Symbolically executing emulators

Daniel J. Bernstein[1,2]

[1] Department of Computer Science, University of Illinois at Chicago, USA
[2] Institute of Information Science, Academia Sinica, Taiwan
djb@cr.yp.to

Abstract. Symbolic execution of machine language has a wide range of applications, including equivalence verification applied directly to binaries that users run. However, there are limits on the instruction sets supported by existing symbolic-execution tools such as `angr`.

This paper reports experiments showing that it is sometimes affordable to carry out symbolic execution of an instruction set by applying a symbolic-execution tool for another instruction set to an emulator for the first instruction set. In particular, this paper reports verifying `sparc32` object code for all 248 functions in the latest version of `cryptoint` (including 76 functions that have `sparc32` assembly implementations), by using the `angr` toolkit to symbolically execute an `amd64` binary that uses the `unicorn` toolkit to emulate `sparc32` instructions. This paper also reports proof-of-concept experiments using symbolic execution to automatically extract partial instruction semantics from an emulator.

Keywords: machine language, symbolic execution, equivalence verification

1 Introduction

A computer executing a program follows one instruction after another inside the program. The computer's state is a sequence of bits modified by the instructions. For example, if the first bit is a 1, and the second bit is also a 1, then XORing the second bit into the first bit will change the first bit to 0.

Symbolic execution also follows one instruction after another, but applies the instructions to a more complicated machine state in which bits are replaced with *symbolic bits*. A symbolic bit is allowed to be not just 0 or 1 but also a more general *formula* in terms of specified variables. For example, if the first symbolic bit is the formula `x&y` (in C notation), and the second symbolic bit is the formula

Permanent ID of this document: `176cb11d2d75435bd573694a62a8eca5b497bf53`. Date: 2026-02-01. This work was funded by the Intel Crypto Frontiers Research Center, and by the Taiwan's Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP). "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s)."

`z`, then XORing the second symbolic bit into the first symbolic bit will change the first symbolic bit to the formula `(x&y)^z`. Typically the user chooses some inputs to replace with formulas, and then a symbolic-execution tool automatically traces through the formulas produced by the program starting from those inputs—in much the same way that a human sometimes writes down formulas for each result computed in the program, but one hopes that a symbolic-execution tool will be faster and less error-prone than a human.

Some tools for symbolic execution (e.g., SymCC [43] and SymQEMU [44]) are limited to *concolic execution*. This means that the instruction pointer is a concrete number at each moment (each bit being 0 or 1), while other parts of the machine state can store more complicated formulas.[3] The traditional form of program execution, where each bit is 0 or 1, is called *concrete execution*.

This paper will say more later about a program-analysis tool called `angr`, which was introduced in [51] and has received many subsequent updates. This tool isn't limited to concrete execution, or even to concolic execution: if a branch condition is more complicated than just "true" or "false" then `angr` creates a symbolic instruction pointer. Internally, `angr` splits symbolic execution into two universes, one for each direction of the branch, and then continues with each universe, so the instruction pointer within each universe is concrete.[4] One can see the full symbolic instruction pointer by asking `angr` for the branch conditions that define each universe.

The literature explains many applications that take advantage of the extra flexibility of symbolic execution compared to concrete execution. As one example that motivated this paper, Section 2 reviews an application of symbolic execution to equivalence verification, checking that two code snippets compute the same output for each possible input.

1.1 Extending instruction sets

What happens if one wants to symbolically execute a type of program not supported by existing symbolic-execution tools? Concretely, `angr` understands how to execute many different machine instructions for today's most popular CPUs, but what happens when a program uses an instruction-set extension beyond what `angr` supports, or has been compiled for a different CPU?

[3] This description assumes that there is no data flow from symbolic bits to branch conditions. More generally, concolic execution replaces each bit with a pair (b, f) where b is a bit and f is a symbolic bit, and uses b to control branches, so the symbolic bits f provide formulas for the behavior of the program *for all inputs that produce the same control flow as the bits* b. The name "concolic" was introduced in [50] as a portmanteau of "concrete" for b and "symbolic" for f. Software automating this type of execution had appeared in [25] without the name "concolic".

[4] Further symbolic branches then split the universes further. The literature often claims that there is an exponential explosion of the number of universes as the number of branches increases. However, it is possible to merge universes that have the same instruction pointer, so one can limit the number of universes to the number of reachable instructions or, better, to the number of reachable basic blocks.

The conventional answer is to add support for further instructions to the symbolic-execution tool. This is easier said than done, especially if one is concerned with accuracy. There are more than 1000 different instructions on Intel's current CPUs (see, e.g., [16]), with tens of thousands of details that one might get wrong (see, e.g., the discrepancies detected in [19]), not to mention all the other CPUs of interest. Fortunately, there is another approach, as we'll see in a moment.

1.2 Contributions of this paper

This paper reports successful equivalence-verification experiments using `angr` to symbolically execute various binaries compiled for `sparc32`, which is a popular architecture for CPUs used today in space applications[5] but not an instruction set supported by `angr`. In particular, these experiments verified equivalence between reference implementations and compiled `sparc32` binaries for all 248 functions in the `cryptoint` library described in [9]; this is an almost-header-only C library that I introduced in 2024 for carrying out various basic integer operations in constant time.

These experiments do *not* involve any new code to interpret `sparc32` instructions. In particular, these experiments do *not* add `sparc32` patches to `angr` or to any of the CPU-support libraries used inside `angr`. Instead these experiments

- take an emulator (compiled for `amd64`) that simulates a `sparc32` CPU running the binaries, and then
- use `angr` to symbolically execute the emulator.

Wait, doesn't simulation of a `sparc32` CPU need to understand the `sparc32` instruction set? Yes, it does, but that isn't *new* code: the popular `qemu` emulator introduced in [7] already includes support for many architectures, including `sparc32`.

Structurally, it's obvious that one can obtain "symbolically execute platform S" by composing "symbolically execute platform A" with "use A to emulate S", as long as one can afford to pile the symbolic-execution slowdowns on top of the emulation slowdowns. In particular, `angr` advertises support for `amd64`, and `qemu` running on those CPUs simulates other CPUs.

On the other hand, a platform is more than a CPU, and running `qemu` under `angr` turns out to encounter a series of mismatches between what `qemu` relies upon and what `angr` provides. Some mismatches are easy to work around; Section 3 handles the others by building a small replacement emulator that is adequate for this paper's applications to equivalence verification. This replacement emulator still does not require new code to interpret `sparc32`

[5] See, e.g., [36], [17], and [42], all of which use the radiation-hardened LEON3-FT CPU. This CPU is part of a series of CPUs that, as explained in [23], selected the SPARC architecture as an established *non-proprietary* architecture. RISC-V is also non-proprietary and perhaps will eventually replace SPARC; see, e.g., [52].

instructions: it uses the `unicorn` instruction-set simulator from [41], which copies the instruction-set handling from `qemu` and in particular supports `sparc32`. See Section 3.

