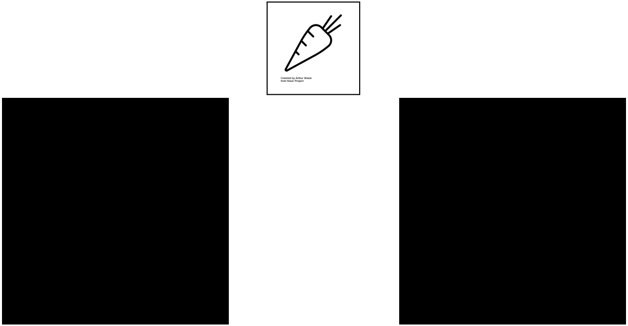
Tamarin: Concolic Disequivalence for MIPS

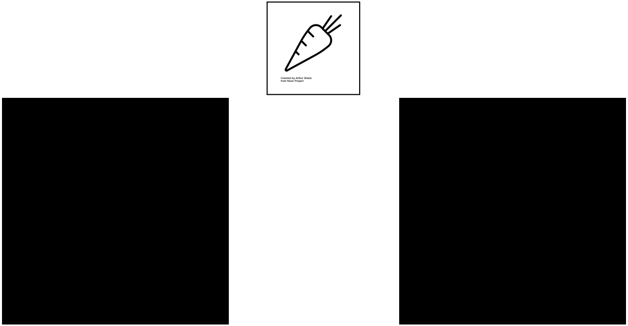
Abel Nieto

A Tale of Two Boxes

A Tale of Two Boxes



A Tale of Two Boxes



nom nom

A Tale of Two Boxes

nom nom

nom nom

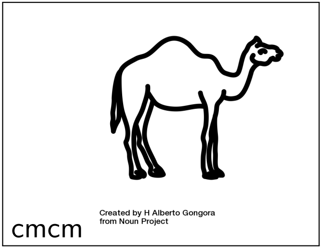
A Tale of Two Boxes

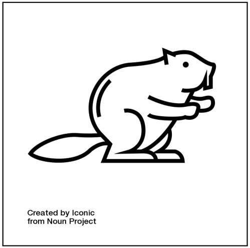
nom nom

=

nom nom

A Tale of Two Boxes





$$\neq$$

The Problem

Given MIPS program P_1 and P_2, when are they equivalent?

The Problem

Attempt 1: two programs are equivalent if they give the same output (resp.) for all inputs.

The Problem

Attempt 1: two programs are equivalent if they give the same output (resp.) for all inputs.

What's an input?

The Problem

Attempt 1: two programs are equivalent if they give the same output (resp.) for all inputs.

What's an input? Register \$1 and \$2.

The Problem

Attempt 1: two programs are equivalent if they give the same output (resp.) for all inputs.

What's an input? Register \$1 and \$2.
What's an output?

The Problem

Attempt 1: two programs are equivalent if they give the same output (resp.) for all inputs.

What's an input? Register \$1 and \$2.
What's an output? Register \$3.

The Problem

Attempt 1: two programs are equivalent if they give the same output (resp.) for all inputs.

What's an input? Register \$1 and \$2.
What's an output? Register \$3.

Don't care about (most) CPU interrupts/IO.

The Problem

Attempt 1: two programs are equivalent if they give the same output (resp.) for all inputs.

The Problem

Attempt 1: two programs are equivalent if they give the same output (resp.) for all inputs.

Problem: undecidable via Rice's theorem.

The Problem

Attempt 2: two programs are S-equivalent if they cannot be told apart after S steps.

The Problem

Attempt 2: two programs are S-equivalent if they cannot be told apart after S steps.

S-equivalent (e.g. for $S = 10$), but not equivalent:

```
# P_1
add $3, $1, $2
```

```
# P_2
  add $4, $0, 1   # counter
  add $5, $0, 42  # upper bound
loop:
  slt $6, $4, $5
  beq $6, $0, end
  add $4, $4, 1
  beq $0, $0, loop
end:
  add $3, $1, $1
```

The Problem

Attempt 2: two programs are S-equivalent if they **cannot be told apart** after S steps.

R_1	R_2	S-equiv

The Problem

Attempt 2: two programs are S-equivalent if they **cannot be told apart** after S steps.

