CHRISTOPH KIRSCH, CTU PRAGUE

SYMBOLIC EXECUTION

WHAT IS SYMBOLIC EXECUTION?

- *Concrete* execution is the execution of code for a *single* input
- *Input* is everything that the executed code depends on (system calls, uninitialized memory or variables; no concurrency for now)
- *Symbolic* execution is the execution of code for *all* inputs (up to a given number of instructions or statements)
- *Concolic* execution is the execution of code for *some* inputs through concrete execution guided by symbolic execution

WHAT IS THE PURPOSE OF SYMBOLIC EXECUTION?

Given an upper bound n on the number of executed instructions or statements:

symbolic execution of code computes

the *set of all inputs* that

make the code produce runtime errors

within executing up to n instructions or statements

on any of the inputs in that set

HOW IS THE SET OF ALL INPUTS THAT LEAD TO RUNTIME ERRORS REPRESENTED?

EXAMPLE: SEQUENTIAL CODE

```
uint64_t x;
// 0 <= x <= UINT64_MAX
x := x + 1;
// x‘ == x + 1
// but still 0 <= x‘ <= UINT64_MAX
x := x * 2;
// x‘‘ == x‘ * 2 && x‘ == x + 1
// that is, x‘‘ == (x + 1) * 2
```

EXAMPLE: BRANCHING

```
...

// x'' == x' * 2 && x' == x + 1
if (x == UINT64_MAX)
  exit(1);
```

path condition ->

```
exit(1) is reachable
  if and only if
x'' == UINT64_MAX &&
  x'' == x' * 2 &&
    x' == x + 1
  is satisfiable
```

REACHABILITY IS SATISFIABILITY

```
code is reachable iff formula is satisfiable
```

but only if

```
     x := x + 1;    <>   x’ == x + 1
     x := x * 2;    <>   x‘ == x * 2
x == UINT64_MAX     <>   x == UINT64_MAX
```

C code

64-bit bit vector theory

SATISFIABILITY MODULO THEORY (SMT)

```
x'' == UINT64_MAX &&
x'' == x' * 2 &&
x' == x + 1
```

reduces to the SAT formula:

```
P && Q && R
```

modulo

the theory of 64-bit bit vectors:

+, *, ==, … over 64-bit bit vector variables x

SMT SOLVERS

Z3 (Microsoft Research)

CVC4 (Stanford)

boolector (JKU Linz)

…

EXAMPLE: BRANCHING

```
…
// x'' == x' * 2 && x' == x + 1
if (x == 0) {
  // x'' == 0 &&
  // x'' == x' * 2 &&
  // x' == x + 1
  // input: x == UINT64_MAX
  exit(1);
}
```

EXAMPLE: BRANCHING, FULLY CONCURRENT!

```
…

// x'' == x' * 2 && x' == x + 1

if (x == 0) {

   // x'' == 0 && x'' == x' * 2 && x' == x + 1

   // input: x == UINT64_MAX

} else {

   // x'' != 0 && x'' == x' * 2 && x' == x + 1

   // input: x != UINT64_MAX

}
```

EXAMPLE: SYMBOLIC EXECUTION

```
…
if (x == 0) {
  // A holds
} else {
  // B holds
}
// try with A
x := x - 1
// x''' == x'' - 1 && A
```

EXAMPLE: SYMBOLIC EXECUTION

```
…
if (x == 0) {
  // A holds
} else {
  // B holds
}
// then or simultaneously try with B
x := x - 1
// x''' == x'' - 1 && B
```

CHALLENGES

Should we explore both branches?

Are we still reachable anyway?

How often should we ask the SMT solver?

How do we organize the *symbolic* store to facilitate fast back tracking?

Can we integrate loop invariants for completeness?

How do we deal with procedures, recursion?

SYMBOLIC EXECUTION ENGINES

SAGE (Microsoft Research)

KLEE (Imperial)

…

EXAMPLE: BOUNDED MODEL CHECKING (MERGING)

```
…
if (x == 0) {
  // A holds
} else {
  // B holds
}
// or try A || B
x := x - 1
// x''' == x'' - 1 && (A || B)
```

CHALLENGES

Should we merge or not?

