Unicorn

Bit-Precise Reasoning for Symbolic Execution and Code Optimization

Michael Starzinger

University of Salzburg
michael.starzinger@cs.uni-salzburg.at

May 30, 2023

Joint work with Christoph Kirsch, Daniel Kocher, and Stefanie Muroya Lei

One Sentence

Unicorn creates bit-precise yet *polynomial-sized* models of programs using *bounded symbolic execution* to reason about safety properties and optimizing transformations by *fully utilizing* modern SMT/SAT solvers.

Motivating Example

```
int main() {
  uint64_t a = read(1);
  uint64_t b = read(1);

  // no trivial solution
  if (a < 2) return 0;
  if (b < 2) return 0;

  // semi-prime: 5 * 7
  if (a * b == 35)
    return 1;

  return 0;
}
```

Motivating Example

```
int main() {
  uint64_t a = read(1);
  uint64_t b = read(1);

  // no trivial solution
  if (a < 2) return 0;
  if (b < 2) return 0;

  // semi-prime: 5 * 7
  if (a * b == 35)
    return 1;

  return 0;
}
```

- all input *symbolic* (`read(·)`)

Motivating Example

```
int main() {
  uint64_t a = read(1);
  uint64_t b = read(1);

  // no trivial solution
  if (a < 2) return 0;
  if (b < 2) return 0;

  // semi-prime: 5 * 7
  if (a * b == 35)
    return 1;

  return 0;
}
```

- all input *symbolic* (`read(·)`)
- program has *safety properties* (here inverse: *bad states*)
  - division by zero
  - out-of-bounds accesss
  - non-zero exit code

Motivating Example

```
int main() {
  uint64_t a = read(1);
  uint64_t b = read(1);

  // no trivial solution
  if (a < 2) return 0;
  if (b < 2) return 0;

  // semi-prime: 5 * 7
  if (a * b == 35)
    return 1;

  return 0;
}
```

- all input *symbolic* (`read(·)`)
- program has *safety properties* (here inverse: *bad states*)
  - division by zero
  - out-of-bounds accesss
  - non-zero exit code

$\Rightarrow$ Evidential Programming

Bounded Symbolic Execution

Bounded Execution: $E_p(s, n, i)$

We say that $E_p : \mathbb{S} \times \mathbb{N} \times \mathbb{I} \rightarrow \mathbb{S}$ performs *bounded execution* of p if $E_p(s, n, i)$ is the machine state after executing exactly $n \in \mathbb{N}$ instructions in p starting from machine state $s \in \mathbb{S}$ under the program input $i \in \mathbb{I}$.

Bounded Symbolic Execution

Bounded Execution: $E_p(s, n, i)$

We say that $E_p : \mathbb{S} \times \mathbb{N} \times \mathbb{I} \to \mathbb{S}$ performs *bounded execution* of p if $E_p(s, n, i)$ is the machine state after executing exactly $n \in \mathbb{N}$ instructions in p starting from machine state $s \in \mathbb{S}$ under the program input $i \in \mathbb{I}$.

Symbolic Encoding: $U_{p,n}(s_a, s_b, i)$

We encode E_p as a function $U_{p,n} : \mathbb{S} \times \mathbb{S} \times \mathbb{I} \to \mathbb{B}$ such that, for all $s_a, s_b \in \mathbb{S}$ and for all $i \in \mathbb{I}$, $U_{p,n}(s_a, s_b, i)$ is true iff $s_b = E_p(s_a, n, i)$.