Note that the fact that something is based upon `qemu` does not imply that it supports all of the instruction sets that `qemu` does. For example, the `qemu`-derived S2E symbolic-execution engine from [15] assumes a "guest OS that runs on x86 or ARM". But `unicorn` is more flexible and sufficed for the task at hand.

I have not found any previous success reports for symbolically executing emulators. Perhaps development was deterred by fear that the results would consume infeasible amounts of CPU time.[6] This paper's verification of `sparc32` object code for all 248 functions in the `cryptoint` library completed in under 6 hours wall-clock time on a single server described below. This was orders of magnitude slower than the same task for various architectures directly supported by `angr`, but was still affordable.

Beyond the usable tool from Section 3, Section 4 reports proof-of-concept experiments that—subject to various limitations—automatically extract instruction semantics from an emulator via symbolic execution of emulation of a single instruction. This extraction is a first step towards automatic compilation of emulators into symbolic-execution tools that will take less CPU time than the approach from Section 3. Section 4 also summarizes other potential applications of this extraction, such as automatic emulator verification.

2 Equivalence verification via symbolic execution

As a specific example of an application of symbolic execution, this section reviews how `saferewrite` uses symbolic execution via `angr` to verify that compiled versions of the `cryptoint` functions match reference implementations for all inputs. I introduced `saferewrite` in a talk [8] in 2021; see [11] for the current version, `saferewrite-20260201`. Almost all of the work in `saferewrite` is handled by `angr`, which as mentioned in Section 1 was introduced in [51]; `saferewrite` is a small wrapper around `angr`.

Section 2.1 reviews the general problem of equivalence verification. The `saferewrite` package includes an analysis tool and many examples of code rewrites; Section 2.2 uses one example to illustrate basic usage of `saferewrite`, and Section 2.3 uses the same example to illustrate equivalence verification. Section 2.4 explains how to add new examples. Section 2.5 explains how `saferewrite` has enough functionality and performance to handle all of the `cryptoint` functions—but Section 2.6 explains why, before the work explained in Section 3, `saferewrite` didn't support code compiled for SPARC.

[6] Imagine *Inception* inside *The Matrix*.

2.1 The danger of rewriting code

Suppose someone is rewriting a code snippet—perhaps to accelerate it, or to simplify it, or to make it more portable, or to make it less likely to trigger compiler bugs, or to avoid leaking secret data through timing. How do we make sure that the rewrite hasn't introduced any bugs?

Maybe someone started with the C code

```
int64_t if_positive_then_else(int64_t x,int64_t p,int64_t n) {
  if (x > 0) return p;
  return n;
}
```

and decided to rewrite it as

```
int64_t if_positive_then_else(int64_t x,int64_t p,int64_t n) {
  return n ^ ((n ^ p) & ((-x) >> 63));
}
```

to remove the conditional branch, making sure to compile with `gcc -fwrapv` so that `int64_t` arithmetic is fully defined (in particular, the signed right shift is then defined on negative inputs). This rewrite passes many random tests—but it still has a bug: namely, if `x` is -2^{63} then `-x` is also -2^{63} because `int64_t` arithmetic is modulo 2^{64}, so `(-x) >> 63` is -1, and the code ends up returning `p` instead of the desired `n`.

There are natural types of tests that will catch this particular bug: for example, trying random inputs with just a few bits set, or testing a generalization from `int64_t` to other sizes—hoping that any bugs in the `int64_t` code are also visible as bugs for, say, `int8_t`; there are only 2^{24} possible inputs to the `int8_t` version of this function (and almost 2^{16} of them will trigger this bug). But the bigger picture is that passing tests cannot guarantee that a rewrite is correct. Experience indicates that bugs apply to varying fractions of all inputs, sometimes caught by tests but sometimes not.

A reviewer can try to catch a bug in a rewrite by thinking through what the code does—but, hmmm, what if the reviewer makes the same mistake that the code author made? Perhaps more convincing is for the reviewer to write a proof of correctness—but, hmmm, does that really stop the reviewer from making a mistake? Even more convincing is a computer-checked proof (see generally [**10**]), but can we afford to scale this effort to many rewrites of many code snippets?

The `cryptoint` library mentioned above is the result of hundreds of rewrites of simple reference code into more complicated code snippets. This poses obvious correctness questions, which are addressed in Section 2.2.

2.2 Using `saferewrite`

The following text focuses on basic usage of `saferewrite` for `cmp_64xint16`, one of the rewrite examples included in the `saferewrite` package.

The src/cmp_64xint16/ref directory has one file, verify.c, which has the following contents modulo line breaks:

```
#include <stdint.h>
int cmp_64xint16(const uint16_t *x,const uint16_t *y) {
  for (int i = 0;i < 64;++i)
    if (x[i] != y[i]) return -1;
  return 0;
}
```

This is reference code for comparing two int16[64] arrays. There are then ten further src/cmp_64xint16/* directories that are (not necessarily correct) rewrites of the reference code. One of those directories, namely src/cmp_64xint16/openssl, has more files than verify.c: there is a memcmp.s straightforwardly derived from assembly in OpenSSL, and there is an architectures file saying amd64, which tells saferewrite to compile this rewrite only for that architecture. Also, src/cmp_64xint16/bitopscpp has verify.cc rather than verify.c.

All measurements in this paper are from a server named rome2, a dual AMD EPYC 7742 running Debian 12. The 128 CPU cores run at 2.245GHz; I disabled overclocking. Running chmod +t src/*; chmod -t src/cmp*; ./analyze on rome2 to analyze cmp_64xint16 with saferewrite completed in 116 seconds wall-clock time, using 4369 core-seconds of user time and 279 core-seconds of system time. The results of the analysis are in 131 directories build/*/*/*/analysis containing 874 files build/*/*/*/analysis/*. One of those files has name

```
build/cmp_64xint16/frodo2/gcc_-O3_-march_native_-mtune_native
/analysis
/unsafe-differentfrom-ref-gcc_-O3_-march_native_-mtune_native
```

(modulo line breaks). The contents of that file include an input for which src/cmp_64xint16/frodo2 compiled with gcc -O3 produces a different output from src/cmp_64xint16/ref compiled with gcc -O3.

The frodo2 code is from real cryptographic software that had a bug pointed out in [46]. What this example is showing is that saferewrite automatically catches this bug. This specific bug is not at all hard to detect—random unit tests would have reliably caught this bug if they had been applied to this function in the first place—but Sections 2.3 and 2.4 illustrate how saferewrite goes beyond random tests.

