Scalable Bit-Blasting with Abstractions

Aina Niemetz[1], Mathias Preiner[1] and Yoni Zohar[2]

[1] Stanford University
[2] Bar-Ilan University

BIU, March 17, 2025

Stanford University

Stanford | Center for Automated Reasoning

Bar-Ilan University
אוניברסיטת בר־אילן

Background

Structure of This Talk

- ▷ Pre-talk: General introduction to the field
- ▷ (Skip technical background – will fill in along the way)
- ▷ Talk: Describe a recent research project
- ▷ Assignment: short google form you can do in class

Pre-talk

Why Formal Verification?

What Does This Code Do?

```
int b(int* a, int size, int key)
{
  int low = 0;
  int high = size - 1;
  while (low <= high)
  {
    int mid = (low + high) / 2;
    int midVal = a[mid];
    if (midVal < key)
    {
      low=mid+1;
    } else if (midVal > key)
    {
      high=mid-1;
    }
    else {
      return mid;
    }
    return -(low + 1);
  }
  return -1;
}
```

Why Formal Verification?

Where is The Bug in This Code?

```
int b(int* a, int size, int key)
{
  int low = 0;
  int high = size - 1;
  while (low <= high)
  {
    int mid = (low + high) / 2;
    int midVal = a[mid];
    if (midVal < key)
    {
      low=mid+1;
    } else if (midVal > key)
    {
      high=mid-1;
    }
    else {
      return mid;
    }
    return -(low + 1);
  }
  return -1;
}
```

Why Formal Verification?

Where is The Bug in This Code?

```
int b(int* a, int size, int key)
{
  int low = 0;
  int high = size - 1;
  while (low <= high)
  {
    int mid = (low + high) / 2;
    int midVal = a[mid];
    if (midVal < key)
    {
      low=mid+1;
    } else if (midVal > key)
    {
      high=mid-1;
    }
    else {
      return mid;
    }
    return -(low + 1);
  }
  return -1;
}
```

- Overflow
- Was undetected for 20+ years
- int mid = (low + high) / 2;
- Fix: int mid = low + (high - low) / 2;

Bugs in Software

- Writing code is a complicated intellectual activity
- We are humans
- Our code has mistakes

Bugs in Software

- Writing code is a complicated intellectual activity
- We are humans
- Our code has mistakes

- Disclaimer: of course you do not have bugs. You are special.

Formal Verification: The Turning Point

Intel FDIV Bug

- ▷ 1994: Pentium Processors
- ▷ Bug in division operation
- ▷ Was found only after distribution
- ▷ Cost: $475M

Result: Massive Progress in Formal Verification

Formal Verification: More Recent Motivation

Blockchain Horror Stories

- ▷ Blockchain is a new technology for distributed finance
- ▷ In the beginning (Bitcoin): simple operations
- ▷ Nowadays (Ethereum): arbitrary complex smart contracts
- ▷ 2016: The DAO Bug – $40M worth of crypto was stolen
- ▷ Many similar bugs caused other losses

Formal Verification: More Recent Motivation

Boeing 737

- 2018,2019: Plane Crashes
- Many reasons, including software bugs

Boeing says it has a software fix ready for its 737 Max airplanes that will be unveiled to airline officials, pilots and aviation authorities from around the world Wednesday, as the aircraft manufacturer works to rebuild trust among its customers and the flying public following two fatal crashes of the planes in recent months.

Formal Verification

Importance

- ▷ Software is everywhere
- ▷ Humans write it and make mistakes (Not you!)
- ▷ If we know of a bug: stop everything until fixed
- ▷ Dangerous: Unaware of bugs

How can we find bugs?

The Traditional Approach: Testing

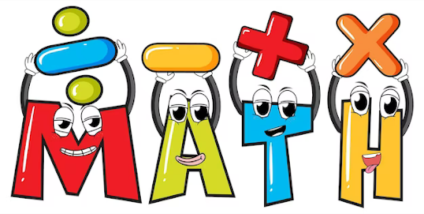
- Run your code on a range of inputs
- QA teams or programmers
- Partial Automation
- Would we ever have found the overflow bug? . . .

We will never cover all cases

An Alternative Approach: Math

- In math: we want to prove that a theorem always holds
- In SW: we want to prove that a program always satisfies something
- Mathematicians do not settle for tests
  - Is Pythagoras Theorem true becuase we tried some examples?
- Why should we?

Idea:
Formulate a math theorem that specifies the correctness of a program.
Then, prove it.



Proving

- ▷ Analogy to math only goes so far
- ▷ programs are huge compared to theorems
- ▷ programs change rapidly, theorems are static

Proofs should be done Automatically

Formal Verification Tools

- ▷ CBMC, JBMC (Oxford)
- ▷ Pono (Stanford)
- ▷ Boogie (Microsoft Research)
- ▷ Zelkova (Amazon Web Services)
- ▷ Move-prover (Stanford, Facebook)
- ▷ KeY (Universities of Chalmers, KIT, Darmstadt)
- ▷ Jasper Gold (Cadence)
- ▷ SeaHorn (University of Waterloo)
- ▷ Many many more. . .

Demo

```
void s(int* x, int* y) {
  int tmp = *x;
  *x = y;
  *y = tmp;
}
```

Example

- ▷ What does the above code do?

Demo

```
void s(int* x, int* y) {
  int tmp = *x;
  *x = y;
  *y = tmp;
}
```

Example

- ▷ What does the above code do?
- ▷ swaps

Demo

```
void s(int* x, int* y) {
  int tmp = *x;
  *x = y;
  *y = tmp;
}
```

Example

- What does the above code do?
- swaps
- But it has a bug: where?

