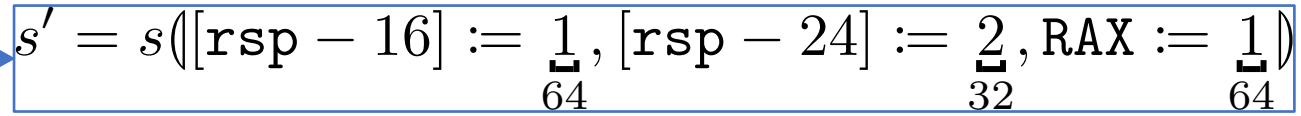
Symbolic Execution of x86 assembly in Isabelle/HOL

Freek Verbeek, Abhijith Bharadwaj, Joshua Bockenek, Ian Roessle, Binoy Ravindran

Symbolic Execution of x86-64

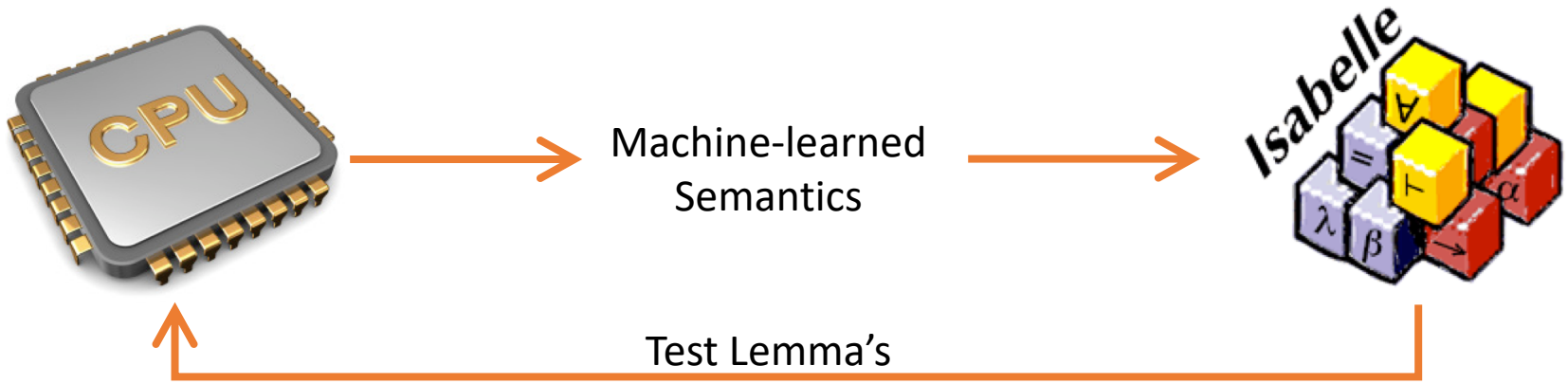
- Symbolic execution = instructions semantics + rewrite rules
- As a **proof technique**
- Requires reasoning over memory regions
- Per-block symbolic execution can be used in Hoare-style reasoning over assembly

```
mov QWORD PTR [rsp-16], 1
mov DWORD PTR [rsp-24], 2
mov rax, QWORD PTR [rsp-16]
```



Formal x86 semantics

x86-64 machine code:
1. Comprehensive formal model of x86-64 semantics
2. Testing setup for formal semantics
3. Symbolic execution engine

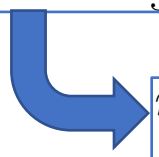


[1] Heule et al.: Stratified Synthesis: *Automatically Learning the x86-64 Instruction Set* (PLDI'16)
[2] Roessle et al.: *Formally Verified Big Step Semantics out of x86-64 Binaries (*CPP'19).