2.3 Equivalence verification

Another file resulting from the saferewrite run from Section 2.2 is an empty file

```
build/cmp_64xint16/bitopscpp/clang++_-O1_-fwrapv_-march_native
/analysis/equals-ref-gcc_-O3_-march_native_-mtune_native
```

whose name asserts that `src/cmp_64xint16/bitopscpp` compiled with `clang++ -O1 -fwrapv` produces the same outputs *for all possible inputs* as `src/cmp_64xint16/ref` compiled with `gcc -O3`.

The justification for this assertion relies on symbolic execution. Internally, `saferewrite` uses `angr` to symbolically execute the compiled binaries, in effect unrolling the binaries into formulas; `saferewrite` then uses an SMT solver (namely Z3 from [38], via a wrapper provided by `angr`) to show that the resulting formulas are equal for all inputs.

The compilers used for the analysis are listed in `./compilers`, currently listing 13 C compilers (where this analyses covers 12; see Section 3 for how to enable the 13th) and 12 C++ compilers. These include various cross-compilers (installed as part of the `saferewrite` installation). Also, as

```
build/cmp_64xint16/ref
/aarch64-linux-gnu-gcc_-O3_
-march_armv8-a_-mtune_cortex-a53_-mgeneral-regs-only
/analysis/equals-ref-gcc_-O3_-march_native_-mtune_native
```

illustrates, the analysis checks equivalence of code compiled for one architecture against code compiled for another architecture, perhaps catching compiler bugs or portability issues that might not be caught by single-architecture tests.

2.4 Adding rewrites

Extending `saferewrite` to test another rewrite is straightforward. For example, here is how to test the `if_positive_then_else` rewrite from Section 2.1:

- Create directories `src/ifpos`, `src/ifpos/ref`, and `src/ifpos/bad`.
- Copy the two snippets from Section 2.1 to `src/ifpos/ref/whatever.c` and `src/ifpos/bad/whatever.c` respectively.
- Add `#include <stdint.h>` at the top of each file to define `int64_t`.
- To tell `saferewrite` what the inputs and outputs are, create a file `src/ifpos/api` with the following lines:

```
return int64 r
in int64 x
in int64 p
in int64 n
call if_positive_then_else
```

- Run `chmod +t src/*; chmod -t src/ifpos; ./analyze` to analyze these `src/ifpos` rewrites.

This `ifpos` analysis is faster than the `cmp_64xint16` analysis from Section 2.2: in 9 seconds wall-clock time (58 core-seconds user time, 48 core-seconds system time) on `rome2`, this analysis produced 6 `unsafe-differentfrom` files, each showing `in_x_0 = 9223372036854775808 = 0x8000000000000000` along with some choices of `p` and `n`. To me, seeing an SMT solver find this example says

that the SMT solver is doing something useful, whereas merely seeing an SMT solver say "yes, equal" is less convincing.

Why are there are only 6 `unsafe-differentfrom` files when there are 12 compilers? Answer: The other 6 compilers use `gcc -O3` for various architectures. As discussed in [9, Section 4.8], `gcc` starting in 2021 includes an "optimization" that, when `-fwrapv` is not set, replaces `(-x)>>63` with `-(x>0)`. For `ifpos/bad`, this change produces compiled code that always matches `ifpos/ref`, and `saferewrite` correctly reports `equals-ref` for the compiled code.

Adding another rewrite `src/ifpos/good/fixed.c` with

```
#include <stdint.h>
int64_t if_positive_then_else(int64_t x,int64_t p,int64_t n) {
  int64_t y = -x;
  return n ^ ((n ^ p) & ((y ^ (x & y)) >> 63));
}
```

and re-running `./analyze` produces, as expected, 12 files with names of the form `build/ifpos/good/*/analysis/equals-ref-*`. Beware that there is still a risk of problems with other compiler options or with future compilers; see [9] for how I recommend writing this type of code.

2.5 Equivalence verification for each `cryptoint` function

Some other symbolic-execution tools analyze higher-level languages than binaries. However, analyzing binaries has the obvious advantage of being able to handle code written in assembly—such as the inline assembly in `cryptoint`—without worrying about whether that code is expressible in some higher-level language. Furthermore, analyzing binaries can catch problems in translations from other languages to binaries, whether the problems are indisputable compiler bugs or merely what one might call surprises. Conventional unit tests have the same feature of testing binaries, but, as the examples from Sections 2.3 and 2.4 illustrate, conventional tests are missing the SMT solver's ability to consider all possible inputs.

SMT solvers promise that whatever answers they give are correct. However, they do not guarantee that they will give answers. For slightly more complicated examples, SMT solving does not complete in a reasonable time. On the other hand, `saferewrite` includes many examples where SMT solving does rapidly give a "yes, this always matches" or "no, it doesn't always match" answer. In particular, `saferewrite` gives `equals-ref` answers for the `cryptoint` implementations of all 248 `cryptoint` functions.

In more detail: One run with 64 threads on `rome2` analyzed all 248 `cryptoint` functions in wall-clock time 606 seconds, using 21884 core-seconds user time and 6898 core-seconds system time. This run covered 7668 implementation-compiler combinations: for each of the 248 functions, the `saferewrite` package includes reference code, the `cryptoint` rewrite, and sometimes further rewrites (e.g., several rewrites of `int32_sort2`), so in total there are 639 implementations of

the 248 functions, times 12 compilers. The analysis cost was thus 3.8 core-seconds on average for each implementation-compiler combination. Memory consumption varied but was never observed to exceed 10GB in total for the 64 threads. Further notes on resource consumption appear in `README-resources` in the `saferewrite` package.

For 201 of the 7668 implementation-compiler combinations, `saferewrite`'s analysis includes `unsafe-unrollsplit` warnings. These indicate that there was a split of the analysis into multiple universes (so concolic execution would not have sufficed). The number of universes ranges from 2 on 123 occasions through 65 on 18 occasions. Typically the split is because of conditional branches in `ref`. The reason splits are marked as `unsafe` is that the timing of conditional branches often leaks secret data; this is one of the common reasons for rewriting simple reference code, and then `saferewrite` checks that the rewrite did not introduce bugs.

I don't recommend *abandoning* conventional tests in favor of `saferewrite`. It is conceivable that a bug in a rewrite will be hidden by a bug in an SMT solver, or by another bug in `angr`, or by a bug in the `saferewrite` code. But *supplementing* conventional tests with symbolic execution reduces risks.

2.6 The case of SPARC

Compiling and assembling C code into a binary involves architecture-specific code in compilers and assemblers: even when the original C code is portable, the target language is not. What `angr` is doing in symbolically executing a binary is similarly architecture-specific: the target language, essentially Z3 formulas, is portable, but the source language is not.