Are we, the symbolic execution engine, or the SMT solver more efficient?

OUR RESEARCH

Is there a way to generate *minimal* formulae

for bounded model checking?

(we generate SMT-LIB formulae)

Can we offload the work entirely to a solver?

(we generate BTOR2 formulae)

SYMBOLIC EXECUTION OF RISC-V CODE

32x 64-bit registers

64-bit program counter

4GB byte-addressed, 64-bit-word-aligned memory

machine instructions are encoded in 32 bits

INITIALIZATION

```
lui rd,imm:

rd = imm * 2^12; pc = pc + 4
with -2^19 <= imm < 2^19


addi rd,rs1,imm:

rd = rs1 + imm; pc = pc + 4
with -2^11 <= imm < 2^11


SMT-LIB: bvadd
```

ARITHMETIC

```
add rd,rs1,rs2: rd = rs1 + rs2; pc = pc + 4

sub rd,rs1,rs2: rd = rs1 - rs2; pc = pc + 4

mul rd,rs1,rs2: rd = rs1 * rs2; pc = pc + 4

divu rd,rs1,rs2: rd = rs1 / rs2; pc = pc + 4
where rs1 and rs2 are unsigned integers.

remu rd,rs1,rs2: rd = rs1 % rs2; pc = pc + 4
where rs1 and rs2 are unsigned integers.

SMT-LIB: bvadd, bvsub, bvmul, bvudiv, bvurem
```

COMPARISON

```
sltu rd,rs1,rs2:

if (rs1 < rs2) {
  rd = 1;
} else {
  rd = 0;
}

pc = pc + 4;
```

where rs1 and rs2 are unsigned integers.

SMT-LIB: bvult

MEMORY

```
ld rd,imm(rs1):

rd = memory[rs1 + imm]; pc = pc + 4
with -2^11 <= imm < 2^11


sd rs2,imm(rs1):

memory[rs1 + imm] = rs2; pc = pc + 4
with -2^11 <= imm < 2^11
```

CONTROL

```
beq rs1,rs2,imm:

if (rs1 == rs2) pc = pc + imm else pc = pc + 4
with -2^12 <= imm < 2^12 and imm % 2 == 0

SMT-LIB: bvcomp

jal rd,imm:

rd = pc + 4; pc = pc + imm
with -2^20 <= imm < 2^20 and imm % 2 == 0

jalr rd,imm(rs1):

tmp = ((rs1 + imm) / 2) * 2;
rd = pc + 4; pc = tmp with -2^11 <= imm < 2^11
```

SYSTEM CALLS

`ecall:`

`system call number is in a7,`
`parameters are in a0-a2,`
`return value is in a0.`