Demo

```c
void s(int* x, int* y) {
  int tmp = *x;
  *x = y;
  *y = tmp;
}
```

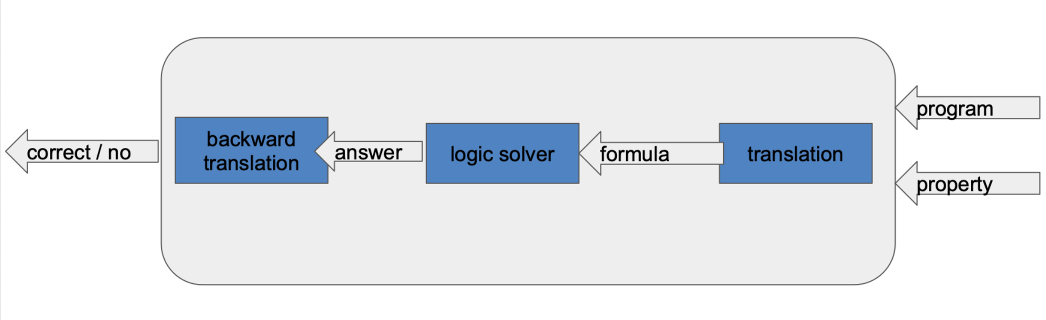
Example

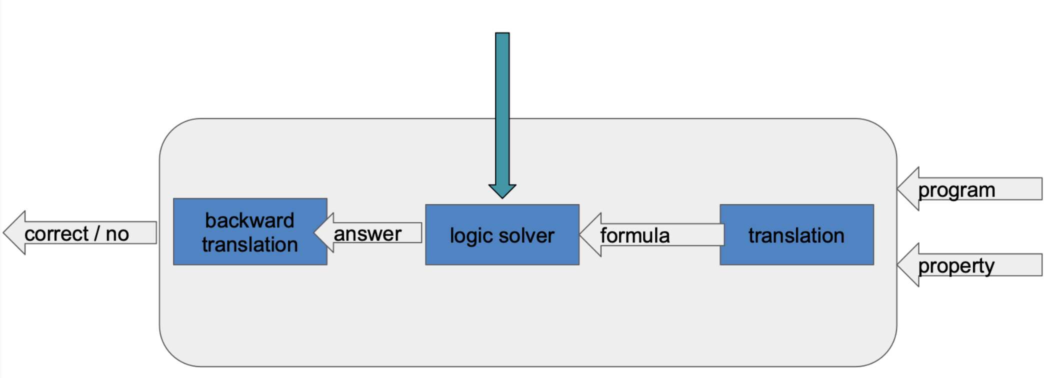
- ▷ What does the above code do?
- ▷ swaps
- ▷ But it has a bug: where?
- ▷ y is missing a $*$

Demo

- Tool: SeaHorn
- sat means there is a bug
- unsat means there is no bug

How Does This Work?





- Our focus: solvers
- In particular: SMT solvers

Tradeoff

- Rice's Theorem says that our alternative approach must fail
- True in worst-case analysis
- Still, it works in many interesting and important cases
- But often, they loop forever or return unknown

Talk

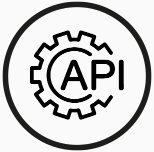
Satisfiability Modulo Theories (SMT)

What

- Boolans, QBFs
- UFs, arrays, numbers
- `cvc5.github.io`

How

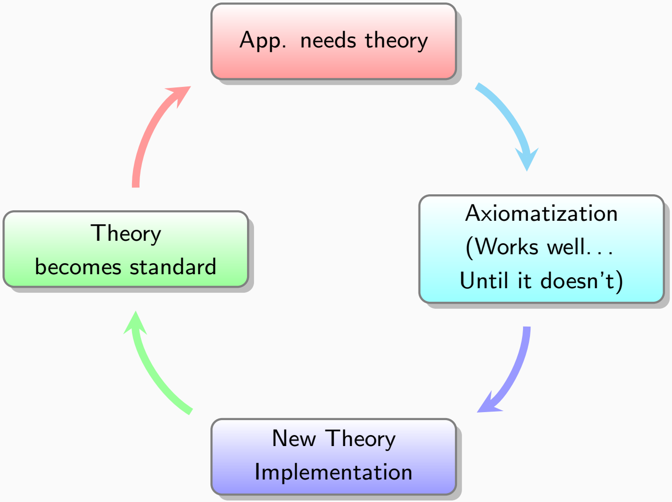
- Textual
- Interactive
- APIs: C++ / Python / Java



Why

- Mainly: Verification (HW / SW)
- General Problem Solving
- More: Biology, Economics, Modeling, ...

The SMT Cycle



The SMT Cycle: Examples

Linear Arithmetic

- ▷ Axioms [Persburger'29]
  - $\forall x. 0 \neq x + 1$
  - $\forall x \forall y. x + 1 = y + 1 \Rightarrow x = y$
  - $\forall x. x + 0 = 0$
  - $\forall x \forall y. x + (y + 1) = (x + y) + 1$
- ▷ Implementation: Simplex

The SMT Cycle: Examples

Non-linear Arithmetic

- ▷ Axioms [Peano'1889]
  - Persburger + Multiplication Axioms
- ▷ Implementation:
  - Simplex + Heuristics
  - Counterexample Abstraction Refinement (CEGAR)

(Formally, the implementation does not really correspond to the axiomatization...)

The SMT Cycle: Examples

Arrays

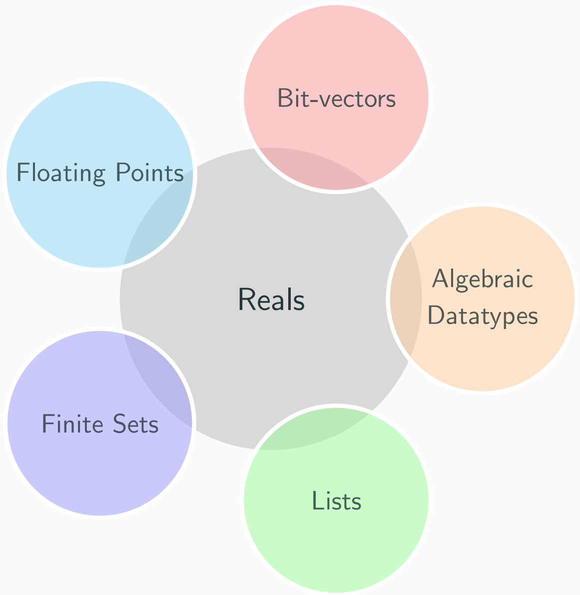
- ▷ Axioms [McCarthy'62]
  - $\forall a, i, k. read(write(a, i, k), i) = k$
  - $\forall a, i, j, k. read(write(a, i, k), j) = read(a, j)$
  - $\forall a, b. a \neq b \Rightarrow \exists i. read(a, i) \neq read(b, i)$
- ▷ Implementation:
  - "Weakly Equivalent Arrays" [Christ,Hoenicke'15]

The SMT Cycle: Examples

Unicode Strings

- ▷ Axioms
  - Concatenation and length
  - Elimination of other operators (sub-string, replace, regex, and more...)
- ▷ Implementation:
  - Automata-based
  - Simplification and search

More Examples



The Second SMT Cycle

App. needs improved theory

Improved encoding (Works well. . . Until it doesn't)

Improved theory implementation

Default / option is added

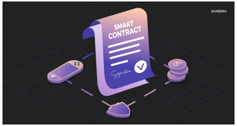
The Second SMT Cycle

Examples:

- ▷ Bools: Many applications work in the SAT level
- ▷ Algebraic Datatypes: eager / lazy approaches
- ▷ Strings: AWS continuosly produce hard string benchmarks
- ▷ Quantifiers: Many applications require quantifiers, and so new patterns emerge
- ▷ Combination: Some bottlenecks arise from theory combination
- ▷ . . .