Internally, `angr` relies on (and is named by reference to) the VEX component of `valgrind`. VEX translates binaries into a somewhat simpler language. Normally `valgrind` executes instructions in that language; `angr` instead translates that language into Z3 formulas. Supporting an instruction set inside `saferewrite` thus requires support from VEX and support from `angr`.

Even for popular CPUs from Intel and AMD, this instruction support is not complete. For example, `valgrind` AVX-512 patches from [37] were not integrated into the official `valgrind` distribution, never mind the further work required for `angr` to support AVX-512. So it's unsurprising that a `valgrind` SPARC patch distributed by Oracle many years ago also wasn't added to `valgrind`.[7]

I have been adding assembly rewrites to `cryptoint` for reasons explained in [9, Section 6.3.1]. Equivalence testing via `saferewrite`, as in Section 2.3, is an important part of the assurance mechanisms described in [9, Section 6.4]. So, when I added `sparc32` assembly for a current radiation-hardened space CPU

[7] I exchanged email in February 2025 with the author of the `valgrind` SPARC patch; he indicated that he no longer had the source code. In June 2025, after the results of this paper for `sparc32` were completed and posted, a copy of the patch appeared in [22], along with comments about how difficult it would be to adapt the patch to the current version of `valgrind`.

to `cryptoint`, I was faced with the real-world problem of how to symbolically execute `sparc32` binaries.

One possibility is to write new patches for `valgrind` and `angr`, but this sounds error-prone, even for an instruction set as small as the SPARC instruction set. The point of Section 3 is a different approach that, as emphasized in Section 1, doesn't require any new code to interpret SPARC instructions.

3 Symbolic execution of emulation of a program

The current version of `saferewrite` includes an option to compile and analyze `sparc32` binaries, despite `angr` not supporting `sparc32`. Internally, what `saferewrite` is doing for `sparc32` is symbolic execution using `angr` of an `amd64` binary that emulates a `sparc32` binary. My original plan was for the `amd64` binary to be `qemu-sparc`, but, as noted in Section 1, I ended up building a replacement emulator on top of `unicorn`. The rest of this section explains various issues that I encountered, and how I worked around those issues.

3.1 The platform for a binary

When the operating-system kernel runs a binary, it allocates the right amount of RAM for the binary, copies the binary from disk into RAM, and then jumps to the entry point of the binary, at which point the CPU starts executing instructions from the binary. One complication is that binaries are usually dynamically linked; there is then initial code that (1) allocates further RAM for libraries and (2) links the libraries appropriately. Another complication is system calls: trap instructions that pass requests such as `read` or `write` to the operating-system kernel. There are many different system calls, with semantics operating on a multifaceted process state: each process has not just RAM but also permissions, timers, file descriptors, and more.

A full-fledged emulator such as `qemu` (or `valgrind`, but `valgrind` is not useful in this section since it does not support SPARC) includes a large amount of code trying to simulate all aspects of the process state.[8] As an illustration of the costs, calling `qemu-x86_64-static` to run a statically linked program that simply calls `_exit(0)` takes more than 30 million instructions, according to `perf stat`. Running the program directly takes 21537 instructions.

Symbolic execution in `angr` is faced with an even tougher simulation job, given the extra complications of applying instructions to a symbolic process state. For example, `angr` simulates a filesystem containing symbolic data, and simulates `read` and `write` functions in a way that can handle symbolic data. The `angr` documentation does not claim completeness of the process simulation; it provides

[8] There is also a full-system mode of `qemu` that tries to simulate a complete computer, but what matters for this paper is `qemu`'s user mode.

a SimProcedure mechanism to extend the simulation with support for further functions as needed.

Unsurprisingly, running qemu under angr encounters qemu calling functions that angr does not support. I started on a cycle of looking at the first call that breaks, fixing that, and trying again, but I abandoned this when it became clear that the approach of Section 3.2 would involve less development time and less CPU time.

3.2 Using unicorn

The unicorn toolkit from [41] was forked from qemu in 2015. The toolkit provides a C library interface to the instruction emulator inside qemu. The toolkit removes qemu's support for loading binaries, for process state beyond RAM, etc.

I wrote a small emulator, elfulator, on top of unicorn. The current version of elfulator is included in saferewrite-20260201 and has in total 704 lines of code. A few hundred lines are for ELF parsing; a few hundred lines are for calling unicorn and handling traps from unicorn. This is far from a full-fledged emulator—for example, it supports only statically linked binaries, and only a few system calls from those binaries (see Section 3.3)—but it does what saferewrite needs.

For testability, I found it convenient to develop elfulator starting with amd64 and arm64 binaries, and continuing with arm32 and x86 as 32-bit platforms supported by more tools than sparc32; later I added sparc64. Each platform uses a few extra lines for each system call, plus some lines of generic platform support. All of this is in elfulator as experimental code, and is counted within the 704 lines mentioned above, but this paper's evaluation of affordability focuses on sparc32.

There are still some library calls from unicorn beyond what angr supports, but few enough that handling them wasn't a serious problem:

- I linked elfulator with simple assembly for setjmp, longjmp, sigsetjmp, and siglongjmp, tweaking assembly available from [33].
- I intercepted clock_gettime and gettimeofday with C functions in elfulator.c returning time 0.
- I patched unicorn to replace some calls to mmap and munmap with, respectively, malloc and nothing.
- In saferewrite's script that calls angr, I added SimProcedures to adequately simulate posix_memalign, and to pass sysconf and getpagesize and strerror through to the surrounding operating system.

The library calls depend somewhat on which CPU is being emulated. For example, unicorn's ARM emulation calls vasprintf; I ended up patching unicorn to eliminate those calls.

3.3 System calls

There are three obviously critical system calls that `elfulator` allows from the binary it is emulating:

- `read` for the emulated program to receive symbolic inputs;
- `write` for the emulated program to provide symbolic outputs; and
- `exit` for the emulated program to say that it's done.

In earlier versions of `saferewrite`, I communicated symbolic inputs and symbolic outputs by directly accessing RAM in the binary being run by `angr`. Functions are provided by `angr` to access the binary's symbol table and the corresponding RAM locations. However, composing this with a layer of emulation would trigger obvious complications, so I switched to `read` and `write`. The implementations of the system calls in `elfulator` have many limitations that are not a problem for `saferewrite`, such as assuming that `read` is from file descriptor 0.

A typical C library invokes more system calls for a variety of reasons not relevant to `saferewrite`. I instead compiled with one of the smallest available C libraries, namely `dietlibc`, which was introduced in [35]. For compilation with a current SPARC cross-compiler, I made a minor patch to `dietlibc`, namely replacing `glob` with `globl` in `sparc/memcmp.S`.