R_1	R_2	S-equiv
v	v	yes

The Problem

Attempt 2: two programs are S-equivalent if they **cannot be told apart** after S steps.

R_1	R_2	S-equiv
v	v	yes
v	$w \neq v$	no

The Problem

Attempt 2: two programs are S-equivalent if they **cannot be told apart** after S steps.

R_1	R_2	S-equiv
v	v	yes
v	$w \neq v$	no
v	error	no

The Problem

Attempt 2: two programs are S-equivalent if they **cannot be told apart** after S steps.

R_1	R_2	S-equiv
v	v	yes
v	$w \neq v$	no
v	error	no
error	error	yes

The Problem

Attempt 2: two programs are S-equivalent if they **cannot be told apart** after S steps.

R_1	R_2	S-equiv
v	v	yes
v	$w \neq v$	no
v	error	no
error	error	yes
non-termination	???	yes
$\ldots$		

The Problem

Lemma

Equivalence implies S-equivalence.

The Problem

Lemma

Equivalence implies S-equivalence.

Corollary (Soundness)

If two programs are not S-equivalent (for any S), then they are not equivalent.

The Problem

Attempt 2: two programs are S-equivalent if they **cannot be told apart** after S steps.

Which inputs?

The Problem

Attempt 2: two programs are S-equivalent if they **cannot be told apart** after S steps.

Which inputs?

Try some inputs by hand: low coverage, fast (unit tests)

The Problem

Attempt 2: two programs are S-equivalent if they **cannot be told apart** after S steps.

Which inputs?

Try some inputs by hand: low coverage, fast (unit tests)

Try all 2^{64}values of $1 and $2: high coverage, slow (but decidable)

The Problem

Attempt 2: two programs are S-equivalent if they **cannot be told apart** after S steps.

Which inputs?

Try some inputs by hand: low coverage, fast (unit tests)

Try all 2^{64}values of $1 and $2: high coverage, slow (but decidable)

Tamarin: use concolic execution: higher coverage(?), not too slow(?)

Idea

Alternating concolic execution

```
# P_1
  bne $1, 42, end
  add $3, $3, $0
end:
  add $3, $1, $2
```

```
# P_2
  add $3, $1, $2
  bne $2, 100, end
  add $3, $3, $2
end:
```

Idea

Alternating concolic execution

```
# P_1
  bne $1, 42, end
  add $3, $3, $0
end:
  add $3, $1, $2
```

```
# P_2
  add $3, $1, $2
  bne $2, 100, end
  add $3, $3, $2
end:
```

Run	Driver	Verifier	\$1	\$2	Path	R_D	R_V

Idea

Alternating concolic execution

```
# P_1
  bne $1, 42, end
  add $3, $3, $0
end:
  add $3, $1, $2
```

```
# P_2
  add $3, $1, $2
  bne $2, 100, end
  add $3, $3, $2
end:
```

Run	Driver	Verifier	\$1	\$2	Path	R_D	R_V
1	P_1	P_2	1	1	$\$1 \neq 42$	2	2

Idea

Alternating concolic execution

```
# P_1
  bne $1, 42, end
  add $3, $3, $0
end:
  add $3, $1, $2
```

```
# P_2
  add $3, $1, $2
  bne $2, 100, end
  add $3, $3, $2
end:
```

Run	Driver	Verifier	\$1	\$2	Path	R_D	R_V
1	P_1	P_2	1	1	$\$1 \neq 42$	2	2
2	P_2	P_1	1	1	$\$2 \neq 100$	2	2

Idea

Alternating concolic execution

```
# P_1
  bne $1, 42, end
  add $3, $3, $0
end:
  add $3, $1, $2
```

```
# P_2
  add $3, $1, $2
  bne $2, 100, end
  add $3, $3, $2
end:
```

Run	Driver	Verifier	\$1	\$2	Path	R_D	R_V
1	P_1	P_2	1	1	$\$1 \neq 42$	2	2
2	P_2	P_1	1	1	$\$2 \neq 100$	2	2
3	P_1	P_2	42	1	$\$1 = 42$	2	2

Idea

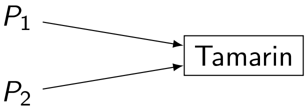
Alternating concolic execution

```
# P_1
  bne $1, 42, end
  add $3, $3, $0
end:
  add $3, $1, $2
```