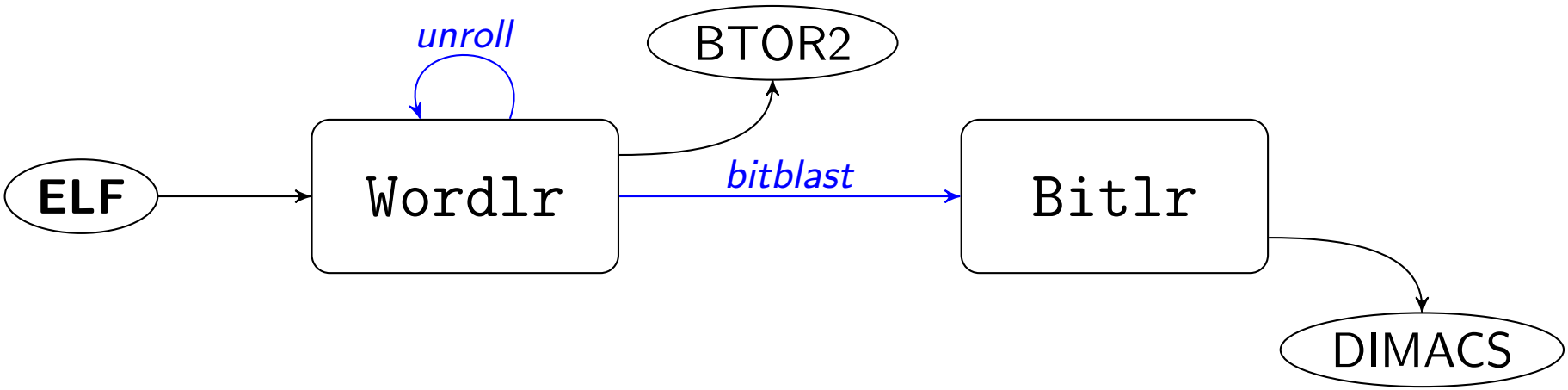
Tool Implementation Notes

- Unicorn is written in Rust (currently ∼12k LoC)
- Integrates state-of-the-art SMT/SAT solvers
  - any SMT-LIB compatible solver
    theory: bit vectors (`QF_BV`), arrays (`QF_ABV`, optional)
  - any SAT solver consuming CNF

Tool Implementation Notes

- Unicorn is written in Rust (currently ∼12k LoC)
- Integrates state-of-the-art SMT/SAT solvers
  - any SMT-LIB compatible solver
    theory: bit vectors (`QF_BV`), arrays (`QF_ABV`, optional)
  - any SAT solver consuming CNF
- Input program given in compiled format (no source needed)
  - operates on machine-level (models RISC-V bit-precisely)
- Intermediate Representations (IR): `Wordlr` and `Bitlr`

Architecture Overview



- Input: RISC-V binary in ELF with `rv64imc` (e.g. unmodified GCC)
- Word-Level: binding to Boolector and Z3; or write BTOR2 files
- Bit-Level: binding to Kissat and CaDiCaL; or write DIMACS files

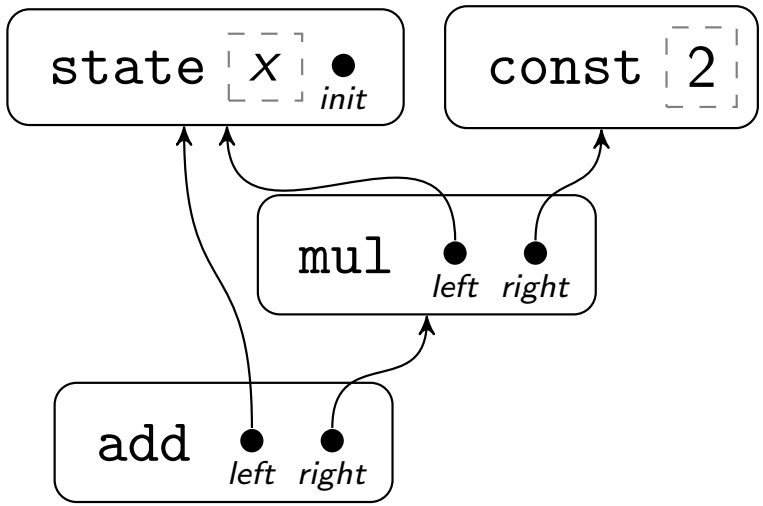
Intermediate Representation

- Wordlr Combinational Graph
  - operator semantics: `QF_ABV`
  - bit-precise mapping: `rv64imc`
  - data/control dependency DAG