The `unicorn` toolkit provides an interface for callers to read and write CPU registers, but the list of registers is generally incomplete. On SPARC, system calls indicate success or failure via a register called `PSR`. I patched `unicorn` to add support for `PSR`, and to adjust the SPARC instruction pointer appropriately after trap instructions. In total the patch to `unicorn` is 177 lines, including 22 lines added to `unicorn`'s SPARC handling.

3.4 Symbolic-execution speed

There are some options built into `angr` to save time in symbolic execution. Perhaps the most important is `angr.options.unicorn`, not to be confused with the usage of `unicorn` in Section 3.2. What `angr.options.unicorn` does is have `angr` call `unicorn` to run blocks of code *if* the relevant program state is concrete, rather than resorting to `angr`'s Python-level simulation of each instruction.

In the context of `saferewrite`, most of the `elfulator` execution is concrete: reading the binary to be tested, setting up `unicorn`, etc. Symbolic data first appears inside `elfulator` when the emulated binary that was cross-compiled by `saferewrite` calls `read`. During `elfulator` development, I killed one `angr` run after 60 hours where `angr.options.unicorn` reduced the time to 10 minutes.

I ran into some `angr` crashes with `angr.options.unicorn`, but did not encounter any crashes after I took the following two steps in `saferewrite`: first, disable the `angr.options.UNICORN_SYM_REGS_SUPPORT` component of `angr.options.unicorn`; second, fully disable `angr.options.unicorn` after any program step that reads file descriptor 0.

I also tried replacing `python3` with `pypy3`. This reduced CPU time by about 2× while increasing RAM usage by about 1.5×. However, I encountered occasional hangs of `pypy3`. (Running `gdb` on `pypy3` shows that the hangs were in `__futex_abstimed_wait_common64`.) Currently `saferewrite` does not know how to recognize the hang and restart the process.

A different possibility for gaining speed would be to run `elfulator` outside `angr`, dumping core after precomputations, and then load the core dump into `angr` for symbolic execution; or similarly dump the `angr` state at that moment.

The code inside `unicorn` to emulate any particular CPU instruction is being symbolically executed every time the instruction appears in the instruction stream for any of the programs being emulated. It would be faster to use symbolic execution just once for each instruction to extract the semantics of the instruction set, and then compile those semantics into a symbolic-execution tool that no longer incurs any of the overhead of an extra layer of emulation. See Section 4 for proof-of-concept experiments in this direction.

3.5 Results

Recall from Section 2.5 that a 64-thread experiment on `rome2` analyzing 12 compilers times 639 implementations of all 248 `cryptoint` functions took a few core-seconds per implementation-compiler combination, in total 606 seconds wall-clock time. After `sparc32` was added via `elfulator`, an experiment re-running `env THREADS=64 time ./analyze` took 314 minutes wall-clock time for 14717 core-minutes user time and 167 core-minutes system time, overall 14404 core-minutes more than the non-`elfulator` experiment. The 639 `elfulator` analyses thus added 22.5 core-minutes on average per implementation. Overall memory consumption on the server with all 64 threads running was never observed to pass 300GB. The maximum observed resident-set size for a single process was 8GB.

For some of the functions, there were problems compiling or unrolling `ref` for `sparc32`. For example, `int16_load/ref/load.c` relies on `le16toh`, which `dietlibc` does not support. However, the `cryptoint` implementation of each of the 248 `cryptoint` functions—including 76 functions where the `cryptoint` implementations are written in `sparc32` assembly—was successfully compiled for `sparc32`, unrolled via `elfulator`, and matched by SMT solving against `ref` for `amd64`, either directly or via an intermediate equality with `ref` for `sparc32`.

At that point I declared success: I released `cryptoint-20250228`, including the `sparc32` code, and `saferewrite-20250228`, including `elfulator`. I re-ran and re-released `saferewrite` for the subsequent `cryptoint-20250414` release. The speeds described in this paper are from further re-runs with `saferewrite-20260201`. The `saferewrite-20260201` package includes instructions for using `buildroot` to install a `sparc-linux-gcc` cross-compiler; compiling a patched `dietlibc` for `sparc32`; compiling a patched `unicorn`; and compiling `elfulator`. After these steps, each `./analyze` run automatically covers `sparc32`.

4 Symbolic execution of emulation of an instruction

Recall from Section 3.4 the possibility of using symbolic execution of an emulator to automatically extract the semantics of the CPU's instruction set. This has a variety of potential applications:

- Projects to verify the correctness of machine code such as [14], [40], [47], and [26] rely on specifications of the semantics of the relevant machine instructions. Official machine-readable instruction-set specifications are already available for some architectures, but one can imagine handling more architectures by automatically deriving specifications from emulators.
- [32], starting with the official machine-readable ARM instruction-set specification, automatically generated test cases for `qemu`, finding some bugs in `qemu`. One can imagine obtaining another specification of the same instruction set via symbolic execution of `qemu` or `unicorn`, and then verifying equivalence with the official instruction-set specification, as in [34, Section IV]; this would, similarly to Section 2, address concerns about bugs slipping past the test cases in [32].
- [28] and [29], starting with the official SPARC documentation, manually built a machine-readable instruction-set specification, and then tested it against a physical SPARC CPU. Again one can imagine verifying equivalence against another specification obtained via symbolic execution, and further verifying equivalence against a freely available SPARC HDL implementation.
- One can imagine using an instruction-set specification to automatically build a full suite of binary-analysis tools for that instruction set, including lifters as in [34] and [18], memory-error detectors such as `valgrind`, and the symbolic-execution engine inside `angr`. Presumably this would be easier and less error-prone than constructing similar tools by hand, and it would provide a speedup mentioned in Section 3.4: symbolic execution for that instruction set would no longer need to incur the overhead of symbolically executing an emulator.

The necessary information about the instruction set is already stated in computer-readable form inside the code for the emulator. The task at hand is to extract that information into an easier-to-use form.

Conceptually, it is clear how to begin: pick an instruction; symbolically execute the emulation of that instruction. This might sound like a straightforward special case of symbolically executing the emulation of a complete program. However, the inputs and outputs in Section 3 were short bit strings, whereas instruction semantics are normally expressed in terms of a larger, more complicated machine state with RAM, an instruction pointer, flags, general-purpose registers, and usually more types of registers such as vector registers.

Section 4.1 reports proof-of-concept experiments focusing on arithmetic instructions, using symbols for the contents of flags and general-purpose registers. One experiment takes as input a single 32-bit `sparc32` arithmetic

instruction, for example `0x82808003`, and extracts semantics for this instruction in under 20 minutes on one core of the machine mentioned earlier in this paper. These semantics are in a simple language, suitable for equivalence checking against other specifications of the same instruction. Note that 20 minutes are probably slower than testing 2^{32} inputs to an instruction but much faster than testing 2^{64} inputs.