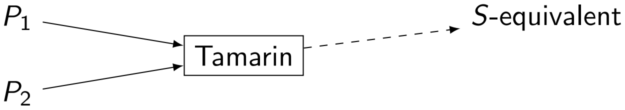
```
# P_2
  add $3, $1, $2
  bne $2, 100, end
  add $3, $3, $2
end:
```

Run	Driver	Verifier	\$1	\$2	Path	R_D	R_V
1	P_1	P_2	1	1	$\$1 \neq 42$	2	2
2	P_2	P_1	1	1	$\$2 \neq 100$	2	2
3	P_1	P_2	42	1	$\$1 = 42$	2	2
4	P_2	P_1	1	100	$\$2 = 100$	201	2

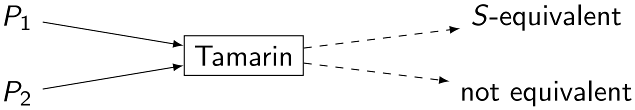
Tamarin (Overview)



Tamarin (Overview)



Tamarin (Overview)



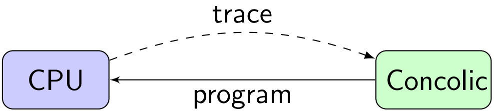
Tamarin (Overview)

Concolic

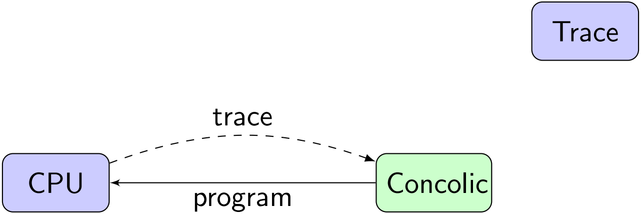
Tamarin (Overview)

Tamarin (Overview)

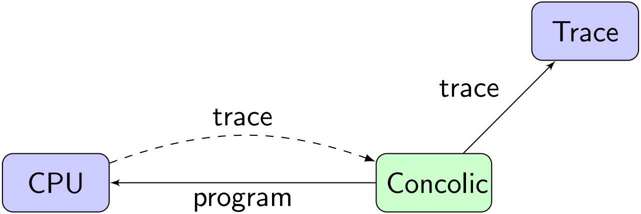
Tamarin (Overview)



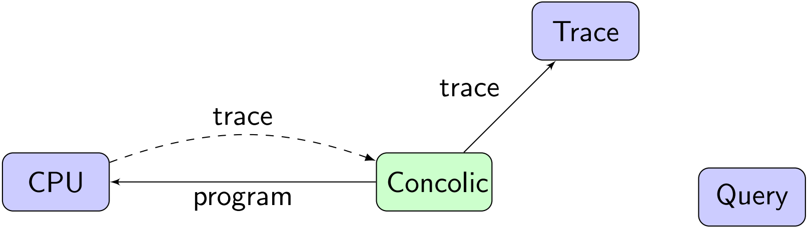
Tamarin (Overview)



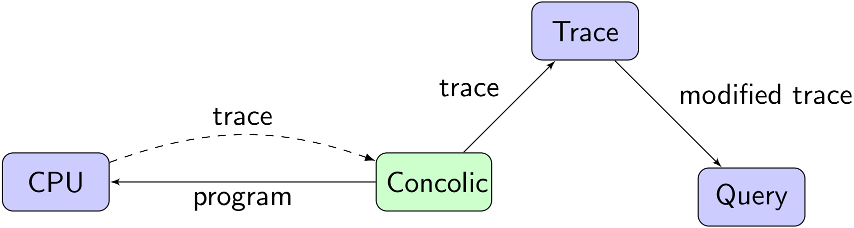
Tamarin (Overview)