Intermediate Representation

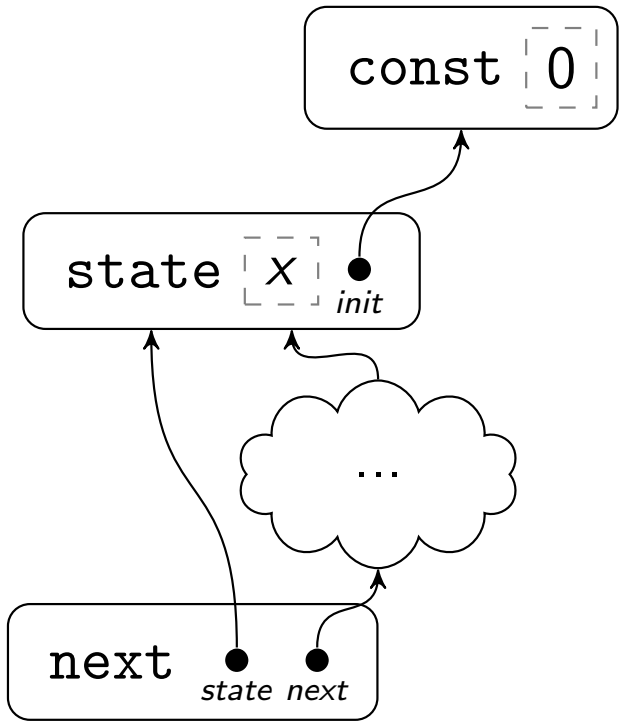
- Wordlr Combinational Graph
  - operator semantics: `QF_ABV`
  - bit-precise mapping: `rv64imc`
  - data/control dependency DAG

$\Rightarrow$ Consider: `x * 2 + x`



Intermediate Representation

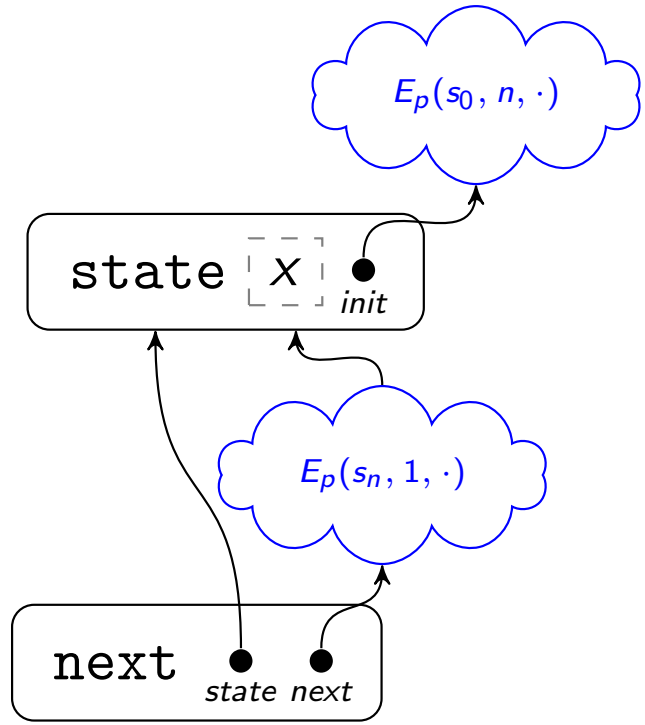
- Wordlr Combinational Graph
  - operator semantics: `QF_ABV`
  - bit-precise mapping: `rv64imc`
  - data/control dependency DAG
- Encodes effects on state (FSM)
  - transition function $E_p(s_0, 1, \cdot)$
  - can be *unrolled* and optimized
  - inspired by `BTOR2` format



$\Rightarrow$ Size: $O(|p|)$

Intermediate Representation

- Wordlr Combinational Graph
  - operator semantics: `QF_ABV`
  - bit-precise mapping: `rv64imc`
  - data/control dependency DAG
- Encodes effects on state (FSM)
  - transition function $E_p(s_0, 1, \cdot)$
  - can be *unrolled* and optimized
  - inspired by `BTOR2` format