The closest work that I am aware of is [27], which symbolically executed the `gcc` code generator to extract a mapping from `gcc`'s intermediate representation to `x86` instructions, and then inverted this mapping to guess semantics of the `x86` instructions. [27, Section 4] argues that these guesses are sufficient for analyzing binaries generated by compilers, despite usually leaving flags undefined. The experiments in Section 4.1 instead produce formulas showing how `unicorn` computes flags.

4.1 Experiments handling emulated register contents as variables

Running `./syminsn-sparc32 0x82808003` in the `saferewrite` directory (after `elfulator` installation) compiles an `amd64` program that does the following:

- Initialize `unicorn` for `sparc32`.
- Read 31 `int32` values (in little-endian form) from standard input, and use those values to initialize `unicorn`'s emulated `sparc32` registers `g1`, `g2`, etc.
- Read 4 bytes from standard input, and use the bottom bits of those bytes to initialize `unicorn`'s emulated `sparc32` flags `cf`, `vf`, `zf`, and `nf`.
- Read 4 more bytes from standard input, and run `unicorn` on those bytes viewed as an instruction (in big-endian form).
- Write the resulting registers and flags to standard output, in the same format as the input.

The `syminsn-sparc32` script then runs this program under `angr`, providing symbolic registers, symbolic flags, and a concrete instruction `0x82808003`.

The output of this experiment is Figure 4.1.1, which gives formulas (in `angr`'s Z3-like language—for example, `ULE` is an unsigned less-than-or-equal-to operation) for various output registers such as `out_g1` in terms of various input registers. These are formulas for the effect of `sparc32` instruction `0x82808003`, or at least for what `unicorn` thinks the effect is. Part of Figure 4.1.1 is setting `out_g1` to `__add__(in_g2,in_g3)`; inspecting other parts shows that, e.g., `out_cf` is the carry bit from that addition.

One can *manually* write down such formulas by studying the official SPARC documentation [30, page 108] for the "`ADDcc`" instruction. This type of manual work is what went into the `qemu` emulation code in the first place. Instead of redoing that work, this experiment reuses that work, extracting the self-contained Figure 4.1.1 as a description of the effect of this instruction. I tried similar experiments with several other arithmetic instructions, and checked that the results looked reasonable.

Fig. 4.1.1. Results of symbolic execution of `unicorn` emulating `sparc32` instruction `0x82808003` on 31 symbolic general-purpose registers and 4 symbolic flags.

```
v1 = in_g2
v2 = in_g3
v3 = __add__(v1,v2)
v4 = in_g4
v5 = in_g5
v6 = in_g6
v7 = in_g7
v8 = in_o0
v9 = in_o1
v10 = in_o2
v11 = in_o3
v12 = in_o4
v13 = in_o5
v14 = in_o6
v15 = in_o7
v16 = in_l0
v17 = in_l1
v18 = in_l2
v19 = in_l3
v20 = in_l4
v21 = in_l5
v22 = in_l6
v23 = in_l7
v24 = in_i0
v25 = in_i1
v26 = in_i2
v27 = in_i3
v28 = in_i4
v29 = in_i5
v30 = in_i6
v31 = in_i7
v32 = ULE(v1,v3)
v33 = constant(1,0)
v34 = constant(1,1)
v35 = If(v32,v33,v34)
v36 = Extract(v2,31,31)
v37 = Extract(v1,31,31)
v38 = __xor__(v36,v37)
v39 = Extract(v3,31,31)
v40 = __xor__(v39,v37)
v41 = __invert__(v40)
v42 = __or__(v38,v41)
v43 = __invert__(v42)
v44 = constant(32,4294967295)
v45 = __mul__(v44,v2)
v46 = __eq__(v1,v45)
v47 = If(v46,v34,v33)
v48 = If(v46,v33,v39)
out_g1 = v3
out_g2 = v1
out_g3 = v2
out_g4 = v4
out_g5 = v5
out_g6 = v6
out_g7 = v7
out_o0 = v8
out_o1 = v9
out_o2 = v10
out_o3 = v11
out_o4 = v12
out_o5 = v13
out_o6 = v14
out_o7 = v15
out_l0 = v16
out_l1 = v17
out_l2 = v18
out_l3 = v19
out_l4 = v20
out_l5 = v21
out_l6 = v22
out_l7 = v23
out_i0 = v24
out_i1 = v25
out_i2 = v26
out_i3 = v27
out_i4 = v28
out_i5 = v29
out_i6 = v30
out_i7 = v31
out_cf = v35
out_vf = v43
out_zf = v47
out_nf = v48
```

I also tried experiments handling multiple instructions at a time—for example, handling an immediate or a register index as a symbol—but encountered errors from `angr` that I didn't figure out how to work around. More work is also required for handling load/store instructions. So I'll leave it as an open question to cover a full instruction set.

References

[1] — (no editor), *Proceedings of the FREENIX track: 2005 USENIX annual technical conference, April 10–15, 2005, Anaheim, CA, USA*, USENIX, 2005. See [7].

[2] — (no editor), *IEEE symposium on security and privacy, SP 2016, San Jose, CA, USA, May 22–26, 2016*, IEEE Computer Society, 2016. ISBN 978-1-5090-0824-7. URL: https://ieeexplore.ieee.org/xpl/conhome/7528194/proceeding. See [51].

[3] — (no editor), *28th annual network and distributed system security symposium, NDSS 2021, virtually, February 21–25, 2021*, The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss2021/. See [44].

[4] — (no editor), *2023 IEEE space computing conference (SCC), Pasadena, CA, USA, 18–21 July 2023*, IEEE Computer Society, 2023. DOI: 10.1109/SCC57168.2023. See [36].

[5] — (no editor), *2024 IEEE aerospace conference, Big Sky, MT, USA, 02–09 March 2024*, IEEE Computer Society, 2024. DOI: 10.1109/AERO58975.2024. See [42].

[6] — (no editor), *31st IEEE international conference on electronics, circuits and systems (ICECS), 18–20 Nov. 2024*, IEEE Computer Society, 2024. DOI: 10.1109/ICECS61496.2024. See [52].

[7] Fabrice Bellard, "QEMU, a fast and portable dynamic translator". In: USENIX ATC FREENIX 2005 [**1**] (2005), 41–46. URL: https://www.usenix.org/events/usenix05/tech/freenix/bellard.html. Citations in this document: §1.2.

[8] Daniel J. Bernstein, "Fast verified post-quantum software" (2021), accessed 2026-02-01. URL: https://cr.yp.to/talks.html#2021.09.03. Citations in this document: §2.

[9] Daniel J. Bernstein, "The `cryptoint` library" (2025), accessed 2026-02-01. URL: https://cr.yp.to/papers.html#cryptoint. Citations in this document: §1.2, §2.4, §2.4, §2.6, §2.6.