Example: `sub r32, m32`

$$
\begin{aligned}
r64 &:= \underset{32}{0} \frown \langle 31,0\rangle(\underset{1}{0} \frown \neg r_{mem}(a,4,\sigma) + \underset{33}{1} + \underset{1}{0} \frown \langle 31,0\rangle(r(r64,\sigma)))\\
ZF &:= \langle 31,0\rangle(\underset{1}{0} \frown \neg r_{mem}(a,4,\sigma) + \underset{33}{1} + \underset{1}{0} \frown \langle 31,0\rangle(r(r64,\sigma))) == \underset{32}{0}\\
CF &:= \langle 32,32\rangle(\underset{1}{0} \frown \neg r_{mem}(a,4,\sigma) + \underset{33}{1} + \underset{1}{0} \frown \langle 31,0\rangle(r(r64,\sigma))) == \underset{32}{1}\\
SF &:= \langle 31,31\rangle(\underset{1}{0} \frown \neg r_{mem}(a,4,\sigma) + \underset{33}{1} + \underset{1}{0} \frown \langle 31,0\rangle(r(r64,\sigma))) == \underset{32}{1}\\
OF &:= \neg\langle 31,31\rangle(r_{mem}(a,4,\sigma)) == \underset{1}{1} \longleftrightarrow \langle 31,31\rangle(r(r64,\sigma) == \underset{1}{1}) \wedge\\
&\quad \neg(\neg(\langle 31,31\rangle(r_{mem}(a,4,\sigma))) == \underset{1}{1} \longleftrightarrow\\
&\qquad \langle 31,31\rangle(\underset{1}{0} \frown \neg r_{mem}(a,4,\sigma) + \underset{33}{1} + \underset{1}{0} \frown \langle 31,0\rangle(r(r64,\sigma))) == \underset{1}{1})
\end{aligned}
$$

$$
\begin{array}{lcl}
r64 & := & \underset{32}{0} \frown \langle 31,0\rangle(\underset{1}{0} \frown \neg r_{mem}(a,4,\sigma) + \underset{33}{1} + \underset{1}{0} \frown \langle 31,0\rangle(r(r64,\sigma))) \\
ZF & := & \langle 31,0\rangle(\underset{1}{0} \frown \neg r_{mem}(a,4,\sigma) + \underset{33}{1} + \underset{1}{0} \frown \langle 31,0\rangle(r(r64,\sigma))) == \underset{32}{0} \\
CF & := & \langle 32,32\rangle(\underset{1}{0} \frown \neg r_{mem}(a,4,\sigma) + \underset{33}{1} + \underset{1}{0} \frown \langle 31,0\rangle(r(r64,\sigma))) == \underset{32}{1} \\
SF & := & \langle 31,31\rangle(\underset{1}{0} \frown \neg r_{mem}(a,4,\sigma) + \underset{33}{1} + \underset{1}{0} \frown \langle 31,0\rangle(r(r64,\sigma))) == \underset{32}{1} \\
OF & := & \neg\langle 31,31\rangle(r_{mem}(a,4,\sigma)) == \underset{1}{1} \longleftrightarrow \langle 31,31\rangle(r(r64,\sigma) == \underset{1}{1}) \wedge \\
 & & \neg(\neg(\langle 31,31\rangle(r_{mem}(a,4,\sigma))) == \underset{1}{1} \longleftrightarrow \\
 & & \quad \langle 31,31\rangle(\underset{1}{0} \frown \neg r_{mem}(a,4,\sigma) + \underset{33}{1} + \underset{1}{0} \frown \langle 31,0\rangle(r(r64,\sigma))) == \underset{1}{1})
\end{array}
$$



$$
\begin{array}{lcl}
r64 & := & \text{zextend}(op1 - op2) \\
ZF & := & op1 = op2 \\
CF & := & op1 < op2 \\
SF & := & \text{sint}(op1 - op2) < 0 \\
OF & := & (op1 \geq 2^{31}) \\
 & & \longleftrightarrow (op1 < 2^{31}) \\
 & & \longleftrightarrow (\text{sint}(op1 - op2) \geq 0)
\end{array}
$$

