On the Self in Selfie

Christoph M. Kirsch
What is the meaning of this sentence?

Selfie as in self-referentiality
Teaching the Construction of Semantics of Formalisms

Compilation

Interpretation

Virtualization

Verification
Joint Work

- Alireza Abyaneh
- Martin Aigner
- Sebastian Arming
- Christian Barthel
- Simon Bauer
- Thomas Hütter
- Alexander Kollert
- Michael Lippautz
- Cornelia Mayer
- Philipp Mayer
- Christian Moesl
- Simone Oblasser
- Clement Poncelet
- Sara Seidl
- Ana Sokolova
- Manuel Widmoser
Inspiration

- Armin Biere: SAT/SMT Solvers
- Donald Knuth: Art
- Jochen Liedtke: Microkernels
- Hennessy/Patterson: RISC
- Niklaus Wirth: Compilers
Selfie: Teaching Computer Science

[自我.cs.uni-salzburg.at]

Selfie is a self-referential 10K-line C implementation (in a single file) of:

1. a self-compiling compiler called **starc** that compiles a tiny subset of C called C Star (C*) to a tiny subset of RISC-V called RISC-U,

2. a self-executing emulator called **mipster** that executes RISC-U code including itself when compiled with starc,

3. a self-hosting hypervisor called **hypster** that virtualizes mipster and can host all of selfie including itself,

4. a self-executing symbolic execution engine called **monster** that executes RISC-U code symbolically when compiled with starc which includes all of selfie,

5. a tiny C* library called **libcstar** utilized by all of selfie, and

6. a tiny, experimental SAT solver called **babysat**.
Selfie supports the official 64-bit RISC-V toolchain and runs on the spike emulator and the pk kernel.
Also, there is a…

- linker (in-memory only)
- disassembler (w/ source code line numbers)
- debugger (tracks full machine state w/ rollback)
- profiler (#proc-calls, #loop-iterations, #loads, #stores)
- ELF boot loader (same code for mipster/hypster)
Code as Prose

```c
uint64_t left_shift(uint64_t n, uint64_t b) {
    // assert: 0 <= b < CPUBITWIDTH
    return n * two_to_the_power_of(b);
}

uint64_t right_shift(uint64_t n, uint64_t b) {
    // assert: 0 <= b < CPUBITWIDTH
    return n / two_to_the_power_of(b);
}

uint64_t get_bits(uint64_t n, uint64_t i, uint64_t b) {
    // assert: 0 < b <= i + b < CPUBITWIDTH
    if (i == 0)
        return n % two_to_the_power_of(b);
    else
        // shift to-be-loaded bits all the way to the left
        // to reset all bits to the left of them, then
        // shift to-be-loaded bits all the way to the right and return
        return right_shift(left_shift(n, CPUBITWIDTH - (i + b)), CPUBITWIDTH - b);
}
```
Discussion of Selfie reached 3rd place on Hacker News

news.ycombinator.com
Website

selfie.cs.uni-salzburg.at

Code

github.com/cksystemsteaching/selfie

Slides (250 done, ~250 todo)

selfie.cs.uni-salzburg.at/slides

Book (draft)

leanpub.com/selfie
```c
uint64_t atoi(uint64_t *s) {
    uint64_t i;
    uint64_t n;
    uint64_t c;

    i = 0;
    n = 0;
    c = *(s+i);

    while (c != 0) {
        n = n * 10 + c - '0';
        if (n < 0)
            return -1;
        i = i + 1;
        c = *(s+i);
    }

    return n;
}
```

- 5 statements: assignment, while, if, return, procedure().
- No data types other than `uint64_t` and `uint64_t*` and dereferencing: the `*` operator.
- Integer arithmetics, pointer arithmetics.
- No bitwise operators, no Boolean operators.
- Character literals, string literals.
- Library: `exit`, `malloc`, `open`, `read`, `write`.

Minimally complex, maximally self-contained system

Programming languages vs systems engineering?
> make
cc -w -O3 -m64 -D'main(a,b)=main(int argc, char** argv)'
-Duint64_t='unsigned long long' selfie.c -o selfie

bootstrapping **selfie.c** into x86 **selfie** executable using standard C compiler
> ./selfie
usage: selfie
{ -c { source } | -o binary | [ -s | -S ] assembly | -l binary | -sat dimacs } [ ( -m | -d | -r | -n | -y | -min | -mob ) 0-64 ... ]
> ./selfie -c selfie.c

selfie compiling selfie.c with starc

289095 characters read in 10034 lines and 1335 comments with 170555 (58.99%) characters in 43772 actual symbols
341 global variables, 438 procedures, 411 string literals
2517 calls, 1139 assignments, 86 while, 874 if, 391 return
symbol table search time was 2 iterations on average and
48795 in total