[10] Daniel J. Bernstein, "Papers with computer-checked proofs" (2025), accessed 2026-02-01. URL: https://cr.yp.to/papers.html#pwccp. Citations in this document: §2.1.

[11] Daniel J. Bernstein, "saferewrite-20260201'' (2026), accessed 2026-02-01. URL: https://pqsrc.cr.yp.to/downloads.html. Citations in this document: §2.

[12] Tim Brecht, Carey Williamson (editors), *Proceedings of the 27th ACM symposium on operating systems principles, SOSP 2019, Huntsville, ON, Canada, October 27–30, 2019*, ACM, 2019. ISBN 978-1-4503-6873-5. URL: https://dl.acm.org/citation.cfm?id=3341301. See [40].

[13] Srdjan Capkun, Franziska Roesner (editors), *29th USENIX security symposium, USENIX Security 2020, August 12–14, 2020*, USENIX Association, 2020. ISBN 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20. See [43].

[14] Mario Carneiro, "Specifying verified x86 software from scratch" (2019). URL: https://arxiv.org/abs/1907.01283. Citations in this document: §4.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, George Candea, "The S2E platform: design, implementation, and applications". In: ACM Transactions on Computer Systems **30** (2012), 2:1–2:49. DOI: 10.1145/2110356.2110358. Citations in this document: §1.2.

[16] Felix Cloutier, "x86 and amd64 instruction reference" (2024), accessed 2026-02-01. URL: https://www.felixcloutier.com/x86/. Citations in this document: §1.1.

[17] AAC Clyde Space, "SIRIUS-OBC-LEON3FT" (2023), accessed 2026-02-01. URL: https://www.aac-clyde.space/what-we-do/space-products-components/command-data-handling/smallsat-sirius-obc. Citations in this document: §1.2.

[18] Nicholas Coughlin, Alistair Michael, Kait Lam, "Lift-offline: instruction lifter generators". In: SAS 2024 [**24**] (2024), 86–119. DOI: 10.1007/978-3-031-74776-2_4. Citations in this document: §4.

[19] Jos Craaijo, Freek Verbeek, Binoy Ravindran, "libLISA: instruction discovery and analysis on x86-64". In: Proceedings of the ACM on Programming Languages **8.OOPSLA2** (2024), 333–361. URL: https://liblisa.nl/liblisa.pdf. DOI: 10.1145/3689723. Citations in this document: §1.1.

[20] Babak Falsafi, Michael Ferdman, Shan Lu, Thomas F. Wenisch (editors), *ASPLOS '22: 27th ACM international conference on architectural support for programming languages and operating systems, Lausanne, Switzerland, 28 February 2022—4 March 2022*, ACM, 2022. ISBN 978-1-4503-9205-1. DOI: 10.1145/3503222. See [32].

[21] John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, Anna Philippou (editors), *FM 2016: formal methods—21st international symposium, Limassol, Cyprus, November 9–11, 2016, proceedings*, 9995, 2016. ISBN 978-3-319-48988-9. DOI: 10.1007/978-3-319-48989-6. See [28].

[22] Paul Floyd, "Re: Looking for a copy valgrind-solaris repository" (2025). URL: https://sourceforge.net/p/valgrind/mailman/message/59192170/. Citations in this document: §2.6.

[23] Jiri Gaisler, "LEON-1 processor—first evaluation results". In: ESCCON 2000 [**49**] (2000). URL: https://adsabs.harvard.edu/pdf/2000ESASP.439..183G. Citations in this document: §1.2.

[24] Roberto Giacobazzi, Alessandra Gorla (editors), *Static analysis—31st international symposium, SAS 2024, Pasadena, CA, USA, October 20–22, 2024, proceedings*, 14995, Springer, 2025. ISBN 978-3-031-74775-5. DOI: 10.1007/978-3-031-74776-2. See [18].

[25] Patrice Godefroid, Nils Klarlund, Koushik Sen, "DART: directed automated random testing". In: PLDI 2005 [**48**] (2005), 213–223. URL: https://cm-bell-labs.github.io/who/god/public_psfiles/pldi2005.pdf. DOI: 10.1145/1065010.1065036. Citations in this document: §1.

[26] John Harrison, "The x25519 function for curve25519" (2023), accessed 2026-02-01. URL: https://github.com/awslabs/s2n-bignum/blob/main/x86/proofs/curve25519_x25519.ml. Citations in this document: §4.

[27] Niranjan Hasabnis, R. Sekar, "Extracting instruction semantics via symbolic execution of code generators". In: SIGSOFT FSE 2016 [**54**] (2016), 301–313. DOI: 10.1145/2950290.2950335. Citations in this document: §4, §4.

[28] Zhe Hou, David Sanán, Alwen Tiu, Yang Liu, Koh Chuen Hoa, "An executable formalisation of the SPARCv8 instruction set architecture: A case study for the LEON3 processor". In: FM 2016 [**21**] (2016), 388–405. URL: https://research-repository.griffith.edu.au/bitstreams/4172ad81-ec01-464d-b430-c2b46430cc36/download. DOI: 10.1007/978-3-319-48989-6_24. Citations in this document: §4.

[29] Zhé Hóu, David Sanán, Alwen Tiu, Yang Liu, Koh Chuen Hoa, Jin Song Dong, "An Isabelle/HOL formalisation of the SPARC instruction set architecture and the TSO memory model". In: Journal of Automated Reasoning **65** (2021), 569–598. URL: https://research-repository.griffith.edu.au/server/api/core/bitstreams/351fbab6-a37a-45f3-90fc-1380b27064e5/content. DOI: 10.1007/s10817-020-09579-4. Citations in this document: §4.

[30] SPARC International, *The SPARC architecture manual: version 8*, 1992. URL: https://web.archive.org/web/20050204100221/https://www.sparc.org/standards/V8.pdf. Citations in this document: §4.1.

[31] Ranjit Jhala, Isil Dillig (editors), *PLDI '22: 43rd ACM SIGPLAN international conference on programming language design and implementation, San Diego, CA, USA, June 13–17, 2022*, ACM, 2022. ISBN 978-1-4503-9265-5. DOI: 10.1145/3519939. See [47].

[32] Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, Kui Ren, "EXAMINER: automatically locating inconsistent instructions between real devices and CPU emulators for ARM". In: ASPLOS 2022 [**20**] (2022), 846–858. URL: https://arxiv.org/abs/2105.14273. DOI: 10.1145/3503222.3507736. Citations in this document: §4, §4.

[33] Nicholas J. Kain, "Port musl to x86-64. One giant commit!" (2011). URL: https://web.archive.org/web/20200815021735/https://git.musl-libc.org/cgit/musl/commit/?id=1e12632591ab98a6ea3af8680716c28282552981. Citations in this document: §3.2.