where $op1 = \langle 31,0\rangle(r(r64,\sigma)), op2 = r_{mem}(a,4,\sigma)$

Formal x86 semantics

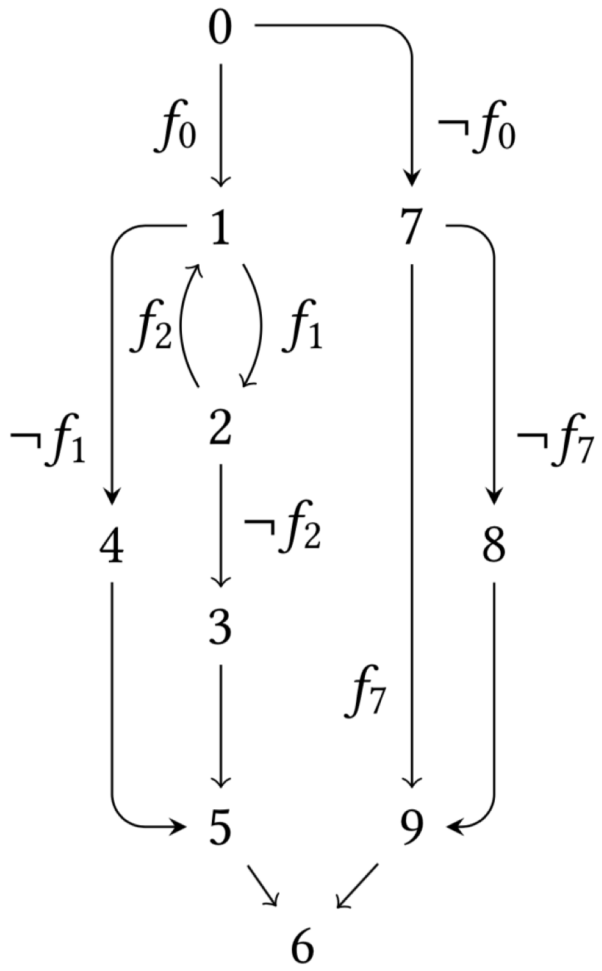
- Machine-learned semantics for 1625 instruction variants
- Floating-points operations supported (non-executable)
- Embedded into Isabelle/HOL

```
0000000000003e90 <dump_snapshots>:
    3e90:      push   r12
    3e92:      push   rbp
    3e93:      push   rbx
    3e94:      sub    rsp,0x120
    3e9b:      mov    rax,QWORD PTR fs:0x28
    3ea4:      mov    QWORD PTR [rsp+0x118],rax
    3eac:      xor    eax,eax
    3eae:      lea    rsi,[rsp+0x8]
    3eb3:      call   19cc0 <bdrv_snapshot_list>
    3eb8:      test   eax,eax
    3eba:      jle    3f33 <dump_snapshots+0xa3>
    3ebc:      mov    ebx,eax
    3ebe:      lea    rdi,[rip+0x1914e
    3ec5:      lea    r12,[rsp+0x10]
    3eca:      call   3130 <puts@plt>
    3ecf:      lea    ebp,[rbx-0x1]
    3ed2:      xor    edx,edx
    3ed4:      mov    esi,0x100
    3ed9:      mov    rdi,r12
    3edc:      add    rbp,0x1
    3ee0:      xor    ebx,ebx
    3ee2:      call   19e60 <bdrv_snapshot_dump>
    3ee7:      imul   rbp,rbp,0x198
    3eee:      mov    rdi,rax
    3ef1:      call   3130 <puts@plt>
    3ef6:      nop    WORD PTR cs:[rax+rax*1+0x0]
    3f00:      mov    rdx,QWORD PTR [rsp+0x8]
    3f05:      mov    esi,0x100
    3f0a:      mov    rdi,r12
    3f0d:      add    rdx,rbx
    3f10:      add    rbx,0x198
    3f17:      call   19e60 <bdrv_snapshot_dump>
    3f1c:      mov    rdi,rax
    3f1f:      call   3130 <puts@plt>
    3f24:      cmp    rbx,rbp
    3f27:      jne    3f00 <dump_snapshots+0x70>
    3f29:      mov    rdi,QWORD PTR [rsp+0x8]
    3f2e:      call   5960 <qemu_free>
    3f33:      mov    rax,QWORD PTR [rsp+0x118]
    3f3b:      xor    rax,QWORD PTR fs:0x28
    3f44:      jne    3f52 <dump_snapshots+0xc2>
    3f46:      add    rsp,0x120
    3f4d:      pop    rbx
    3f4e:      pop    rbp
    3f4f:      pop    r12
    3f51:      ret
    3f52:      call   31c0 <__stack_chk_fail@plt>
    3f57:      nop    WORD PTR [rax+rax*1+0x0]
```