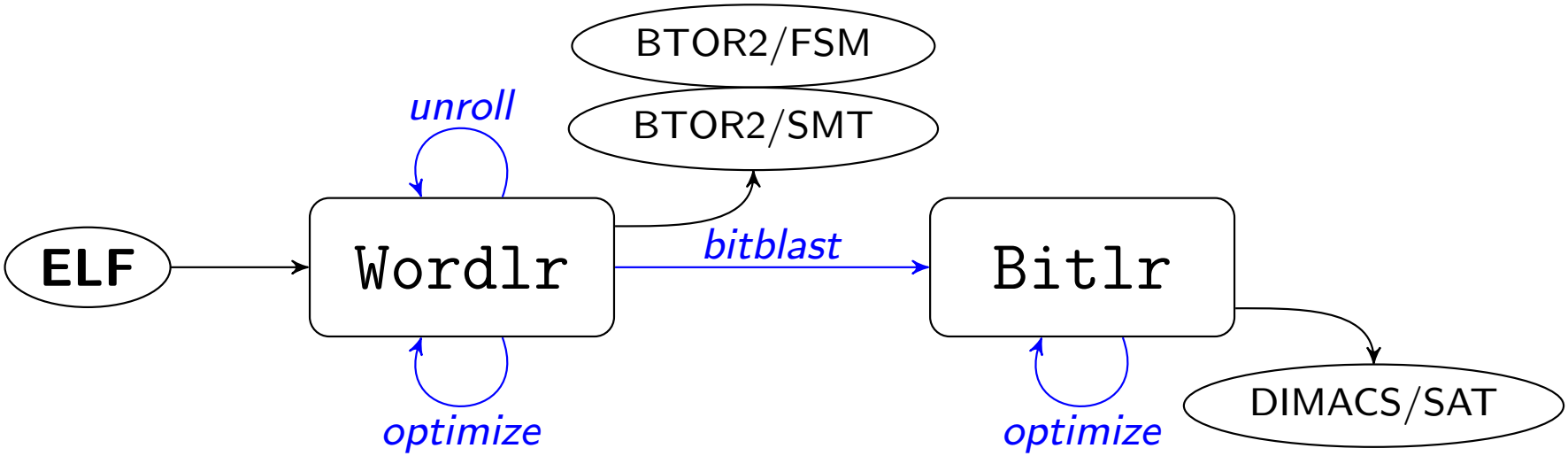
$\Rightarrow$ Size: $O(n * |p|)$

$$E_p(s_0, n+1, \cdot) = E_p(E_p(s_0, n, \cdot), 1, \cdot)$$

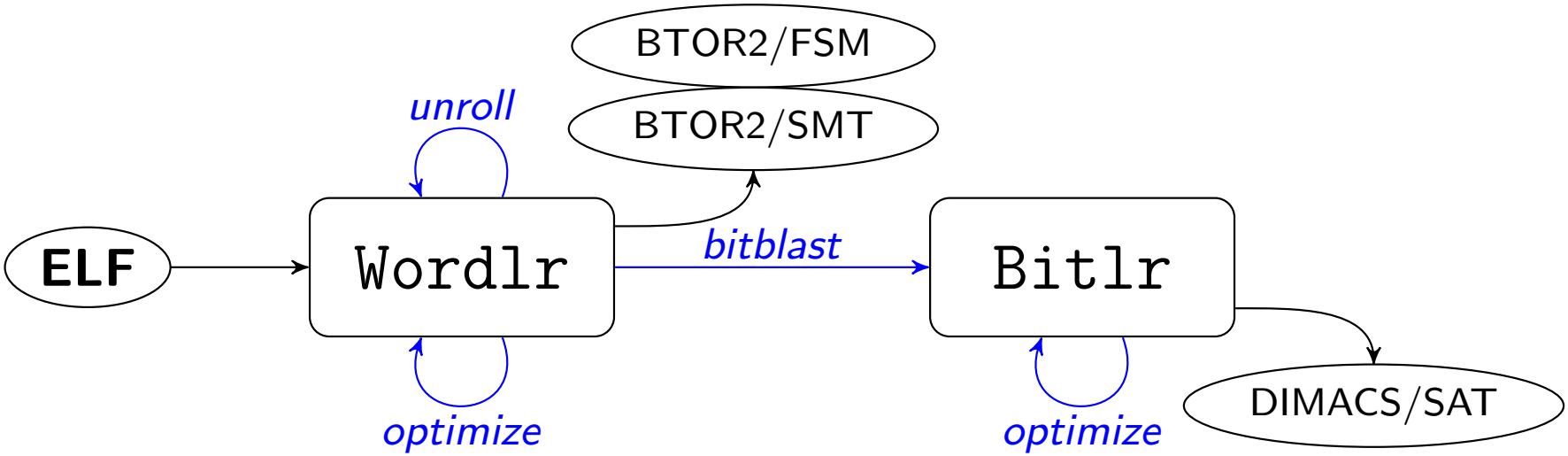
Intermediate Representation

- Wordlr Combinational Graph
  - operator semantics: `QF_ABV`
  - bit-precise mapping: `rv64imc`
  - data/control dependency DAG
- Encodes effects on state (FSM)
  - transition function $E_p(s_0, 1, \cdot)$
  - can be *unrolled* and optimized
  - inspired by `BTOR2` format
- Models RISC-V machine state
  - registers: $32 \times BV_{[64]}$
  - PC-flags: $|p| \times BV_{[1]}$
  - memory: $A[BV_{[64]}, BV_{[64]}]$