`exit, brk, open, read, write`

SELFIE MONSTER

```
./selfie -c example.c -se 0 30
```

generates

example.smt

```
./selfie -c example.c -se 0 30 --merge-enabled
```

BOUNDED MODEL CHECKING WITH BTOR2

arithmetic: add, sub, mul, udiv, urem

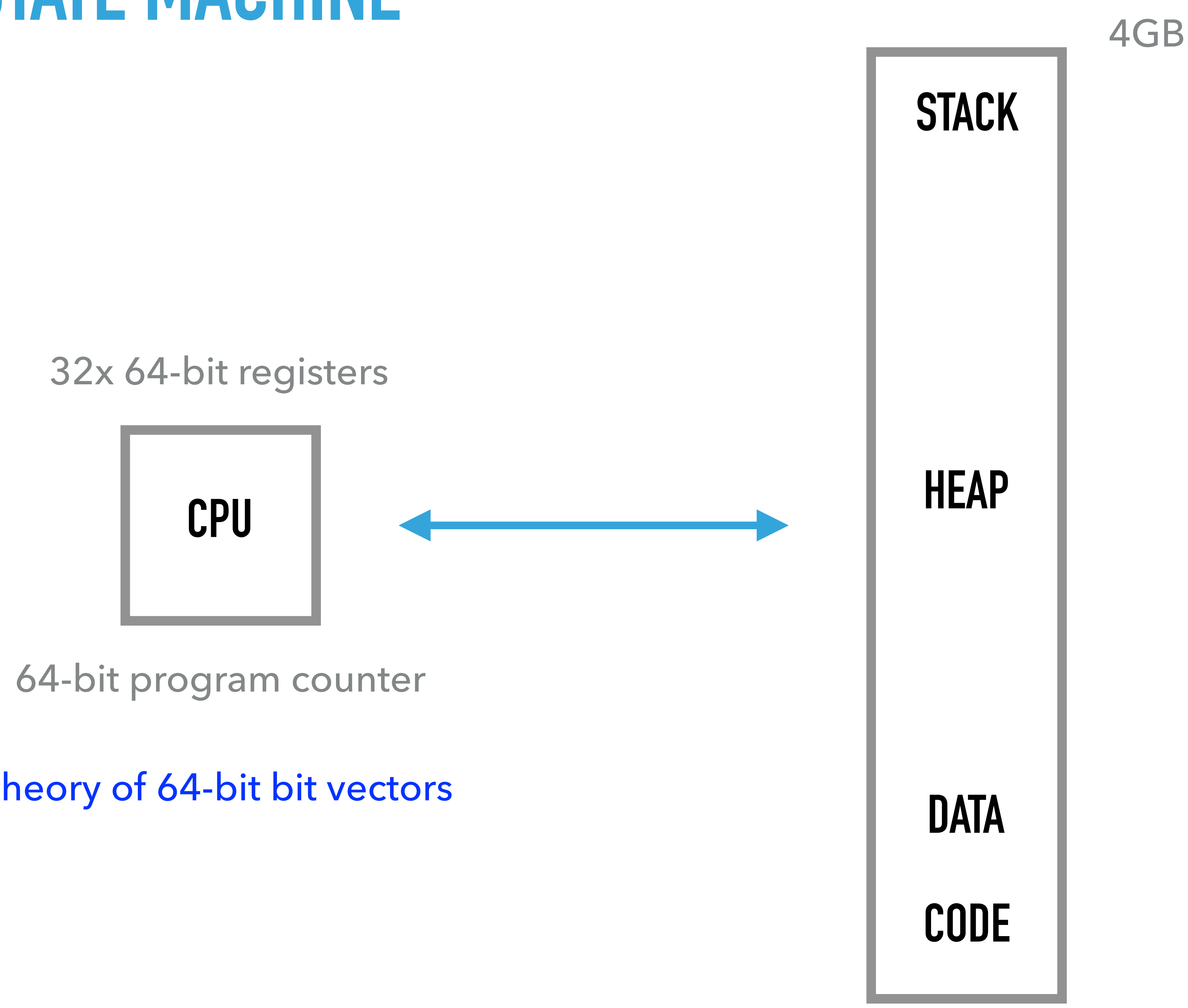
comparison: ult

memory: read, write

control: eq, ite

runtime error: bad

CODE AS STATE MACHINE



theory of 64-bit bit vectors

theory of 64-bit bit vector arrays

RISC-V MACHINE STATE

each register is a 64-bit bit vector

the program counter is encoded by
one bit for each instruction that is set
if the instruction is currently being executed

memory is a 64-bit bit vector array
initialized with data segment and stack

plus control and data flow

RISC-V INSTRUCTIONS AS STATE TRANSITIONS

an instruction changes at most two 64-bit words:

a 64-bit register or a 64-bit memory word or nothing (data flow)

and

the 64-bit program counter (control flow)

RISC-V SYSTEM CALLS AS STATE TRANSITIONS

system calls are implemented as follows:

`exit`: final state

`brk`: bump pointer

`open`: file descriptor

`read`: write to memory

`write`: read from memory

SELFIE MONSTER

```
./selfie -c example.c -mc 0
```

generates

```
example.btor2
```

SELFIE MONSTER

DEMO