[34] Kait Lam, Nicholas Coughlin, "Lift-off: trustworthy ARMv8 semantics from formal specifications". In: FMCAD 2023 [**39**] (2023), 274–283. URL: https://repositum.tuwien.at/bitstream/20.500.12708/188857/1/Lam-2023-Lift-off%20Trustworthy%20ARMv8%20semantics%20from%20formal%20specifications-vor.pdf. DOI: 10.34727/2023/isbn.978-3-85448-060-0_36. Citations in this document: §4, §4.

[35] Felix von Leitner, "diet libc" (2001). URL: https://web.archive.org/web/20250305021644/https://www.fefe.de/dietlibc/talk.pdf. Citations in this document: §3.3.

[36] Daniel Lüdtke, Thomas Firchau, Carlos Gonzalez Cortes, Andreas Lund, Ayush Mani Nepal, Mahmoud M. Elbarrawy, Zain Haj Hammadeh, Jan-Gerd Meß, Patrick Kenny, Fiona Brömer, Michael Mirzaagha, George Saleip, Hannah Kirstein, Christoph Kirchhefer, Andreas Gerndt, "ScOSA on the way to orbit: reconfigurable high-performance computing for spacecraft". In: SCC 2023 [**4**] (2023), 34–44. URL: https://elib.dlr.de/196642/1/ScOSA-SCC2023-preprint.pdf. DOI: 10.1109/SCC57168.2023.00015. Citations in this document: §1.2.

[37] Tanya Mineeva, "Add support for AVX-512 instructions" (2017), accessed 2026-02-01. URL: https://bugs.kde.org/show_bug.cgi?id=383010. Citations in this document: §2.6.

[38] Leonardo Mendonça de Moura, Nikolaj S. Bjørner, "Z3: an efficient SMT solver". In: TACAS 2008 [**45**] (2008), 337–340. DOI: 10.1007/978-3-540-78800-3_24. Citations in this document: §2.3.

[39] Alexander Nadel, Kristin Yvonne Rozier (editors), *Formal methods in computer-aided design, FMCAD 2023, Ames, IA, USA, October 24–27, 2023*, IEEE, 2023. ISBN 978-3-85448-060-0. URL: https://ieeexplore.ieee.org/xpl/conhome/10329310/proceeding. See [34].

[40] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, Xi Wang, "Scaling symbolic evaluation for automated verification of systems code with Serval". In: SOSP 2019 [**12**] (2019), 225–242. URL: https://www.cs.columbia.edu/~rgu/publications/sosp19-nelson.pdf. DOI: 10.1145/3341301.3359641. Citations in this document: §4.

[41] Anh Quynh Nguyen, Hoang Vu Dang, "Unicorn: next generation CPU emulator framework" (2015). URL: https://www.blackhat.com/docs/us-15/materials/us-15-Nguyen-Unicorn-Next-Generation-CPU-Emulator-Framework.pdf. Citations in this document: §1.2, §3.2.

[42] Noah Perryman, Nicholas Franconi, Gary Crum, Christopher Wilson, Alan D. George, "SpaceCube GHOST: a resilient processor for low-power, high-reliability space computing". In: AeroConf 2024 [**5**] (2024), 1–11. DOI: 10.1109/AERO58975.2024.10521244. Citations in this document: §1.2.

[43] Sebastian Poeplau, Aurélien Francillon, "Symbolic execution with SymCC: don't interpret, compile!". In: USENIX Security 2020 [**13**] (2020), 181–198. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau. Citations in this document: §1.

[44] Sebastian Poeplau, Aurélien Francillon, "SymQEMU: compilation-based symbolic execution for binaries". In: NDSS 2021 [**3**] (2021). URL: https://www.ndss-symposium.org/ndss-paper/symqemu-compilation-based-symbolic-execution-for-binaries/. Citations in this document: §1.

[45] C. R. Ramakrishnan, Jakob Rehof (editors), *Tools and algorithms for the construction and analysis of systems, 14th international conference, TACAS 2008, held as part of the joint European conferences on theory and practice of software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008, proceedings*, 4963, Springer, 2008. ISBN 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3. See [38].

[46] Markku Saarinen, "ROUND 3 OFFICIAL COMMENT: FrodoKEM – CCA Bug" (2020), email dated 10 Dec 2020 07:11:18 -0800. URL: https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/kSUKzDNc5ME/m/EMFYz9RNCAAJ. Citations in this document: §2.2.

[47] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, Peter Sewell, "Islaris: verification of machine code against authoritative ISA semantics". In: PLDI 2022 [**31**] (2022), 825–840. URL: https://pub.ista.ac.at/~msammler/paper/islaris.pdf. DOI: 10.1145/3519939.3523434. Citations in this document: §4.

[48] Vivek Sarkar, Mary W. Hall (editors), *Proceedings of the ACM SIGPLAN 2005 conference on programming language design and implementation, Chicago, IL, USA, June 12–15, 2005*, ACM, 2005. ISBN 1-59593-056-6. DOI: 10.1145/1065010. See [25].

[49] B. Schürmann (editor), *Proceedings of the European Space Components Conferences, ESCCON 2000, 21–23 March 2000, ESTEC, Noordwijk, The Netherlands*, ESA-SP, 439, European Space Agency, 2000. See [23].

[50] Koushik Sen, Darko Marinov, Gul Agha, "CUTE: a concolic unit testing engine for C". In: ESEC/FSE 2005 [**53**] (2005), 263–272. URL: https://www.ideals.illinois.edu/items/11128/bitstreams/40973/data.pdf. DOI: 10.1145/1081706.1081750. Citations in this document: §1.

[51] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, Giovanni Vigna, "(State of) the art of war: offensive techniques in binary analysis". In: S&P 2016 [**2**] (2016), 138–157. URL: https://sites.cs.ucsb.edu/~vigna/publications/2016_SP_angrSoK.pdf. DOI: 10.1109/SP.2016.17. Citations in this document: §1, §2.

[52] Eleonora Vacca, Corrado De Sio, Sarah Azimi, Luca Sterpone, "On assessing the robustness of RISC-V soft cores for space systems by mission-tailored SEU analysis". In: ICECS 2024 [**6**] (2024), 1–4. DOI: 10.1109/ICECS61496.2024.10848907. Citations in this document: §1.2.

[53] Michel Wermelinger, Harald C. Gall (editors), *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT*

international symposium on foundations of software engineering, 2005, Lisbon, Portugal, September 5–9, 2005, ACM, 2005. ISBN 1-59593-014-0. DOI: 10.1145/1081706. See [50].

[54] Thomas Zimmermann, Jane Cleland-Huang, Zhendong Su (editors), *Proceedings of the 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE 2016, Seattle, WA, USA, November 13–18, 2016*, ACM, 2016. ISBN 978-1-4503-4218-6. DOI: 10.1145/2950290. See [27].