This Time

1. Application: Smart Contracts Verification
2. Theory: Bit-blasting
3. Improvement: 256-bits machine integers




Theory of Fixed-Size Bit-Vectors

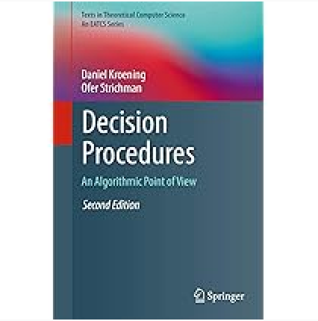
$$(x \ll 001) \geq_s 000 \ \wedge \ x <_u 100 \ \wedge \ (x \cdot 010) \bmod 011 \ = \ x + 001$$

sat: $x = 001$

- **constants, variables:** $010, \ 2_{[3]}, \ x_{[3]}$
- **bit-vector** operators: $<_u$, $>_s$, $\sim$, $\&$, $\gg$, $\gg_a$, $\circ$, $[:]$, $+$, $\cdot$, $\div$, ...
- **arithmetic** operators modulo 2^n (**overflow semantics!**)

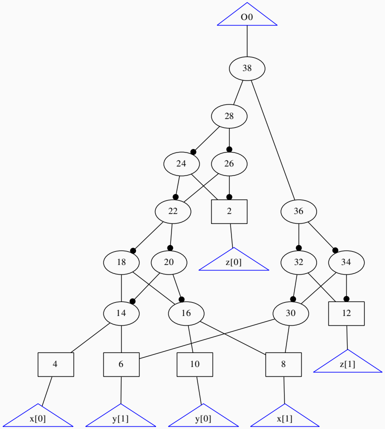
Bit-Blasting

- current **state-of-the-art**
  - rewriting **eager reduction** to SAT
  - BV terms » CNF
- efficient in practice
  - **significant** increase in formula size



Bit-Blasting

Example $x_{[2]} * y_{[2]} = z_{[2]}$



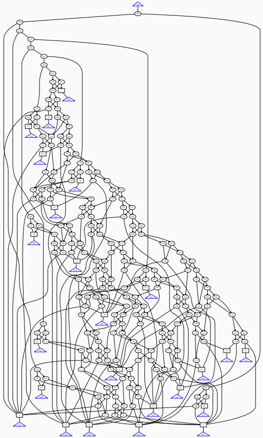
- ▶ current **state-of-the-art**
- ▷ rewriting **eager reduction** to SAT
- ▷ BV terms » AIG circuit » CNF

- ▶ efficient in practice
- ▷ **significant** increase in formula size

Bit-Blasting

Example $x_{[8]} * y_{[8]} = z_{[8]}$

- current **state-of-the-art**
  - rewriting **eager reduction** to SAT
  - BV terms » AIG circuit » CNF



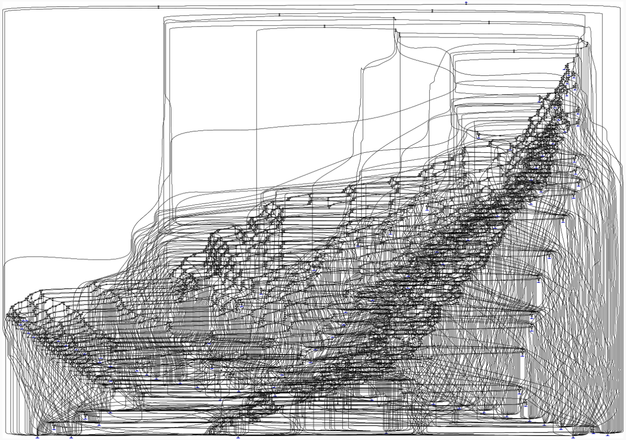
- efficient in practice
  - **significant** increase in formula size

Bit-Blasting

- current **state-of-the-art**
  - rewriting **eager reduction** to SAT
  - BV terms » AIG circuit » CNF

- efficient in practice
  - **significant** increase in formula size

Example $x_{[32]} * y_{[32]} = z_{[32]}$



Limitations of Bit-Blasting

Scalability

- ▷ does not generally scale well with **increasing** bit-width
- ▷ does not generally scale well with **arithmetic** operators
- ▷ **smart contracts:** 256 bits, heavy use of $\{\cdot, \div, \text{mod}\}$
- ▷ lemma.smt2

Intuition (debatable)

- ▷ Semantics via integers (not bits)
- ▷ Information is "lost"
- ▷ Feels wrong !

Alternatives to Bit-blasting

- ▷ Int-blasting [Bozzano et al. 2006, Zohar et al. 2022]
  - Reduction to integer arithmetic
- ▷ Layering [Bruttomesso et al. 2007, Hadarean et al. 2014]
  - Cheap checks + bit-blasting
- ▷ MC-SAT [Zeljic et al. 2016]
  - Word-level explanations + bit-blasting
- ▷ Local Search [Niemetz et al. 2017]
  - Fast procedures for SAT isntances
- ▷ PolySAT [Rath et al. 2024]
  - Aimed for non-linear BV polynomials

Every Technique Has Scalability Issues

1,500 benchmarks[1] instantiated with bit-widths 16,. . . ,8192, ∼ 85 sat, ∼ 1415 unsat

Solved Benchmarks

bw	Bit-Blast Bitwuzla	Lazy+Layered CVC4	MCSAT Yices2	Int-Blast cvc5	PolySAT Z3	Bit-Blast+Abstr Bitwuzla
16	1,495	1,458	1,394	1,116	696	
32	1,459	1,390	1,194	1,102	672	
64	1,440	1,368	1,112	1,077	668	
128	1,433	1,308	1,076	1,017	648	
256	1,388	1,232	987	916	637	
512	1,277	1,162	916	788	620	
1,024	1,065	774	794	613	608	
2,048	844	401	668	528	576	
4,096	816	300	572	428	562	
8,192	744	202	492	389	552	
	99% → 49%	97% → 13%	93% → 33%	74% → 26%	46% → 37%	

Limits : 1,200 seconds, 8GB memory

[1] 500 term and formula equivalence checks enumerated with cvc5's SyGuS solver using SyGuS grammar $\{0, 1, x, s, t, \approx, \not\approx, <_u, \leq_u, \sim, \&, \ll, \gg, \diamond\}$ for $\diamond \in \{\cdot, \div, \text{mod}\}$.

Every Technique Has Scalability Issues

1,500 benchmarks[1] instantiated with bit-widths 16,. . . ,8192, $\sim$ 85 sat, $\sim$ 1415 unsat

Solved Benchmarks

bw	Bit-Blast Bitwuzla	Lazy+Layered CVC4	MCSAT Yices2	Int-Blast cvc5	PolySAT Z3	Bit-Blast+Abstr Bitwuzla
16	1,495	1,458	1,394	1,116	696	**1,495**
32	1,459	1,390	1,194	1,102	672	**1,459**
64	1,440	1,368	1,112	1,077	668	**1,454**
128	1,433	1,308	1,076	1,017	648	**1,458**
256	1,388	1,232	987	916	637	**1,456**
512	1,277	1,162	916	788	620	**1,449**
1,024	1,065	774	794	613	608	**1,434**
2,048	844	401	668	528	576	**1,365**
4,096	816	300	572	428	562	**1,300**
8,192	744	202	492	389	552	**1,274**
	99% → 49%	97% → 13%	93% → 33%	74% → 26%	46% → 37%	**99% → 85%**

Limits : 1,200 seconds, 8GB memory

[1] 500 term and formula equivalence checks enumerated with cvc5's SyGuS solver using SyGuS grammar $\{0, 1, x, s, t, \approx, \not\approx, <_u, \leq_u, \sim, \&, \ll, \gg, \diamond\}$ for $\diamond \in \{\cdot, \div, \text{mod}\}$.

This Talk

How this performance was achieved?

u really 'bout to do it?