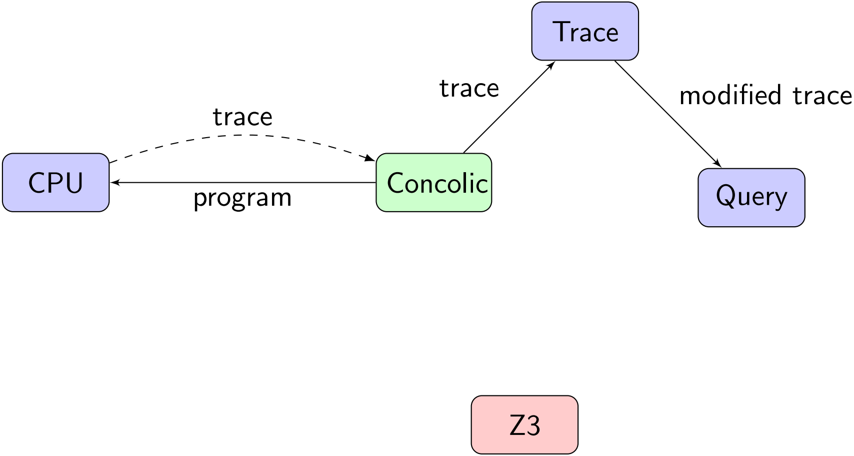
Tamarin (Overview)



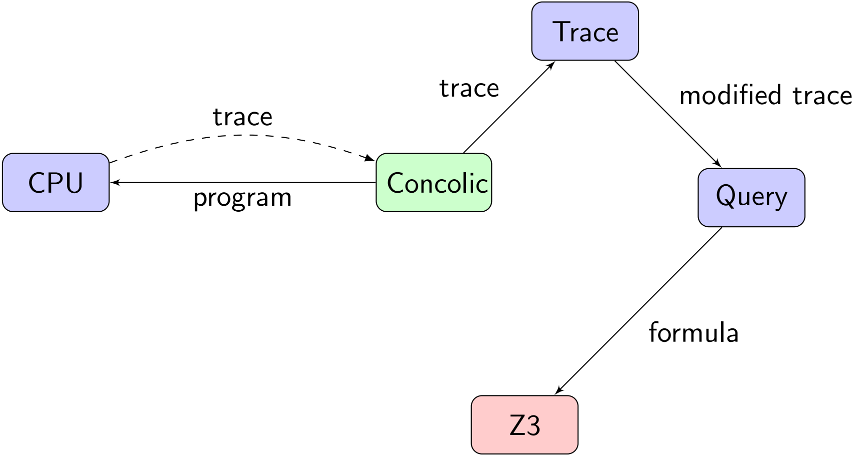
Tamarin (Overview)



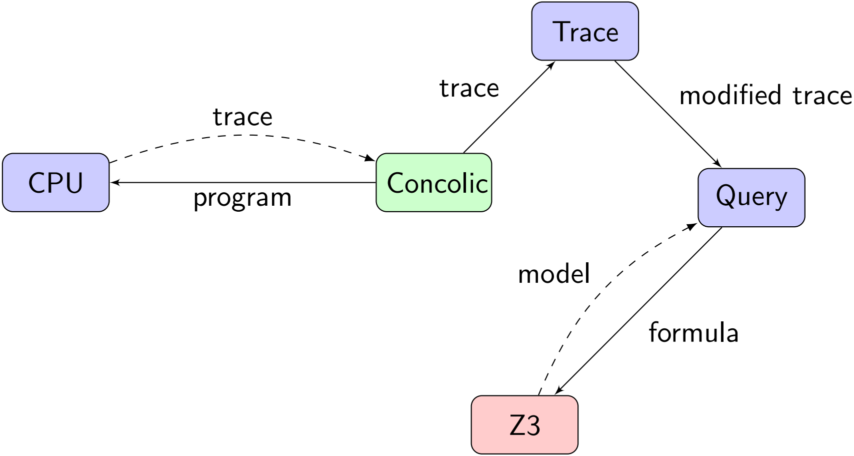
Tamarin (Overview)



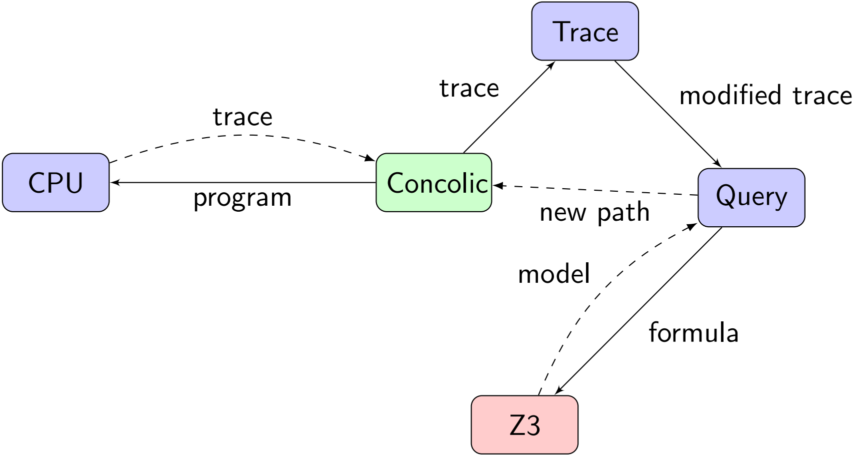
Tamarin (Overview)