170504 bytes generated with 39496 instructions and 12520 bytes of data

init:  lui: 2296 (5.81%), addi: 13595 (34.40%)
memory: ld: 7106 (17.98%), sd: 5884 (14.89%)
compute: add: 3422 (8.65%), sub: 704 (1.78%), mul: 807 (2.40%),
divu: 78 (0.19%), remu: 35 (0.80%)
control: sltu: 624 (1.57%), beq: 964 (2.43%),
jal: 3555 (8.99%), jalr: 438 (1.10%), ecall: 8 (0.20%)

compiling selfie.c with x86 selfie executable
(takes seconds)
> ./selfie -c selfie.c -m 3 -c selfie.c

selfie compiling selfie.c with starc

...selfie executing selfie.c with 3MB physical memory on mipster

selfie compiling selfie.c with starc

selfie.c exiting with exit code 0 and 2.11MB mallocated memory

summary: 285261695 executed instructions and 2.10MB mapped memory

init: lui: 836418 (0.29%), addi: 120536779 (42.25%)

memory: ld: 61562613 (21.58%), sd: 39713446 (13.92%)

compute: add: 7234823 (2.53%), sub: 5903746 (2.60%), mul: 6878318 (2.41%), div: 2100676 (0.73%), rem: 2016943 (0.70%)

control: sltu: 4436689 (1.55%), beq: 6011381 (2.10%), jal: 18600397 (6.52%), jalr: 9118787 (3.19%), ecall: 310679 (0.10%)

profile: total, max(ratio%)@addr(line#), 2max, 3max

calls: 9118787, 2492778 (27.33%)@0x282C (~1671), ...

loops: 500189, 164040 (32.79%)@0x355C (~1859), ...

loads: 61562613, 2492778 (4.40%)@0x2840 (~1671), ...

stores: 39713446, 2492778 (6.27%)@0x2830 (~1671), ...

compiling selfie.c with x86 selfie executable into a RISC-U executable and then running that RISC-U executable to compile selfie.c again (takes a minute)
compiling `selfie.c` into a RISC-U executable `selfie1.m` and then running `selfie1.m` to compile `selfie.c` into another RISC-U executable `selfie2.m` (takes a minute)
compiling `selfie.c` with x86 selfie executable and then running that executable to compile `selfie.c` again and then running that executable to compile `selfie.c` again (takes hours)
> ./selfie -c selfie.c -m 6 -c selfie.c -y 3 -c selfie.c

compiling `selfie.c` with x86 `selfie` executable

and

then running that executable to compile `selfie.c` again

and

then hosting that executable in a virtual machine to compile `selfie.c` again

(takes 2 minutes)
On the Self in Selfie

How does self-referentiality work in selfie?
Selfie Stick!

- Selfie compiler (RISC-V)
- Selfie hypervisor (RISC-V)
- Selfie hypervisor (RISC-V)
- Selfie emulator (RISC-V)
- Selfie emulator (x86)
- Spike emulator (x86) + pk kernel (RISC-V)
Self-Compilation

C* program

selfie source code

open read

selfie compiler

scanner

parser
code generator

malloc

exit

open
write

library code

generated code

selfie emulator

spike emulator + pk kernel

system call wrappers

exit
open
read
write
malloc
Generated Code: unsigned + code

```c
1 uint64_t x;
2 uint64_t main() {
3     x = 0;
4     x = x + 1;
5     if (x == 1)
6         x = x + 1;
7     else
8         x = x - 1;
9     while (x > 0)
10        x = x - 1;
11     return x;
12 }
```

64-bit RISC-V add instruction

- `0x150(~6): ld $t0,-16($gp)`
- `0x154(~6): addi $t1,$zero,1`
- `0x158(~6): add $t0,$t0,$t1`
- `0x15C(~6): sd $t0,-16($gp)`

C code for unsigned 64-bit integer addition
add implementation in selfie emulator

64-bit RISC-V add instruction

```c
void do_add() {
    if (rd != REG_ZR) {
        // semantics of add
        *(registers + rd) = *(registers + rs1) + *(registers + rs2);
        pc = pc + INSTRUCTIONSIZE;
        ic_add = ic_add + 1;
    }
}
```