Contributions

Approach

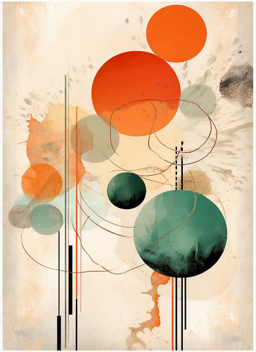
- ~~Replacing~~ Augmenting bit-blasting
- Algorithm that builds on and falls back to bit-blasting

Techniques

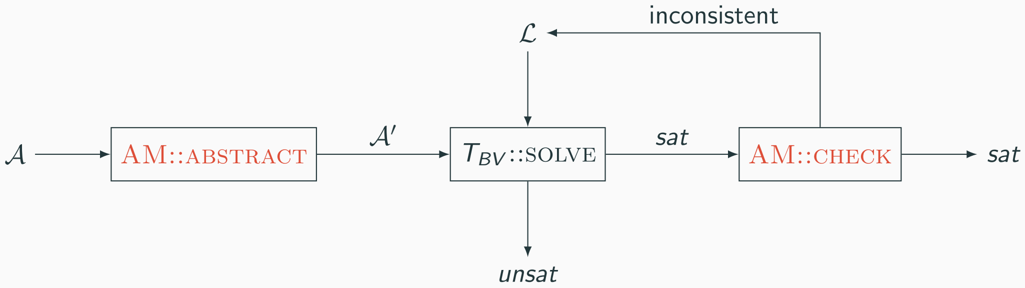
- Abstraction of $\{\cdot, \div, \text{mod}\}$
- Hand-crafted lemmas
- Synthesized lemmas (offline)
- Lemma scoring scheme
- Implementation and evaluation in Bitwuzla

Results

- Goal: good for blockchains, not embaressing for everything else
- End result: general improvement on diverse benchmarks



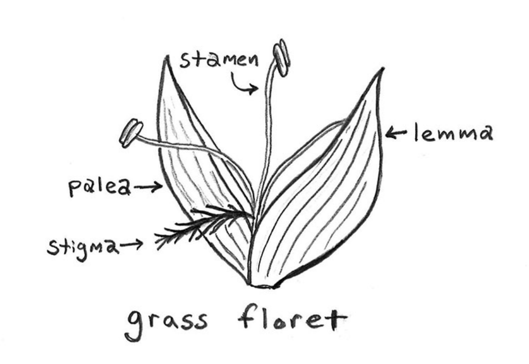
Abstraction-Refinement Loop



- **over-approximation**
- abstract $\cdot$, $\div$, mod

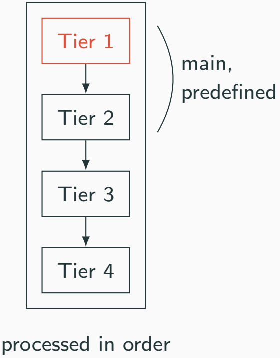
- **check consistency**
  - consistent: ✓
  - inconsistent: refine abstraction

The Big Question: Which Lemmas?



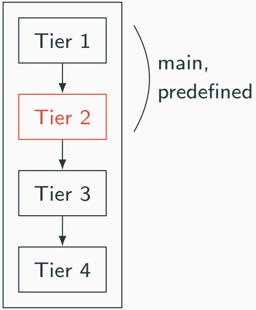
4-Tiered Refinement Schemes

- **Tier 1 Hand-Crafted Lemmas**
  - **basic** properties of the abstracted operator
  - properties based on **invertibility conditions**



4-Tiered Refinement Schemes

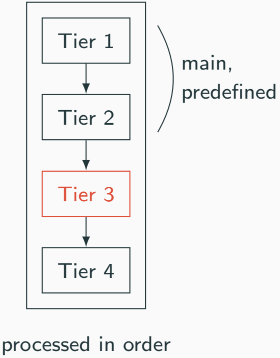
- **Tier 2 Synthesized Lemmas**
  - synthesized via **syntax-restricted abduction** with **cvc5**
  - offline



processed in order

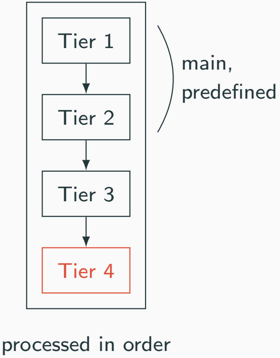
4-Tiered Refinement Schemes

- **Tier 3 Value Instantiation** Lemmas
  - rule out **current inconsistent model value**



4-Tiered Refinement Schemes

- **Tier 4 bit-blasting** lemmas



Main Lemmas (Tiers 1+2)

Tier 1 (hand-crafted) have *
Tier 2 (abduction) don't

bvmul			
1*	$s \approx 2^i \Rightarrow t \approx x \ll i$	11	$t \not\approx (1 \mid \sim(x \oplus s))$
2*	$s \approx -2^i \Rightarrow t \approx -x \ll i$	12	$t \not\approx (\sim 1 \mid (x \oplus s))$
3*	$t[0] \approx (x[0] \mathbin{\&} s[0])$	13	$x \not\approx ((x \ll (s + t)) - 1)$
4*	$((-s \mid s) \mathbin{\&} t) \approx t$	14	$x \not\approx (1 - (x \ll (s - t)))$
5	$s \not\approx \sim(t \mid (1 \mathbin{\&} (x \mid s)))$	15	$s \not\approx (1 + (s \ll (t - x)))$
6	$(x \mathbin{\&} t) \not\approx (s \mid \sim t)$	16	$s \not\approx (1 - (s \ll (t - x)))$
7	$t \not\approx ((s \mid 1) \ll (t \ll x))$	17	$s \not\approx (1 + (s \ll (x - t)))$
8	$s \approx (s \ll (x \mathbin{\&} (1 \gg t)))$	18	$t \not\approx (1 \mid (x + s))$
9	$t \geq_u (1 \mathbin{\&} ((x \mathbin{\&} s) \gg 1))$	19	$x \not\approx \sim(x \ll (s + t))$
10	$x \not\approx (1 \oplus (x \ll (s \oplus t)))$		

Main Lemmas (Tiers 1+2)