Apply Hoare logic to assembly

1. Extract control flow

Control Flow Extraction



Block 0;
If f_0 Then
  Loop
    Block 1;
    If f_1 Then
      Block 2; If f_2 Then Continue Else Break 2 Fi
    Else Break 1 Fi
  Pool Resume{(2, Block 3), (1, Block 4)}
  Block 5
Else
  Block 7;
  If f_7 Then Skip Else Block 8 Fi;
  Block 9
Fi;
Block 6

```
definition dump_snapshots_acode :: ACode where
  ‹dump_snapshots_acode =
    Block 8 0x3eae 0;
    Block (Suc 0) 0x3eb3 (Suc 0);
    CALL bdrv_snapshot_list_acode;
    Block (Suc 0) 0x3eb8 2;
    IF jle THEN
      Block (Suc 0) 0x3eba 3
    ELSE
      Block 4 0x3ec5 4;
      Block (Suc 0) 0x3eca 5;
      CALL puts_acode;
      Block 6 0x3ee0 6;
      Block (Suc 0) 0x3ee2 7;
      CALL bdrv_snapshot_dump_acode;
      Block 2 0x3eee 8;
      Block (Suc 0) 0x3ef1 9;
      CALL puts_acode;
      Block (Suc 0) 0x3ef6 10;
      WHILE P_0x3f00_11 DO
        Block 5 0x3f10 11;
        Block (Suc 0) 0x3f17 12;
        CALL bdrv_snapshot_dump_acode;
        Block (Suc 0) 0x3f1c 13;
        Block (Suc 0) 0x3f1f 14;
        CALL puts_acode;
        Block (Suc 0) 0x3f24 15;
        IF !ZF THEN
          Block (Suc 0) 0x3f27 16
        ELSE
          Block (Suc 0) 0x3f27 17
        FI
      OD;
      Block (Suc 0) 0x3f29 18;
      Block (Suc 0) 0x3f2e 19;
      CALL qemu_free_acode
    FI;
    Block 2 0x3f3b 20;
    IF !ZF THEN
      Block (Suc 0) 0x3f44 21;
      Block (Suc 0) 0x3f52 22;
      CALL stack_chk_fail_acode
    ELSE
      Block 6 0x3f51 23
    FI
  ›
```

```
schematic_goal dump_snapshots_0_8_0x3e90_0x3eae_0[blocks]:
  assumes ‹(P_0x3e90_0 && P_0x3e90_0_regions) σ›
  shows ‹exec_block 8 0x3eae 0 σ ≜ ?σ ∧ Q_0x3eb3_0 ?σ›
proof-
```

Apply Hoare logic to assembly

1. Extract control flow
2. Formulate a Hoare triple over each basic block

Basic block:

```
4f0: movsxd rdi, edi
4f3: mov rax, qword ptr [rsi+rdi*8-8]
4f8: movsx eax, byte ptr [rax]
4fb: ret
```

Introduction rule:

$$\frac{\begin{array}{l}\forall \sigma\ \sigma' \cdot P(\sigma) \wedge \text{symb\_exec}(a_0, a_1, \sigma, \sigma') \Longrightarrow Q(\sigma') \wedge \text{usage}(M(\sigma), \sigma, \sigma') \\ M' = \{\, r \mid \exists \sigma \cdot P(\sigma) \wedge r \in M(\sigma) \,\}\end{array}}{\{P\}\ \texttt{Block}\ a_0 \texttt{->} a_1\ \{Q; M'\}}$$

Formal Symbolic Execution requires memory reasoning

```
mov QWORD PTR [rsp-16], 1
mov DWORD PTR [rsp-24], 2
mov rax, QWORD PTR [rsp-16]
```

$$s' = s(\langle [\texttt{rsp} - 16] := \underset{64}{1}, [\texttt{rsp} - 24] := \underset{32}{2}, \texttt{RAX} := \underset{64}{1} \rangle)$$

requires a ***proof*** that two memory regions are separate:

$$\texttt{rsp} - 16 + 8 \leq \texttt{rsp} - 24 \vee \texttt{rsp} - 24 + 4 \leq \texttt{rsp} - 16$$

- Linear equations modulo 2^{64}
- Address computations can become very complicated
  - Multi-dimensional arrays
  - Structs
  - Pointer arithmetic

Basic block:

```
4f0: movsxd rdi, edi
4f3: mov rax, qword ptr [rsi+rdi*8-8]
4f8: movsx eax, byte ptr [rax]
4fb: ret
```