C code for unsigned 64-bit integer addition

selfie compiler

gcc/clang
// RISC-V R Format
//   | 7 | 5 | 5 | 3 | 5 | 7 |
//   | funct7 | rs2 | rs1 | funct3 | rd | opcode |
//   | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 |
//   | 7 | 6 | 0 |

uint64_t encode_r_format(uint64_t funct7, uint64_t rs2, uint64_t rs1, uint64_t funct3, uint64_t rd, uint64_t opcode) {
    // assert: 0 <= funct7 < 2^7
    // assert: 0 <= rs2 < 2^5
    // assert: 0 <= rs1 < 2^3
    // assert: 0 <= funct3 < 2^3
    // assert: 0 <= rd < 2^5
    // assert: 0 <= opcode < 2^7
    return left_shift(left_shift(left_shift(left_shift(funct7, 5) + rs2, 5) + rs1, 3) + funct3, 5) + rd, 7) + opcode;
}

uint64_t get_funct7(uint64_t instruction) {
    return get_bits(instruction, 25, 7);
}

uint64_t get_rs2(uint64_t instruction) {
    return get_bits(instruction, 20, 5);
}

uint64_t get_rs1(uint64_t instruction) {
    return get_bits(instruction, 15, 5);
}

uint64_t get_funct3(uint64_t instruction) {
    return get_bits(instruction, 12, 3);
}

uint64_t get_rd(uint64_t instruction) {
    return get_bits(instruction, 7, 5);
}

uint64_t get_opcode(uint64_t instruction) {
    return get_bits(instruction, 0, 7);
}

void decode_r_format() {
    funct7 = get_funct7(ir);
    rs2 = get_rs2(ir);
    rs1 = get_rs1(ir);
    funct3 = get_funct3(ir);
    rd = get_rd(ir);
    imm = 0;
}
void emit_exit() {
  create_symbol_table_entry(LIBRARY_TABLE, (uint64_t*) "exit", 0, PROCEDURE, VOID_T, 0, binary_length);

  // load signed 32-bit integer argument for exit
  emit_ld(REG_A0, REG_SP, 0);

  // remove the argument from the stack
  emit_addi(REG_SP, REG_SP, REGISTERSIZE);

  // load the correct syscall number and invoke syscall
  emit_addi(REG_A7, REGZR, SYSCALL_EXIT);

  emit_ecall();

  // never returns here
}

void implement_exit(uint64_t* context) {
  if (disassemble) {
    print((uint64_t*) "(exit): ");
    print_register_hexadecimal(REG_A0);
    print((uint64_t*) " | - ->\n");
  }

  set_exit_code(context, sign_shrink(*(get_regs(context) + REG_A0), SYSCALL_BITWIDTH));
}
Library Code: open wrapper

parameters

syscall ID

selfie emulator

spike emulator + pk kernel

0xA8(~1): 0x00013603: ld $a2, 0($sp)
0xAC(~1): 0x00810113: addi $sp, $sp, 8
0xB0(~1): 0x00013583: ld $a1, 0($sp)
0xB4(~1): 0x00810113: addi $sp, $sp, 8
0xB8(~1): 0x00013503: ld $a0, 0($sp)
0xBC(~1): 0x00810113: addi $sp, $sp, 8
0xC0(~1): 0x40000893: addi $a7, $zero, 1024
0xC4(~1): 0x00000073: ecall
0xC8(~1): 0x00008067: jalr $zero, 0($ra)
open implementation in selfie emulator

```c
void implement_open(uint64_t* context) {
    // parameters
    uint64_t vfilename;
    uint64_t flags;
    uint64_t mode;

    // return value
    uint64_t fd;

    if (disassemble) {
        print(((uint64_t*)"(open): "));
        print_register_hexadecimal(REG_A0);
        print(((uint64_t*)","));
        print_register_hexadecimal(REG_A1);
        print(((uint64_t*)","));
        print_register_octal(REG_A2);
        print(((uint64_t*)" |- "));
        print_register_value(REG_A0);
    }

    vfilename = *(get_regs(context) + REG_A0);
    flags = *(get_regs(context) + REG_A1);
    mode = *(get_regs(context) + REG_A2);

    if (down_load_string(get_pt(context), vfilename, filename_buffer)) {
        fd = sign_extend(open(filename_buffer, flags, mode), SYSCALL_BITWIDTH);
    }
}
```
malloc is different!