Tier 1 (hand-crafted) have *
Tier 2 (abduction) don't

`bvudiv`			
1*	$s \approx 2^i \Rightarrow t \approx x \gg i$	19	$(x \gg t) \not\approx (s \mid t)$
2*	$(s \approx x \wedge s \not\approx 0) \Rightarrow t \approx 1$	20	$s \not\approx {\sim}(s \gg (t \gg 1))$
3*	$s \approx 0 \Rightarrow t \approx {\sim}0$	21	$x \not\approx {\sim}(x \,\&\, (t \ll 1))$
4*	$(x \approx 0 \wedge s \not\approx 0) \Rightarrow t \approx 0$	22	$t \geq_u ((x \ll 1) \gg s)$
5*	$(s \approx {\sim}0 \wedge x \not\approx {\sim}0) \Rightarrow t \approx 0$	23	$x \geq_u (s \ll {\sim}(x \mid t))$
6*	$s \not\approx 0 \Rightarrow t \leq_u x$	24	$x \geq_u (t \ll {\sim}(x \mid s))$
7	$x \geq_u -(-s \,\&\, -t)$	25	$x \geq_u (t \oplus (t \gg (s \gg 1)))$
8	$-(s \mid 1) \geq_u t$	26	$x \geq_u (s \oplus (s \gg (t \gg 1)))$
9	$t \not\approx -(s \,\&\, {\sim}x)$	27	$x \geq_u (s \ll {\sim}(x \oplus t))$
10	$(s \mid t) \not\approx (x \,\&\, {\sim}1)$	28	$x \geq_u (t \ll {\sim}(x \oplus s))$
11	$(s \mid 1) \not\approx (x \,\&\, {\sim}t)$	29	$x \not\approx (t + (s \mid (x + s)))$
12	$(x \,\&\, -t) \geq_u (s \,\&\, t)$	30	$x \not\approx (t + (1 + (1 \ll x)))$
13	$s \geq_u (x \gg t)$	31	$s \geq_u ((x + t) \gg t)$
14	$x \geq_u ((s \gg (s \ll t)) \ll 1)$	32	$x \not\approx (t + (t + (x \mid s)))$
15	$x \geq_u ((t \ll 1) \gg (t \ll s))$	33	$(s \oplus (x \mid t)) \geq_u (t \oplus 1)$
16	$t \geq_u ((x \gg s) \ll 1)$	34	$t \geq_u (x \gg (s - 1))$
17	$x \geq_u ((x \mid t) \,\&\, (s \ll 1))$	35	$(s - 1) \geq_u (x \gg t)$
18	$x \geq_u ((x \mid s) \,\&\, (t \ll 1))$	36	$x \not\approx (1 - (x \ll (x - t)))$

Main Lemmas (Tiers 1+2)

Tier 1 (hand-crafted) have *
Tier 2 (abduction) don't

bvurem			
1*	$s \approx 2^i \Rightarrow t \approx (0_{[\kappa(x)-i]} \circ x[i-1:0])$	9	$x \geq_u (t \mid (x \,\&\, s))$
2*	$s \not\approx 0 \Rightarrow t \leq_u s$	10	$1 \not\approx (t \,\&\, {\sim}(x \mid s))$
3*	$x \approx 0 \Rightarrow t \approx 0$	11	$t \not\approx ({\sim}x \mid -s)$
4*	$s \approx 0 \Rightarrow t \approx x$	12	$(t \,\&\, (x \mid s)) \geq_u (t \,\&\, 1)$
5*	$s \approx x \Rightarrow t \approx 0$	13	$x \not\approx (-x \mid -{\sim}t)$
6*	$x <_u s \Rightarrow t \approx x$	14	$(x + -s) \geq_u t$
7*	${\sim} - s \geq_u t$	15	$(-s \oplus (x \mid s)) \geq_u t$
8	$x \approx (x \,\&\, (s \mid (t \mid -s)))$		

Tier 1 Lemmas (hand-crafted): multiplication

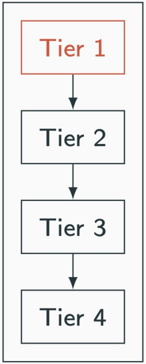
Lemmas

(t abstracts $x \cdot s$)

- ▷ 1* $s \approx 2^i \Rightarrow t \approx x \ll i$
- ▷ 2* $s \approx -2^i \Rightarrow t \approx -x \ll i$
- ▷ 3* $t[0] \approx (x[0] \mathbin{\&} s[0])$
- ▷ 4* $((-s \mid s) \mathbin{\&} t) \approx t$

Description

- ▷ Lemmas 1 – 2: powers of two
  - Actually, classes of lemmas
- ▷ Third lemma: “evenness” (lsb)
- ▷ Fourth lemma: invertibility conditions



On Invertibility Conditions [Niemetz et al. 2018]

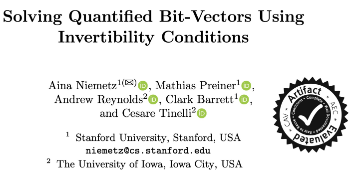
- For a literal $\ell(x, s, t)$, $IC_\ell(s, t)$ satisfies:

$$\exists x.\, \ell(x, s, t) \Leftrightarrow IC_\ell(s, t)$$

- Used for quantifier-elimination / instantiation

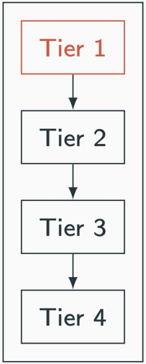
For our context

- ICs are lemmas: $\ell(x, s, t) \Rightarrow \exists x. \ell(x, s, t) \Rightarrow IC_\ell(s, t)$
- $x \cdot s = t \Rightarrow \exists x.\, x \cdot s = t \Leftrightarrow IC_{x \cdot s = t}(s, t) \;=\; ((-s \mid s)\ \&\ t) \approx t$


Solving Quantified Bit-Vectors Using Invertibility Conditions

Aina Niemetz[1(✉)], Mathias Preiner[1], Andrew Reynolds[2], Clark Barrett[1], and Cesare Tinelli[2]

[1] Stanford University, Stanford, USA
niemetz@cs.stanford.edu
[2] The University of Iowa, Iowa City, USA



Tier 1 Lemmas (hand-crafted): division

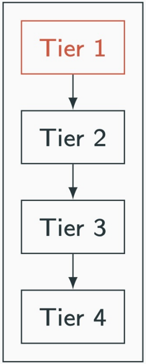
Lemmas

(t abstracts $x \div s$)

- ▷ 1^* $s \approx 2^i \Rightarrow t \approx x \gg i$
- ▷ 2^* $(s \approx x \wedge s \not\approx 0) \Rightarrow t \approx 1$
- ▷ 3^* $s \approx 0 \Rightarrow t \approx {\sim}0$
- ▷ 4^* $(x \approx 0 \wedge s \not\approx 0) \Rightarrow t \approx 0$
- ▷ 5^* $(s \approx {\sim}0 \wedge x \not\approx {\sim}0) \Rightarrow t \approx 0$
- ▷ 6^* $s \not\approx 0 \Rightarrow t \leq_u x$

Description

- ▷ First lemma: powers of 2
- ▷ Lemmas 2 – 5: special cases
- ▷ Lemma 6: division $\leq$ numerator
- ▷ Did not use invertibility conditions:
  - $(s \cdot t) \div s = t$ and $s \div (s \div t) = t$
  - introduce new abstracted terms
  - Compromises CEGAR termination



Tier 1 Lemmas (hand-crafted): remainder

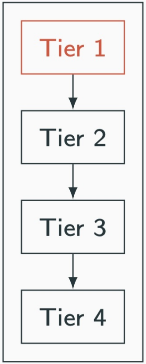
Lemmas

(t abstracts x mod s)

- ▷ 1^* $s \approx 2^i \;\Rightarrow\; t \approx (0_{[\kappa(x)-i]} \circ x[i-1:0])$
- ▷ 2^* $x \approx 0 \;\Rightarrow\; t \approx 0$
- ▷ 3^* $s \approx 0 \;\Rightarrow\; t \approx x$
- ▷ 4^* $s \approx x \;\Rightarrow\; t \approx 0$
- ▷ 5^* $x <_u s \;\Rightarrow\; t \approx x$
- ▷ 6^* $s \not\approx 0 \;\Rightarrow\; t \leq_u s$
- ▷ 7^* $\sim -s \geq_u t$