Introduction rule:

$$\frac{\forall \sigma\ \sigma' \ \cdot\ P(\sigma) \wedge \text{symb\_exec}(a_0, a_1, \sigma, \sigma') \Longrightarrow Q(\sigma') \wedge \text{usage}(M(\sigma), \sigma, \sigma') \qquad M' = \{\ r \mid \exists \sigma \ \cdot\ P(\sigma) \wedge r \in M(\sigma)\ \}}{\{P\}\ \texttt{Block}\ \mathtt{a_0}\texttt{->}\mathtt{a_1}\ \{Q; M'\}}$$

Hoare Triple for basic block:

$$MRR \Longrightarrow \{P\} f \{Q; M\}$$

where

$$\begin{aligned}
P &\equiv \mathtt{rip} = \mathtt{4f0} \wedge \mathtt{rsp} = \mathtt{rsp}_0 \wedge \ldots \wedge *[\mathtt{rsp}, 8] = \mathtt{ret_addr} \\
Q &\equiv \mathtt{rip} = \mathtt{ret_addr} \wedge \mathtt{rsp} = \mathtt{rsp}_0 + 8 \wedge \ldots \wedge *[\mathtt{rsp}_0, 8] = \mathtt{ret_addr} \\
M &\equiv \{\ [\mathtt{rsp}_0, 8], && (a) \\
&\quad\ [\mathtt{rsi}_0 + \text{sextend}\langle 31, 0\rangle \mathtt{rdi}_0 * 8 - 8, 8], && (b) \\
&\quad\ [*[\mathtt{rsi}_0 + \text{sextend}\langle 31, 0\rangle \mathtt{rdi}_0 * 8 - 8, 8], 1]\ \} && (c) \\
MRR &\equiv \begin{array}{lcl} \bowtie & = & \text{symmetric closure of } \{(a,b),(a,c),(b,c)\} \\ \sqsubseteq & = & \{\} \end{array}
\end{aligned}$$

Demo

```
definition dump_snapshots_acode :: ACode where
  ‹dump_snapshots_acode =
    Block 8 0x3eae 0;
    Block (Suc 0) 0x3eb3 (Suc 0);
    CALL bdrv_snapshot_list_acode;
    Block (Suc 0) 0x3eb8 2;
    IF jle THEN
      Block (Suc 0) 0x3eba 3
    ELSE
      Block 4 0x3ec5 4;
      Block (Suc 0) 0x3eca 5;
      CALL puts_acode;
      Block 6 0x3ee0 6;
      Block (Suc 0) 0x3ee2 7;
      CALL bdrv_snapshot_dump_acode;
      Block 2 0x3eee 8;
      Block (Suc 0) 0x3ef1 9;
      CALL puts_acode;
      Block (Suc 0) 0x3ef6 10;
      WHILE P_0x3f00_11 DO
        Block 5 0x3f10 11;
        Block (Suc 0) 0x3f17 12;
        CALL bdrv_snapshot_dump_acode;
        Block (Suc 0) 0x3f1c 13;
        Block (Suc 0) 0x3f1f 14;
        CALL puts_acode;
        Block (Suc 0) 0x3f24 15;
        IF !ZF THEN
          Block (Suc 0) 0x3f27 16
        ELSE
          Block (Suc 0) 0x3f27 17
        FI
      OD;
      Block (Suc 0) 0x3f29 18;
      Block (Suc 0) 0x3f2e 19;
      CALL qemu_free_acode
    FI;
    Block 2 0x3f3b 20;
    IF !ZF THEN
      Block (Suc 0) 0x3f44 21;
      Block (Suc 0) 0x3f52 22;
      CALL stack_chk_fail_acode
    ELSE
      Block 6 0x3f51 23
    FI
  ›
```

Apply Hoare logic to assembly

1. Extract control flow
2. Formulate a Hoare triple over each basic block
3. Compose proof over whole function using Hoare logic

Hoare Logic with Memory Usage

$$\frac{\{P\}\ f\ \{Q; M_1\} \qquad \{Q\}\ g\ \{R; M_2\} \qquad M = M_1 \cup M_2}{\{P\}\ f; g\ \{R; M\}}$$

(b) Sequence rule

$$\frac{\{P \wedge B\}\ f\ \{Q_1; M_1\} \qquad \{P \wedge \neg B\}\ g\ \{Q_2; M_2\} \qquad Q_1 \vee Q_2 \Longrightarrow Q \qquad M = M_1 \cup M_2}{\{P\}\ \texttt{If}\ B\ \texttt{Then}\ f\ \texttt{Else}\ g\ \texttt{Fi}\ \{Q; M\}}$$