malloc invokes the brk system call

both manage pure address spaces

actual memory storage is done in the paging system

```c
void implement_brk(uint64_t* context) {
    // parameter
    uint64_t program_break;

    // local variables
    uint64_t previous_program_break;
    uint64_t valid;
    uint64_t size;

    if (disassemble) {
        print((uint64_t*) "(brk): ");
        print_register_hexadecimal(REG_A0);
    }

    program_break = *(get_regs(context) + REG_A0);

    previous_program_break = get_program_break(context);
    valid = 0;

    if (program_break >= previous_program_break)
        if (program_break < *(get_regs(context) + REG_SP))
            if (program_break % SIZEOFUINT64 == 0)
                valid = 1;

    if (valid) {
        if (disassemble)
            print((uint64_t*) "  |--->\n");

        if (debug_brk)
            printf2((uint64_t*) "%s: setting program break to %p\n", set_program_break(context, program_break));
    }
}
```
Self-Execution

- **Execution Flow**:
  - **emulator0** executes user code in **context0**.
  - User code triggers exception, which is handled by **context1**.
  - **emulator1** resumes execution in **context1**.
RISC-U Machine State

**context**

- 32x 64-bit CPU registers
- 1x 64-bit program counter

**4GB of byte-addressed 64-bit-word-aligned main memory**
Virtual Memory in Selfie

- 4GB of byte-addressed 64-bit-word-aligned virtual memory
- 4KB-paged on demand
- MBs of byte-addressed 64-bit-word-aligned physical memory
Code Execution and Exceptions

13+1 instructions:
- lui addi
- add sub mul
- divu remu
- ld sd
- sltu beq
- jal jalr
- ecall

1. division-by-0
2. page fault
3. timer interrupt
4. system call

selfie emulator
spike emulator + pk kernel
Self-Execution Revisited

Diagram:
- emulator0
- context0
- emulator1
- context1
- user code

Arrows:
- execute
- exception

Relationships:
- emulator0 to context0
- emulator1 to context1
- user code to context1
- exception from context0 to emulator0
- exception from context1 to user code
Self-Execution: Concurrency
Hosting: Concurrency

- hypervisor
- context0
- context1
- context2
- user1
- user2

Execution flow:
- Emulator
  - Execute
  - Exception
- Hypervisor
  - Switch
  - Exception
- User 1
  - Context 1
  - Context 2
- User 2
  - Context 1
  - Context 2
Emulation versus Virtualization

```c
while (1) {
    if (mix) {
        from_context = mipster_switch(to_context, TIMESLICE);
    } else {
        from_context = hypster_switch(to_context, TIMESLICE);
    }
    if (get_parent(from_context) != MY_CONTEXT) {
        // switch to parent which is in charge of handling exceptions
        to_context = get_parent(from_context);
        timeout = TIMEROFF;
    } else if (handle_exception(from_context) == EXIT) {
        return get_exit_code(from_context);
    } else {
        // TODO: scheduler should go here
        to_context = from_context;
        if (mix) {
            if (mslice != TIMESLICE) {
                mix = 0;
                timeout = TIMESLICE - mslice;
            } else if (mslice > 0) {
                mix = 1;
                timeout = mslice;
            }
        }
    }
}
```
Self-Hosting: Hierarchy

- **emulator**
- **hypervisor**
- **context0**
- **context1**
- **context2**
- **user**

Flow:
- **emulator** → **executes** → **context0**
- **hypervisor** → **switches** → **context1**
- **context1** → **context2**
- **context2** → **user**
- **hypervisor** → **exceptions**
Selfie Teaching Experience

SPLASH-E
Tomorrow, 10.30am, Room Cambridge
Minimal Symbolic Execution?

What exactly is needed to execute systems code like selfie’s symbolically?
Selfie Symbolic Execution monster

Integrate w/ Standard SMT Solver

DIY solver to maintain self-containment
Replay vs. Symbolic Execution

- Selfie supports replay of RISC-U execution upon detecting runtime errors such as division by zero.
- Selfie first rolls back $n$ instructions (undo(!) semantics, system calls?) and then re-executes them but this time printed on the console.
- We use a cyclic buffer for replaying $n$ instructions.
- That buffer is also used in symbolic execution but then for recording symbolic execution of up to $n$ instructions.
Symbolic Execution: Status

- We fuzz input read from files
- Symbolic execution proceeds by computing integer interval constraints, only recording memory stores
- Sound but only complete for a subset of all programs
- Selfie compiler falls into that subset, so far…
- We detect division by zero, (some) unsafe memory access
Symbolic Execution: Future

- Witness generation and on-the-fly validation
- Loop termination through manually crafted invariants
- Parallelization on our 64-core machine
- And support for utilizing 0.5TB of physical memory
Got Research Ideas?

- Selfie is a simple but still realistic sandbox
- You control everything!
- Want to play with an idea that requires compiler / operating systems / architecture support?
- We are glad to help you get started!
Thank you!