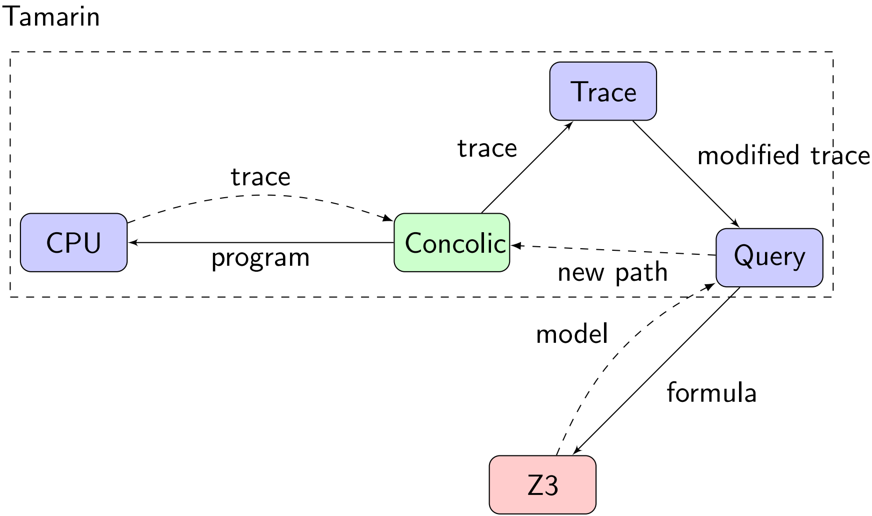
Tamarin (Overview)



Tamarin (Overview)



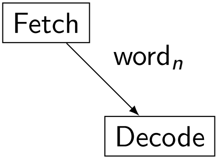
Tamarin (Overview)