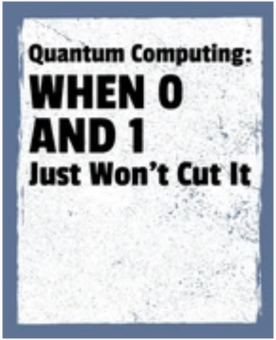
Description

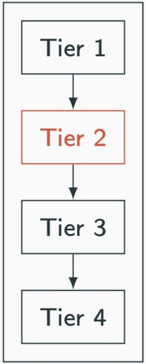
- ▷ First lemma: powers of 2
- ▷ Lemmas 2 – 4: special cases
- ▷ Lemmas 5 – 6: general properties
- ▷ Lemma 7: IC



Abduction-based Lemmas

- ▷ Initial experiments have shown that Tier 1 is not enough
- ▷ Had to come up with more lemmas
- ▷ Went with an automatic approach





Abduction-based Lemmas

Abduction

Assuming $A \not\models B$, find C s.t.:

- $A \wedge C \models B$
- $A \wedge C \not\models \bot$

Example: abduction{1,2}.smt2

From abducts to lemmas

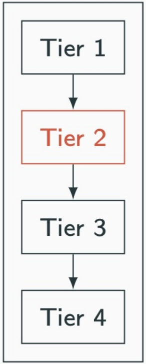
Assuming $\top \not\models (x \cdot s \neq t)$, find C s.t.:

- $\top \wedge C \models (x \cdot s \neq t)$
- $\top \wedge C \not\models \bot$

In particular:

- $x \cdot s = t \Rightarrow \neg C$
- $\neg C$ is not trivial

$\neg C$ is a lemma!



Grammars

- Mostly cheap operators (for bit-blasting)
- However, still use $+$
- Several small grammars rather than one big grammar

$$\gamma_c = \{x, s, t, \approx, \not\approx, <_u, \leq_u, 0, 1\}$$

$$\gamma_0 = \gamma_c \cup \{\sim, \&, |, \oplus\}$$

$$\gamma_1 = \gamma_c \cup \{-, \sim, \&, |\}$$

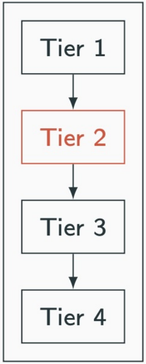
$$\gamma_2 = \gamma_1 \cup \{\oplus\}$$

$$\gamma_3 = \gamma_1 \cup \{\ll, \gg\}$$

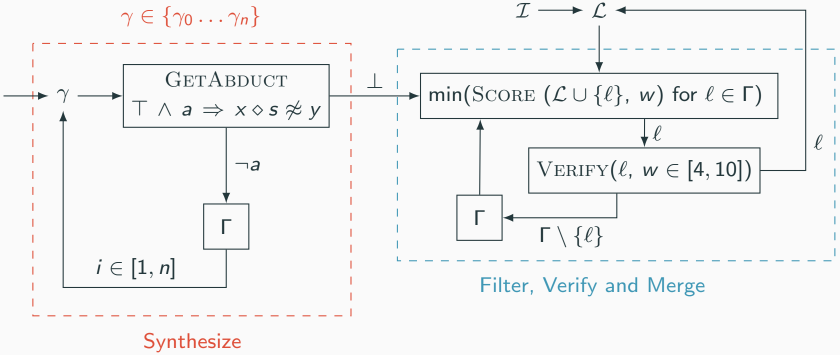
$$\gamma_4 = \gamma_3 \cup \{\oplus\}$$

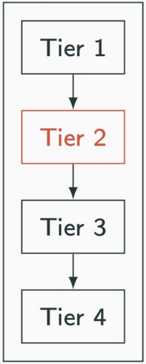
$$\gamma_5 = \gamma_4 \cup \{+\}$$

$$\gamma_6 = \gamma_c \cup \{-, +, -_+, \ll, \gg\}$$



Lemma Synthesis

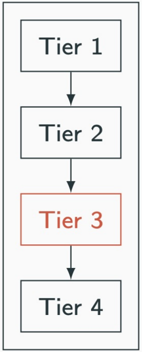




- n: number of abducts per grammar
- $\mathcal{I}$: hand-crafted lemmas
- Γ: candidate lemmas
- $\mathcal{L}$: result

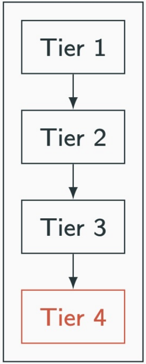
Tier 3 Lemmas (value instantiations)

- rule out **current inconsistent model value**
- only added if none in tiers 1–2 are violated
- Heuristically limited to #instantiations$= 1/8$ of the bit-width
- Example:
  t abstracts $x \cdot s$
  $\mathcal{M} = \{x_{[32]} = 3, s_{[32]} = 6, t_{[32]} = 1\}$,
  $\longrightarrow$ add lemma $(x = 3 \wedge s = 6) \Rightarrow t = 18$



Tier 4 Lemmas (bit-blasting)

- **last resort**
- add lemma to **enforce bit-blasting** of the abstracted term
- Example: t abstracts $x \cdot s$
  $\longrightarrow$ add lemma $t \approx x \cdot s$



Lemma Score

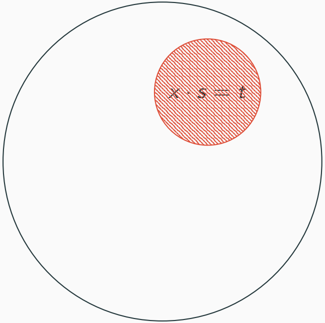
- When stating, we evaluated lemmas on benchmarks
- But as we made progress, this became costly to check
- Decided on a scoring mechanism, independent of benchmarks

Lemma Score

$\textsc{Score}(\ell, w) := \#$ triplets (v^x, v^s, v^t) where $\ell[v^x, v^s, v^t] = \top$.

Example. multiplication with $w = 4$

- Worst score: $2^4 \times 2^4 \times 2^4 = 4096$
- Best score: $2^4 \times 2^4 = 256$

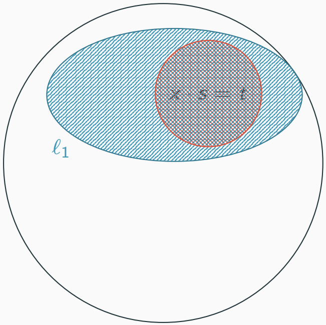


Lemma Score

$\textsc{Score}(\ell, w) :=$ # triplets (v^x, v^s, v^t) where $\ell[v^x, v^s, v^t] = \top$.

Example. multiplication with $w = 4$

- Worst score: $2^4 \times 2^4 \times 2^4 = 4096$
- Best score: $2^4 \times 2^4 = 256$
- Score for 1^*: 2416

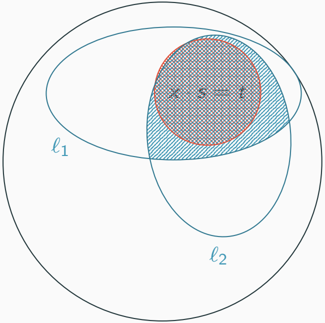


Lemma Score

$\textsc{Score}(\ell, w) := \#$ triplets (v^x, v^s, v^t) where $\ell[v^x, v^s, v^t] = \top$.