Architecture Overview



Architecture Overview



Generated Representation	Format	Complexity
Wordlr graph for p (unbounded)	BTOR2/FSM	$\mathcal{O}(\|p\|)$
as above, w/o array logic	BTOR2/FSM	$\mathcal{O}(m \cdot \|p\|)$
Wordlr graph unrolled to given n	BTOR2/SMT	$\mathcal{O}(n \cdot \|p\|)$
as above, w/o array logic	BTOR2/SMT	$\mathcal{O}(n \cdot m \cdot \|p\|)$
Bitlr graph converted to CNF	DIMACS/SAT	$\mathcal{O}(n \cdot m \cdot \|p\| \cdot w^2)$

Optimization and Minimization

- **Constant Folding**: all *concrete* operations can be folded
- **SMT Solvers**: remaining *symbolic* operations (result bit-width 1)
- **SAT Solvers**: bits in bit-blasted *symbolic* operations (not done)

- Combinational graph fragments map directly to SMT-LIB queries

Optimization and Minimization

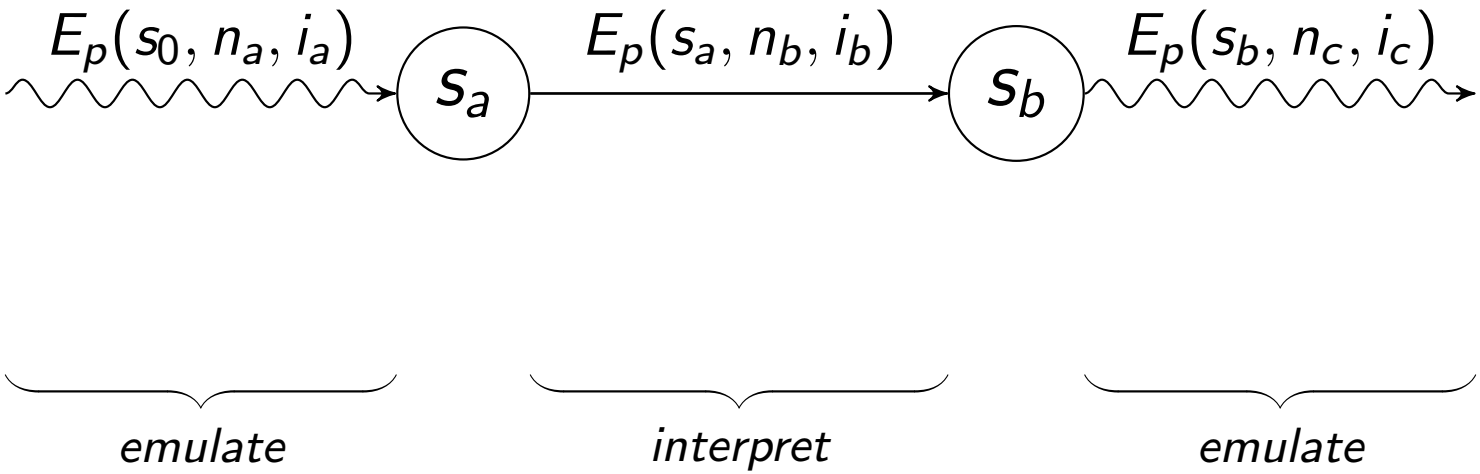
- **Constant Folding**: all *concrete* operations can be folded
- **SMT Solvers**: remaining *symbolic* operations (result bit-width 1)
- **SAT Solvers**: bits in bit-blasted *symbolic* operations (not done)