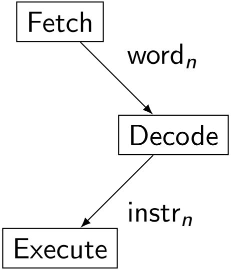
CPU

$\boxed{\text{Fetch}}$

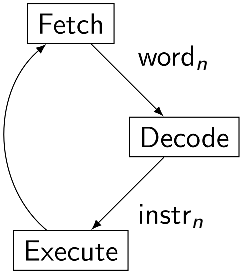
CPU



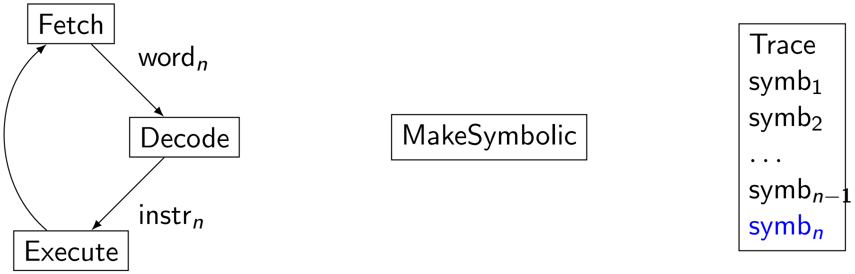
CPU



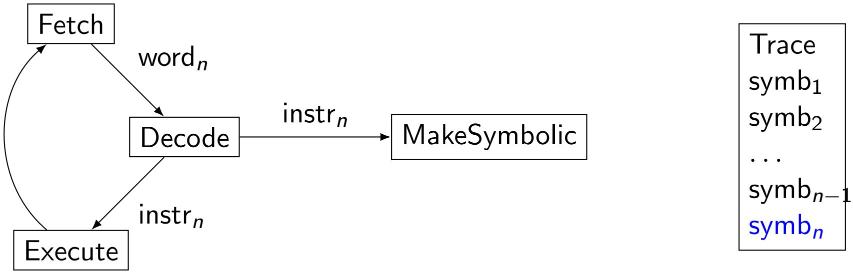
CPU



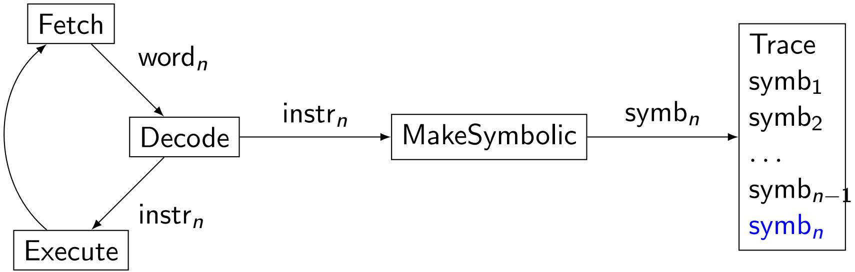
CPU


Fetch
word$_n$
Decode
instr$_n$
Execute
MakeSymbolic
Trace
symb$_1$
symb$_2$
...
symb$_{n-1}$
symb$_n$

CPU


Fetch
word$_n$
Decode
instr$_n$
MakeSymbolic
instr$_n$
Execute
Trace
symb$_1$
symb$_2$
...
symb$_{n-1}$
symb$_n$

CPU


Fetch
word$_n$
Decode
instr$_n$
MakeSymbolic
symb$_n$
Execute
instr$_n$
Trace
symb$_1$
symb$_2$
...
symb$_{n-1}$
symb$_n$

CPU (MakeSymbolic)

Instruction	Symbolic

CPU (MakeSymbolic)

Instruction	Symbolic
`add $3, $1, $2`	$r_3 \leftarrow r_1 + r_2$

CPU (MakeSymbolic)

Instruction	Symbolic
`add $3, $1, $2`	$r_3 \leftarrow r_1 + r_2$
`beq $1, $2, label`	$r_1 = r_2$ or $r_1 \neq r_2$
`add $3, $pc, $0`	$r_3 \leftarrow$ 0x8BADF00D
`lis $3; 42`	$r_3 \leftarrow 42$

Trace

$$T \longrightarrow \boxed{\text{Trace}} \longrightarrow T'$$

Trace

$$T \longrightarrow \boxed{\text{Trace}} \longrightarrow T'$$

```
Seq[Trace => Trace]
```

Trace

$$T \longrightarrow \boxed{\text{Trace}} \longrightarrow T'$$

`Seq[Trace => Trace]`

- Desugar

Trace

$$T \longrightarrow \boxed{\text{Trace}} \longrightarrow T'$$

`Seq[Trace => Trace]`

- Desugar
- Simplify

Trace

$$T \longrightarrow \boxed{\text{Trace}} \longrightarrow T'$$

`Seq[Trace => Trace]`

- Desugar
- Simplify
- Trim

Trace

$$T \longrightarrow \boxed{\text{Trace}} \longrightarrow T'$$

`Seq[Trace => Trace]`

- Desugar
- Simplify
- Trim
- SSA convert

Trace

Desugar

Trace

Desugar

`Mult(s, t)` $\longrightarrow$ `Mult64(tmp, s, t); Low32(lo, tmp); High32(hi, tmp)`

Trace

Desugar

`Mult(s, t)` $\longrightarrow$ `Mult64(tmp, s, t); Low32(lo, tmp); High32(hi, tmp)`

Simplify

Trace

Desugar

`Mult(s, t)` $\longrightarrow$ `Mult64(tmp, s, t); Low32(lo, tmp); High32(hi, tmp)`

Simplify

`beq $0, $0, label` $\longrightarrow \emptyset$

Trace

Desugar

`Mult(s, t)` $\longrightarrow$ `Mult64(tmp, s, t); Low32(lo, tmp); High32(hi, tmp)`

Simplify

`beq $0, $0, label` $\longrightarrow \emptyset$

Trim

Trace

Desugar

`Mult(s, t)` $\longrightarrow$ `Mult64(tmp, s, t); Low32(lo, tmp); High32(hi, tmp)`

