Other ongoing work on infrastructural level is multicore support. We are exploring simple ways to have selfie fork itself to run in parallel on multicore hardware. This would allow us to execute RISC-U code symbolically in parallel.

## 3 ENGINE

Inspired by the success of symbolic execution engines [2, 3], we have recently started developing a symbolic execution engine for RISC-U within selfie. Here, the difference to regular concrete execution of RISC-U code, as performed by the emulator in selfie, is that data read from a file is assumed to be symbolic. For example, if symbolically executed code reads one byte from a file the value of that byte is kept unspecified such that it could be any value between 0 and 255. Any subsequent computation involving that symbolic value needs to keep track of the concrete values it still represents. Arithmetic and logical operations and in particular comparisons change or even constrain the choice of concrete values possibly to the point that no concrete value remains at which point the engine backtracks. The information which values still remain on a path through the control-flow graph is represented by a so-called path condition. As long as a path condition is satisfiable there are concrete values that will drive the code down that path.

Another reason for backtracking is lack of memory in which case the engine is unable to explore the control-flow graph any further. In other words, symbolic execution attempts to execute code for all possible inputs up to the available memory and CPU time. If a runtime error such as division by zero is detected the engine may output a witness of that bug, that is, a file that will, when used as input, drive the code into that bug.

We are currently able to execute RISC-U code symbolically up to a fixed depth into the control-flow graph. We are also able to check satisfiability of path conditions, that is, whether code is actually reachable for some input, but only for a subset of all possible RISC-U programs. The restriction is due to the simplicity of the checker which is based on integer interval constraints [5]. However, the engine is able to detect if it is executing code within the limitations of the checker. Preliminary results show that at least the selfie compiler is within the supported fragment.

In the current version of selfie, only data read from a file by a `read` system call can be symbolic. We adapted the `read` implementation in selfie accordingly. Besides `read`, selfie uses and implements three other system calls, namely `open`, `write`, and `exit`, which we have not modified. Also, `malloc` works as before using a simple bump pointer and there is of course no `free` in selfie.

The key data structure for symbolic execution in selfie is a trace of machine state transitions. Essentially, we record for each executed instruction the current program counter value, the concrete or symbolic side effect of the instruction, and the state transition containing the previous concrete or symbolic value of the affected register or memory location. A trace therefore contains the whole execution history.

An interesting prerequiste of symbolic execution in selfie is replay. We noticed that recording the last, say, hundred instructions during regular concrete execution is a special case of symbolic execution. In the current version of selfie, the system is able to do that and, upon exceptions such as division by zero, undo the side effects of the last hundred instructions and then redo them but this time printing assembly code (with approximate source code line numbers) and its side effects on the console. Backtracking during symbolic execution corresponds to the undo operations during replay. In other words, there are now four implementations of each RISC-U instruction in selfie: concrete execution, concrete undo (inverse execution), symbolic execution, and symbolic backtracking.

We are now working on generating witnesses of bugs, that is, files that will drive RISC-U code into these bugs. The idea is to generate witnesses during symbolic execution of selfie and then provide the witnesses as input to selfie during concrete executions for validation, all in a single invocation of the system.

Another line of work focuses on simple ways for terminating loops during symbolic execution through sufficiently strong invariants provided by us. Also, we will eventually integrate symbolic execution with selfie's fork in order to explore independent code paths in parallel on multicore machines.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This work has been supported by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund (FWF) Grant S11404-N23 and S11411-N23), a Google PhD Fellowship, and a Google Research Grant.

## REFERENCES

[1] A.S. Abyaneh and C.M. Kirsch. 2017. You can program what you want but you cannot compute what you want. In *Edward A. Lee Festschrift (LNCS)*. Springer.
[2] C. Cadar, D. Dunbar, and D. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proc. USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 209–224.
[3] P. Godefroid, M. Y. Levin, and D. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. Symposium on Network and Distributed Systems Security (NDSS)*. 151–166.
[4] C.M. Kirsch. 2017. Selfie and the Basics. In *Proc. ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM.
[5] Eric Larson and Todd Austin. 2003. High Coverage Detection of Input-related Security Faults. In *Proc. 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 9–9.
[6] J. Liedtke. 1996. Toward Real Microkernels. *Commun. ACM* 39, 9 (Sept. 1996), 70–77. https://doi.org/10.1145/234215.234473
[7] Niklaus Wirth. 1996. *Compiler Construction*. Addison Wesley.

[2]https://github.com/cksystemsteaching/selfie
[3]https://riscv.org