- Combinational graph fragments map directly to SMT-LIB queries

Graph Node	Operator Semantics	Transformations
$z :=$ `ult` x, y	unsigned less than: $x <_u y$	$z \rightsquigarrow 0$, if $\mathit{unsat}(x <_u y)$ $z \rightsquigarrow 1$, if $\mathit{unsat}(x \geq_u y)$ $z \rightsquigarrow \mathit{eval}(x <_u y)$, if x and y constant
$\vdots$	$\vdots$	$\vdots$

Emulation vs. Interpretation

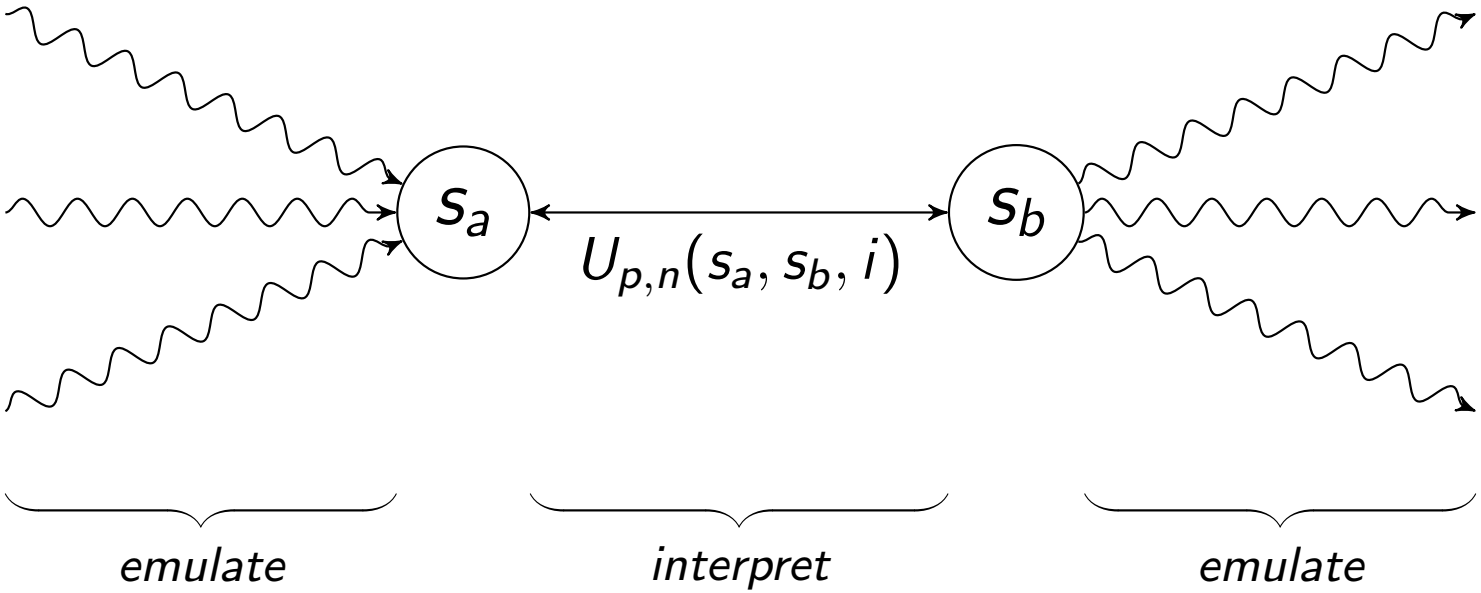
- Switch between RISC-V program and bounded execution model:



- **emulate**: instruction emulation of $E_p(s, n, i)$ on concrete s and i
- **interpret**: evaluate optimized model $U_{p,n}(s, \cdot, i)$ on concrete s and i

Emulation vs. Interpretation

- Switch between RISC-V program and bounded execution model:



- **emulate**: instruction emulation of $E_p(s, n, i)$ on concrete s and i
- **interpret**: evaluate optimized model $U_{p,n}(s, \cdot, i)$ on concrete s and i

Future Work

- Insert code to evaluate model into input program
- Synthesize new RISC-V machine code from model
- Challenges and open questions:
  - selecting good (optimal) instructions during synthesis
  - choose n when multiple paths of differing length exist

Thank You!