Simplify

`beq $0, $0, label` $\longrightarrow \emptyset$

Trim

limit trace to D path conditions

Trace

Desugar

`Mult(s, t)` $\longrightarrow$ `Mult64(tmp, s, t); Low32(lo, tmp); High32(hi, tmp)`

Simplify

`beq $0, $0, label` $\longrightarrow \emptyset$

Trim

limit trace to D path conditions

SSA convert

Trace (SSA convert)

Trace	Incorrect	Correct

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`		

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`		

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`	$x = x - y$	

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`	$x = x - y$	$x_2 = x_1 - y_1$

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`	$x = x - y$	$x_2 = x_1 - y_1$
`Add($2, $1, $2)`		

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`	$x = x - y$	$x_2 = x_1 - y_1$
`Add($2, $1, $2)`	$y = x + y$	

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`	$x = x - y$	$x_2 = x_1 - y_1$
`Add($2, $1, $2)`	$y = x + y$	$y_2 = x_2 + y_1$

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`	$x = x - y$	$x_2 = x_1 - y_1$
`Add($2, $1, $2)`	$y = x + y$	$y_2 = x_2 + y_1$

‘

$$\$3 = 9 \land \$1 = 3$$

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`	$x = x - y$	$x_2 = x_1 - y_1$
`Add($2, $1, $2)`	$y = x + y$	$y_2 = x_2 + y_1$

‘

$$\$3 = 9 \wedge \$1 = 3$$

$$z = 9 \wedge x = 3$$

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`	$x = x - y$	$x_2 = x_1 - y_1$
`Add($2, $1, $2)`	$y = x + y$	$y_2 = x_2 + y_1$

'

$\$3 = 9 \wedge \$1 = 3$

$z = 9 \wedge x = 3$ (no sol)

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`	$x = x - y$	$x_2 = x_1 - y_1$
`Add($2, $1, $2)`	$y = x + y$	$y_2 = x_2 + y_1$

‘

$$\$3 = 9 \wedge \$1 = 3$$

$$z = 9 \wedge x = 3 \text{ (no sol)}$$

$$z_1 = 9 \wedge x_2 = 3$$

Trace (SSA convert)

Trace	Incorrect	Correct
`Add($3, $1, $2)`	$z = x + y$	$z_1 = x_1 + y_1$
`Sub($1, $1, $2)`	$x = x - y$	$x_2 = x_1 - y_1$
`Add($2, $1, $2)`	$y = x + y$	$y_2 = x_2 + y_1$

‘

$\$3 = 9 \wedge \$1 = 3$

$z = 9 \wedge x = 3$ (no sol)

$z_1 = 9 \wedge x_2 = 3$ ($x_1 = 6 \wedge y_1 = 3$)

Query

$$\text{trace} \longrightarrow \boxed{\text{Query}} \longrightarrow \text{formula}$$

Query

trace:

```
add $3, $1, $2
slt $4, $1, $2
add $5, $1, $0
$4 != $5
```

Query

trace:

```
add $3, $1, $2
slt $4, $1, $2
add $5, $1, $0
$4 != $5
```

formula (SMT-LIB):

```
(declare-const r0 (_ BitVec 32))
(declare-const r1 (_ BitVec 32))
...
```

Query

trace:

```
add $3, $1, $2
slt $4, $1, $2
add $5, $1, $0
$4 != $5
```

formula (SMT-LIB):

```
(declare-const r0 (_ BitVec 32))
(declare-const r1 (_ BitVec 32))
...
(assert (= r0 (_ bv0 32)))
```

Query

trace:

```
add $3, $1, $2
slt $4, $1, $2
add $5, $1, $0
$4 != $5
```

formula (SMT-LIB):

```
(declare-const r0 (_ BitVec 32))
(declare-const r1 (_ BitVec 32))
...
(assert (= r0 (_ bv0 32)))
(assert (= r3 (bvadd r1 r2)))
```

Query

trace:

```
add $3, $1, $2
slt $4, $1, $2
add $5, $1, $0
$4 != $5
```

formula (SMT-LIB):

```
(declare-const r0 (_ BitVec 32))
(declare-const r1 (_ BitVec 32))
...
(assert (= r0 (_ bv0 32)))
(assert (= r3 (bvadd r1 r2)))
(assert
  (= r4 (ite (bvslt r1 r2)
    (_ bv1 32)
    (_ bv0 32))))
```