Example. multiplication with $w = 4$

- ▷ Worst score: $2^4 \times 2^4 \times 2^4 = 4096$
- ▷ Best score: $2^4 \times 2^4 = 256$
- ▷ Score for 1^*: 2416
- ▷ Score for 2^*: 2791

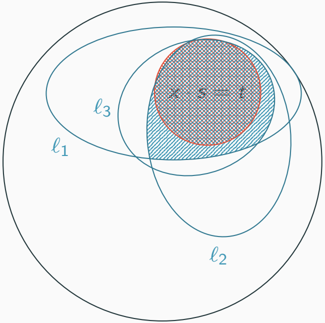


Lemma Score

$$\textsc{Score}(\ell, w) := \#\ \text{triplets}\ (v^x, v^s, v^t)\ \text{where}\ \ell[v^x, v^s, v^t] = \top.$$

Example. multiplication with $w = 4$

- ▷ Worst score: $2^4 \times 2^4 \times 2^4 = 4096$
- ▷ Best score: $2^4 \times 2^4 = 256$
- ▷ Score for 1^*: 2416
- ▷ Score for 2^*: 2791
- ▷ Score for 3^*: 2048

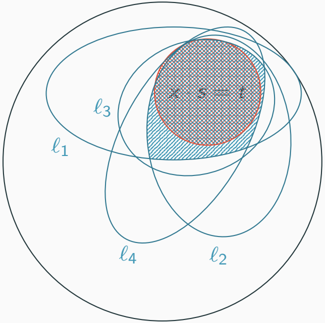


Lemma Score

$\textsc{Score}(\ell, w) := \#$ triplets (v^x, v^s, v^t) where $\ell[v^x, v^s, v^t] = \top$.

Example. multiplication with $w = 4$

- ▷ Worst score: $2^4 \times 2^4 \times 2^4 = 4096$
- ▷ Best score: $2^4 \times 2^4 = 256$
- ▷ Score for 1^*: 2416
- ▷ Score for 2^*: 2791
- ▷ Score for 3^*: 2048
- ▷ Score for 4^*: 1961

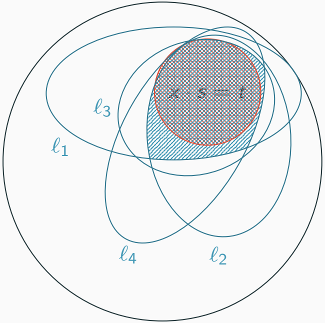


Lemma Score

$\textsc{Score}(\ell, w) := \#$ triplets (v^x, v^s, v^t) where $\ell[v^x, v^s, v^t] = \top$.

Example. multiplication with $w = 4$

- ▷ Worst score: $2^4 \times 2^4 \times 2^4 = 4096$
- ▷ Best score: $2^4 \times 2^4 = 256$
- ▷ Score for 1^*: 2416
- ▷ Score for 2^*: 2791
- ▷ Score for 3^*: 2048
- ▷ Score for 4^*: 1961
- ▷ **Score for $\{1^*, 2^*, 3^*, 4^*\}$: 704**
  - ▶ rules out 88% of incorrect triplets



Lemma Score

- worst possible: 4096
- best possible: 256

Hand-crafted

- multiplication: 704
- division: 1366
- remainder: 616

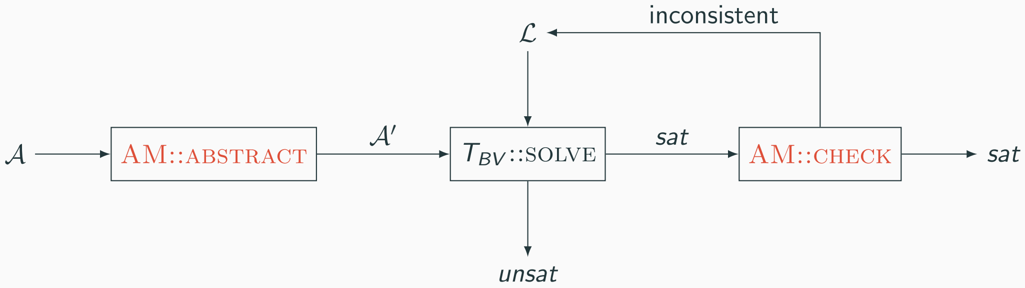
Adding Abducted Lemmas

- multiplication: 490
- division: 394
- remainder: 400

Verification of Lemmas

- Hand-crafted lemmas are intuitive, but maybe we were wrong?
- Abduction-based lemmas are correct-by-construction only for bit-width 4.
- **verified lemmas for bit-widths [1, 512]**
- Bitwuzla, cvc5, Yices, Z3
- 8 hours time limit, 8 GB memory limit
- 16,896 benchmark, 6348 CPU hours

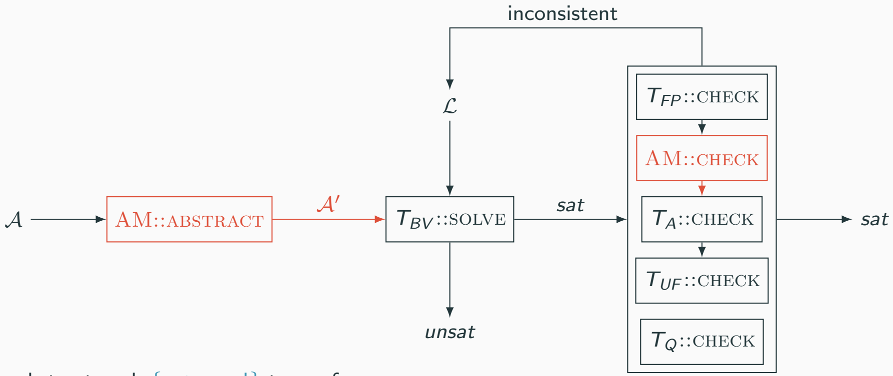
Implementation in Bitwuzla



- **over-approximation**
- abstract $\cdot$, $\div$, mod

- **check consistency**
  - » consistent: ✓
  - » inconsistent: refine abstraction

Implementation in Bitwuzla



- abstract each $\{\cdot, \div, \text{mod}\}$ term of size ≥ 32 with a **fresh constant**
- optional: **assertion abstraction**
  - interleaved with term abstraction
  - effective if unsat core very small

- order **not arbitrary**
  - T_{FP} word-blasted to T_{BV}
  - T_A, T_{UF} and T_Q require consistent T_{BV} abstraction

Evaluation

- ▷ Original goal: improve on benchmarks with hard arithmetic operators with large bit-widths
- ▷ Dropped benchmarks and used an abstract grade
- ▷ Got a good grade
- ▷ But what about the original goal?

- ▷ Performed an extensive evaluation.