Backup Slide: Output of Motivating Example

```
// Simple example testing if
// two bytes when multiplied
// yield the semiprime 35.

uint64_t main() {
  uint64_t a;
  uint64_t b;

  // read program inputs
  a = read_bytes(1);
  b = read_bytes(1);

  // exclude '1' as factor
  a = a + 2;
  b = b + 2;

  // semiprime: 5 * 7
  if (a * b == 35)
    return 1;

  return 0;
}
```

```
1 sort bitvec 1 ; Boolean
2 sort bitvec 64 ; 64-bit
3 sort bitvec 8 ; 1 byte

1000 constd 2 35
1001 state 3 input[n=80]
1002 uext 2 1001 56
1003 constd 2 2
1004 add 2 1002 1003
1005 state 3 input[n=150]
1006 uext 2 1005 56
1007 constd 2 2
1008 add 2 1006 1007
1009 mul 2 1004 1008
1010 sub 2 1000 1009
1011 constd 2 1
1012 ult 1 1010 1011
1013 uext 2 1012 63
1014 constd 2 0
1015 eq 1 1013 1014
1016 not 1 1015
1017 bad 1016 exit-code[n=198]
```

Backup Slide: Experiments and Performance Evaluation

Name	n	$\#sat$	BtorMC	Unicorn Boolector			Unicorn Z3			Unicorn Kissat			Unicorn CaDiCal			Kissat	KLEE
				A	R	1	A	R	1	A	R	1	A	R	1		
empty (sat)	696	1	3.79	4.72	4.85	4.80	4.78	4.78	4.78	4.73	4.78	4.84	4.71	4.68	4.89	0.002	0.02
empty (unsat)	812	0	4.51	5.52	5.68	5.51	5.50	5.61	5.52	5.61	5.70	5.48	5.59	5.49	5.57	0.001	0.01
div-zero	698	1	3.84	4.88	4.90	4.76	4.74	4.85	4.84	4.79	4.77	4.87	4.83	4.68	4.76	0.001	0.02
assignment	703	255	4.05	4.85	4.90	4.91	4.81	4.81	4.98	4.97	4.91	4.90	4.89	4.99	4.92	0.002	0.04
invalid-access	920	1	5.06	6.86	6.61	6.71	6.81	6.59	6.62	6.51	6.49	6.50	6.69	6.45	6.47	0.003	0.03
dynamic-memory	920	255	5.43	6.45	6.29	6.31	6.24	6.43	6.62	6.30	6.31	6.41	6.34	6.40	6.27	0.002	0.75
factorize (35)	765	2	4.50	5.51	5.35	5.33	5.42	5.29	5.28	5.48	5.33	5.28	5.23	5.28	5.21	0.002	0.05
factorize (8bit)	766	2	4.44	5.30	5.31	5.32	5.36	5.39	5.43	5.36	5.27	5.33	5.26	5.24	5.37	0.002	0.05
factorize (16bit)	766	2	5.17	5.67	5.57	5.69	7.22	7.17	6.09	5.29	5.27	5.57	5.57	5.69	5.39	0.090	0.24
factorize (32bit)	765	2	11.76	7.39	7.51	7.24	23.75	23.54	30.60	7.65	7.39	6.07	17.23	17.42	15.62	2.875	154.98
factorize (32bit/easy)	767	119	4.49	5.44	5.54	5.62	8.43	8.60	8.28	5.40	5.36	5.45	5.51	5.39	5.27	0.034	28.94
simple-if	712	1	18.64	6.00	5.54	5.59	timeout			9.87	7.18	7.94	8.75	7.38	7.63	0.569	0.02
simple-if (reverse)	712	1	19.18	5.70	5.61	5.67	timeout			6.61	5.64	5.78	6.14	5.79	5.83	0.281	0.02
simple-if (w/o else)	712	1	18.97	5.65	5.65	5.84	timeout			6.37	5.91	5.56	6.17	5.70	5.83	0.553	0.02
nested-if	716	1	36.66	6.60	6.38	6.54	timeout			51.15	8.56	10.07	9.98	8.82	9.17	6.718	0.03
nested-if (reverse)	715	1	5.27	5.77	5.79	5.90	timeout			7.72	5.86	6.08	6.08	5.81	6.02	0.388	0.03
loop	718	1	11.91	5.74	5.60	5.52	timeout			8.24	7.56	7.14	9.19	7.51	7.33	0.372	0.05