Query

trace:

```
add $3, $1, $2
slt $4, $1, $2
add $5, $1, $0
$4 != $5
```

formula (SMT-LIB):

```
(declare-const r0 (_ BitVec 32))
(declare-const r1 (_ BitVec 32))
...
(assert (= r0 (_ bv0 32)))
(assert (= r3 (bvadd r1 r2)))
(assert
  (= r4 (ite (bvslt r1 r2)
    (_ bv1 32)
    (_ bv0 32))))
(assert (= r5 (_ bv1 32)))
```

Query

trace:

```
add $3, $1, $2
slt $4, $1, $2
add $5, $1, $0
$4 != $5
```

formula (SMT-LIB):

```
(declare-const r0 (_ BitVec 32))
(declare-const r1 (_ BitVec 32))
...
(assert (= r0 (_ bv0 32)))
(assert (= r3 (bvadd r1 r2)))
(assert
  (= r4 (ite (bvslt r1 r2)
    (_ bv1 32)
    (_ bv0 32))))
(assert (= r5 (_ bv1 32)))
(assert (not (= r4 r5)))
```

Query

trace:

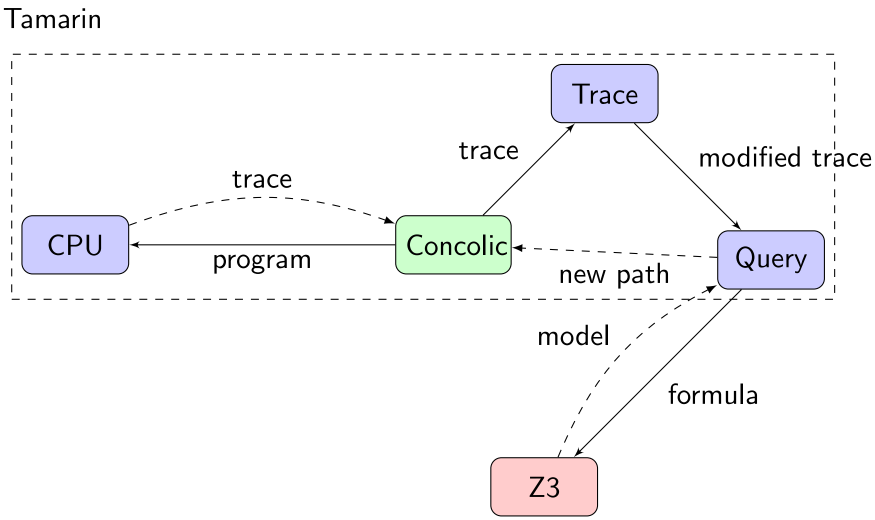
```
add $3, $1, $2
slt $4, $1, $2
add $5, $1, $0
$4 != $5
```

formula (SMT-LIB):

```
(declare-const r0 (_ BitVec 32))
(declare-const r1 (_ BitVec 32))
...
(assert (= r0 (_ bv0 32)))
(assert (= r3 (bvadd r1 r2)))
(assert
  (= r4 (ite (bvslt r1 r2)
    (_ bv1 32)
    (_ bv0 32))))
(assert (= r5 (_ bv1 32)))
(assert (not (= r4 r5)))

(check-sat) (get-model)
```

Tamarin (Recap)



Demo time

Conclusions

Program equivalence is tricky. Program equivalence for assembly programs is even trickier.

Conclusions

Program equivalence is tricky. Program equivalence for assembly programs is even trickier.

Alternating concolic execution might be an effective way to tackle the problemb (surprinsingly, not done before?).

Conclusions

Program equivalence is tricky. Program equivalence for assembly programs is even trickier.

Alternating concolic execution might be an effective way to tackle the problemb (surprinsingly, not done before?).

Need experiments and to eliminate restrictions before we can say if useful for real-world programs.

Conclusions

Program equivalence is tricky. Program equivalence for assembly programs is even trickier.

Alternating concolic execution might be an effective way to tackle the problemb (surprinsingly, not done before?).

Need experiments and to eliminate restrictions before we can say if useful for real-world programs.

The whole field is not just theory. Z3 actually works! (and it's not hard to integrate with)