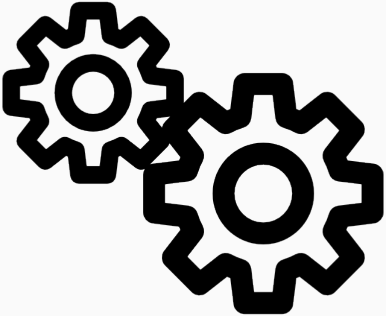
Evaluation

▸ Benchmarks

- **smart contract verification**
  - » *certora*$_1$, *certora*$_2$ (Certora Prover)
  - » *ethereum* (hevm, Ethereum Foundation)
  - 256 bit bit-vectors
  - heavy use of $\{\cdot, \div, \text{mod}\}$
- **crafted benchmarks**
  - » *syrew*
  - controlled set to evaluate effectiveness
  - equivalence checks for each operator
  - enumerated by SyGuS (cvc5) for $w = 4$
  - instantiated for 2^k with $k \in [4, 13]$
- **translation validation of ZK proofs**
  - » *ff*
  - $T_{FF} \rightarrow T_{BV}$
  - 510 bit bit-vectors
- **SMT-LIB**
  - » all supported quantifier-free and quantified logics (24 in total)

Evaluation

- **Configurations**
  - **Abstr-t** (Bitwuzla + term abstraction)
  - **Abstr-a** (Bitwuzla + assertion abstraction)
  - **Abstr-ta** (Bitwuzla + term and assertion abstraction)
  - Bitwuzla
  - cvc5
  - cvc5-ib (cvc5 with int-blasting)
  - cvc5-ff (cvc5's finite field solver)
  - Z3



- **Setup**
  - Limits: 1200 seconds, 8GB memory

Results

- New approach outperforms all other bit-blasting approaches
- Also outperforms bit-blasting
- Does not outperform the native finite fields solver of cvc5
- Reduces running time and memory in most cases

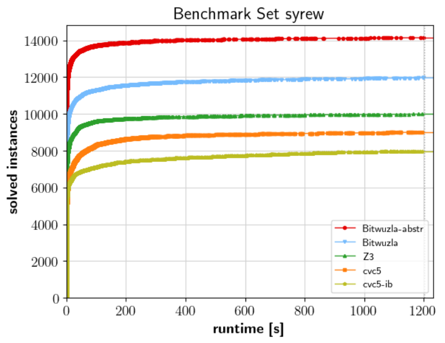
Evaluation

Benchmarks (common / total)	Solver	Solved	TO	MO	T [s]	M [GB]	T_c [s]
certora$_1$ (10/850)	ABSTR-TA	573	231	46	448k	2,492	234
	ABSTR-A	386	140	324	681k	5,201	963
	ABSTR-T	258	155	437	760k	4,807	83
	cvc5-ib	147	674	0	879k	667	52
	Bitwuzla	111	86	653	915k	6,182	192
	cvc5	90	113	610	923k	6,064	341
	Z3	30	447	373	989k	4,944	484
certora$_2$ (227/1,138)	ABSTR-TA	866	264	8	370k	1,024	11k
	ABSTR-T	866	263	9	384k	1,402	17k
	ABSTR-A	844	269	25	433k	2,661	19k
	Bitwuzla	843	266	29	439k	2,944	23k
	cvc5	705	223	210	603k	4,027	22k
	cvc5-ib	666	472	0	643k	106	15k
	Z3	612	492	34	679k	1,866	24k
ethereum (3,138/3,173)	ABSTR-T	3,173	0	0	407	11	102
	Bitwuzla	3,173	0	0	720	29	228
	Z3	3,169	4	0	6k	107	679
	cvc5	3,158	0	1	18k	36	377
	cvc5-ib	3,141	20	0	39k	21	128

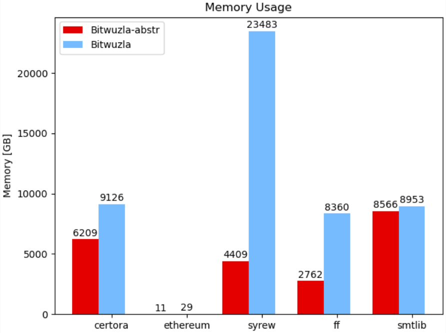
Evaluation

Benchmarks (common / total)	Solver	Solved	TO	MO	T [s]	M [GB]	T_c [s]
syrew (5,528/15,000)	Abstr-T	14,142	583	276	1,225k	4,409	2k
	Bitwuzla	11,961	744	2,296	3,955k	23,483	24k
	Z3	9,992	833	4,175	6,198k	39,506	78k
	cvc5	9,003	797	5,200	7,498k	48,421	109k
	cvc5-ib	7,974	5,137	1,632	8,836k	19,850	180k
ff (12/1,224)	cvc5-ff	973	129	122	313k	1,364	0
	Abstr-T	480	729	15	913k	2,762	0
	cvc5-ib	304	822	98	1,104k	1,074	0
	Bitwuzla	223	71	930	1,211k	8,360	277
	Z3	145	56	1,023	1,299k	8,893	3
	cvc5	40	0	1,184	1,422k	9,523	589
smtlib (125,037/155,269)	Abstr-T	148,554	1,944	152	8,770k	8,566	64k
	Bitwuzla	148,492	1,966	193	8,748k	8,953	64k
	Z3	145,121	4,846	565	13,528k	18,278	693k
	cvc5	144,829	3,775	285	13,513k	11,029	213k
	cvc5-ib	127,144	24,479	194	39,647k	15,233	5,666k

Cool Plots



Benchmark Set syrew
solved instances
runtime [s]
0
2000
4000
6000
8000
10000
12000
14000
0
200
400
600
800
1000
1200
Bitwuzla-abstr
Bitwuzla
Z3
cvc5
cvc5-ib



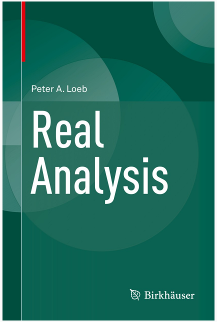
Memory Usage
Bitwuzla-abstr
Bitwuzla
Memory [GB]
0
5000
10000
15000
20000
6209
9126
11
29
4409
23483
2762
8360
8566
8953
certora
ethereum
syrew
ff
smtlib

Analysis

	Terms	Refinement Tier				
Operator	*Abstracted*	*1*	*2*	*3*	*4*	*Total*
$\cdot$	367,101	579,369	67,221	650,086	134,525	1,431,201
$\div$	55,461	126,223	109,137	73,019	7,024	315,403
mod	62,328	161,270	5,614	30,350	1,326	198,560

Refinements

- ▷ Most terms were never bit-blasted
- ▷ 80% of benchmarks solved without bit-blasting any abstracted terms
- ▷ All tiers were used overall
- ▷ Without abduction: lose 336 benchmarks, 23% slower, 61% more memory



Conclusion

Contributions:

- ▷ CEGAR
- ▷ Strong hand-crafted lemmas (including ICs)
- ▷ Novel abduction-based generation of lemmas
- ▷ Scoring scheme
- ▷ Significant Improvement, especially for blockchains

Future Work:

- ▷ (Formal) Proofs of lemmas
- ▷ Addition
- ▷ Undera-pproximations
- ▷ Integration to cvc5 / library

Conclusion

Contributions:

- CEGAR
- Strong hand-crafted lemmas (including ICs)
- Novel abduction-based generation of lemmas
- Scoring scheme
- Significant Improvement, especially for blockchains

Future Work:

- (Formal) Proofs of lemmas
- Addition
- Undera-pproximations
- Integration to cvc5 / library

Thank You!

Mandatory Quiz

https://forms.gle/any63krmDsC4B66T9