(c) Conditional rule

$$\frac{\{I \wedge B\}\ f\ \{I'; M\} \qquad I' \Longrightarrow I \qquad I \wedge \neg B \Longrightarrow Q}{\{I\}\ \texttt{While}\ B\ \texttt{DO}\ f\ \texttt{OD}\ \{Q; M\}}$$

(d) While rule

$$\frac{M = \varnothing \qquad P \Longrightarrow Q}{\{P\}\ \texttt{Skip}\ \{Q; M\}}$$

(e) Skip rule

$$\frac{\forall 0 \le j \le n\ \cdot\ \{P\}\ a_j\ \{Q_j; M_j\} \qquad (\bigvee_{0 \le j \le n} Q_j) \Longrightarrow Q \qquad M = \bigcup_{0 \le j \le n} M_j}{\{P\}\ \texttt{Resume}\{(i_0, a_0), \ldots, (i_n, a_n)\}\ \{Q; M\}}$$

(f) Resume rule

```
0000000000003e90 <dump_snapshots>:
    3e90:      push   r12
    3e92:      push   rbp
    3e93:      push   rbx
    3e94:      sub    rsp,0x120
    3e9b:      mov    rax,QWORD PTR fs:0x28
    3ea4:      mov    QWORD PTR [rsp+0x118],rax
    3eac:      xor    eax,eax
    3eae:      lea    rsi,[rsp+0x8]
    3eb3:      call   19cc0 <bdrv_snapshot_list>
    3eb8:      test   eax,eax
    3eba:      jle    3f33 <dump_snapshots+0xa3>
    3ebc:      mov    ebx,eax
    3ebe:      lea    rdi,[rip+0x1914e
    3ec5:      lea    r12,[rsp+0x10]
    3eca:      call   3130 <puts@plt>
    3ecf:      lea    ebp,[rbx-0x1]
    3ed2:      xor    edx,edx
    3ed4:      mov    esi,0x100
    3ed9:      mov    rdi,r12
    3edc:      add    rbp,0x1
    3ee0:      xor    ebx,ebx
    3ee2:      call   19e60 <bdrv_snapshot_dump>
    3ee7:      imul   rbp,rbp,0x198
    3eee:      mov    rdi,rax
    3ef1:      call   3130 <puts@plt>
    3ef6:      nop    WORD PTR cs:[rax+rax*1+0x0]
    3f00:      mov    rdx,QWORD PTR [rsp+0x8]
    3f05:      mov    esi,0x100
    3f0a:      mov    rdi,r12
    3f0d:      add    rdx,rbx
    3f10:      add    rbx,0x198
    3f17:      call   19e60 <bdrv_snapshot_dump>
    3f1c:      mov    rdi,rax
    3f1f:      call   3130 <puts@plt>
    3f24:      cmp    rbx,rbp
    3f27:      jne    3f00 <dump_snapshots+0x70>
    3f29:      mov    rdi,QWORD PTR [rsp+0x8]
    3f2e:      call   5960 <qemu_free>
    3f33:      mov    rax,QWORD PTR [rsp+0x118]
    3f3b:      xor    rax,QWORD PTR fs:0x28
    3f44:      jne    3f52 <dump_snapshots+0xc2>
    3f46:      add    rsp,0x120
    3f4d:      pop    rbx
    3f4e:      pop    rbp
    3f4f:      pop    r12
    3f51:      ret
    3f52:      call   31c0 <__stack_chk_fail@plt>
    3f57:      nop    WORD PTR [rax+rax*1+0x0]
```

```
schematic_goal dump_snapshots:
  assumes a0
      and a1
      and a2
  shows ‹{{?P}} dump_snapshots_acode {{?Q;?M}}›
```

Demo

Conclusion

- Symbolic execution of x86-64 assembly in Isabelle/HOL
- Based on machine-learned instruction semantics
- Generate the proof code:
  - Generate assumptions on memory layout (using Z3 theorem prover)
  - Generate assumptions on called functions
  - Generate pre- and postconditions
  - Generate invariants
  - Generate information on memory usage (for compositionality)
- Final proof is manual in case of loops

Future Work

- Deal with indirect branching
- Combine with pointer-analysis
- Formally prove correctness of diversified binaries
- …