Proving LTL Properties of Bitvector Programs and Decompiled Binaries

Y. Cyrus Liu[1] Chengbin Pang[1] Daniel Dietsch[2] Eric Koskinen[1]
Ton-Chanh Le[1] Georgios Portokalidis[1] Jun Xu[1]

[1]Stevens Institute of Technology

[2]University of Freiburg

APLAS 2021 • 18 October 2021

Background

$P \vDash \varphi$, challenges vary in types of program P and assertion logic of φ.

Our setting: bitvector programs and LTL (including Reachability, Termination).

Background

$P \vDash \varphi$, challenges vary in types of program P and assertion logic of φ.

Our setting: bitvector programs and LTL (including Reachability, Termination).

Bitvector Applications

- System/low level programs.
- Precise integer reasoning.
- Memory safety.
- Binary verification.

Tool Support

SMT solvers like MathSAT, CVC4, Z3, SMTInterpol support various theories:[a]

- Bools, Ints, Floats.
- FixedSizeBitVectors.

[a] http://smtlib.cs.uiowa.edu/theories.shtml

Problems

Challenges in verification of bitvector programs

- Bit-blasting in SMT practical applications[a], leads to exponential growth ($\mathcal{O}(2^n)$).
- Verification tools (e.g. CPAchecker, Ultimate) have limited support for liveness verification over the bitvector domain.
- LTL verification tasks are absent from SV-COMP.
- Very limited bitvector benchmarks in SV-COMP.

[a] Kovásznai et al. - 2016 - Complexity of Fixed-Size Bit-Vector Logics

Example 1: Reachability

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

- CPAchecker, Ultimate etc., tools can verify it via bitvector theory for safety only.

Example 2: Termination

```
1  a = *;
2  assume(a>0);
3  while(x>0){
4    a--;
5    x = x & a;
6  }
```

- Fewer tools can handle termination of bitvector programs.
- For example, ULTIMATE does not support Bitvector logics for termination, reports Unknown results with Overapproximation.

Example 3: LTL ($\varphi = \Box(\Diamond(n < 0))$)

```
1     while(1) {
2       n = *; x = *; y = x-1;
3       while(x>0 && n>0) {
4         n++;
5         y = x | n;
6         x = x - y;
7       }
8       n = -1;
9     }
```

- No techniques can prove LTL of bitvector programs. The closest possible verifier is ULTIMATE.

Linear Approximation in SMT Solving

How can we address all of these examples?

Build on ideas from SMT solving: linear approximation.

Support bitvector through linear constraints:

$$in_w(x) =_{def} (0 \leq x < 2^w)$$

Key Idea

Approximate bitvector reasoning with integer reasoning through source translation.

- Various state-of-art verifiers support the integer domain.
- Overapproximate bit-vector operations with linear constraints.
- Sound transformation.
- Open path to binary verification.

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3    s = x >> 31;
4    x--;
5    r = x + (s&(1-s));
6    if (r<0) error();
7 }
```

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

1. Consider program expression `x >> 31`:
   - Under **condition** `x >= 0`, this expression is just "0". So replace it with `x >= 0 ? 0 : (x >> 31)`

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

1. Consider program expression `x >> 31`:
   - Under **condition** `x >= 0`, this expression is just "0". So replace it with `x >= 0 ? 0 : (x >> 31)`
   - Also, under **condition** `x < 0`, this expression is just "1", so further replace `x<0 ? 1 : ( x>=0?0:x>>31 )`

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

1. Consider program expression `x >> 31`:
   - Under **condition** `x >= 0`, this expression is just "0". So replace it with `x >= 0 ? 0 : (x >> 31)`
   - Also, under **condition** `x < 0`, this expression is just "1", so further replace `x<0 ? 1 : ( x>=0?0:x>>31 )`

After this step, **Bitwise Branching** translates above program to:

```
1 int r, s, x;
2 while(x>0){
3   s = x>=0 ? 0 : ( x<0 ? 1 : x >> 31);
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

1. Consider program expression `x >> 31`:
   - Under **condition** `x >= 0`, this expression is just "0". So replace it with `x >= 0 ? 0 : (x >> 31)`
   - Also, under **condition** `x < 0`, this expression is just "1", so further replace `x<0 ? 1 : ( x>=0?0:x>>31 )`

After this step, **Bitwise Branching** translates above program to:

```
1 int r, s, x;
2 while(x>0){
3   s = x>=0 ? 0 : ( x<0 ? 1 : x >> 31);
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

General Rule: $e_1 \geq 0 \wedge e_2 = \texttt{CHAR\_BIT} * \texttt{sizeof}(e_1) - 1 \vdash_E e_1 \texttt{>>} e_2 \rightsquigarrow 0$

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3    s = x >> 31;
4    x--;
5    r = x + (s&(1-s));
6    if (r<0) error();
7 }
```

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

2. Now consider expression `s&(1-s)`:

- Under **condition** s>=0∧(1-s)=1, this expression is equal to s&1, which is s%2, so replace it with `(s>=0&&(1-s)==1?s%2 :(s&(1-s)))`

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

2. Now consider expression `s&(1-s)`:

- Under **condition** s>=0∧(1-s)=1, this expression is equal to s&1, which is s%2, so replace it with `(s>=0&&(1-s)==1?s%2 :(s&(1-s)))`

After this 2nd step, **Bitwise Branching** translates this program to:

```
1 int r, s, x;
2 while(x>0){
3   s = x>=0 ? 0 : ( x<0 ? 1 : x >> 31);
4   x--;
5   r = x + (s>=0 && (1-s)==1 ? s%2 : (s&(1-s)));
6   if (r<0) error();
7 }
```

Transformation with bitwise branching

```
1 int r, s, x;
2 while(x>0){
3   s = x >> 31;
4   x--;
5   r = x + (s&(1-s));
6   if (r<0) error();
7 }
```

2. Now consider expression `s&(1-s)`:

- Under **condition** s>=0∧(1-s)=1, this expression is equal to s&1, which is s%2, so replace it with `(s>=0&&(1-s)==1?s%2 :(s&(1-s)))`

After this 2nd step, **Bitwise Branching** translates this program to:

```
1 int r, s, x;
2 while(x>0){
3   s = x>=0 ? 0 : ( x<0 ? 1 : x >> 31);
4   x--;
5   r = x + (s>=0 && (1-s)==1 ? s%2 : (s&(1-s)));
6   if (r<0) error();
7 }
```

General rule: $e_1 \geq 0 \wedge e_2 = 1 \vdash_E e_1 \& e_2 \rightsquigarrow e_1 \% 2$

Rewriting Rules

Table: Rewriting rules for arithmetic expressions.

Linear Condition		BV Expr.	Linear Apx.	
$e_1 = 0$	$\vdash_E$	$e_1 \& e_2$	$\rightsquigarrow 0$	[R-And-0]
$(e_1 = 0 \vee e_1 = 1) \wedge e_2 = 1$	$\vdash_E$	$e_1 \& e_2$	$\rightsquigarrow e_1$	[R-And-1]
$(e_1 = 0 \vee e_1 = 1) \wedge (e_2 = 0 \vee e_2 = 1)$	$\vdash_E$	$e_1 \& e_2$	$\rightsquigarrow e_1 \&\& e_2$	[R-And-LOG]
$e_1 \geq 0 \wedge e_2 = 1$	$\vdash_E$	$e_1 \& e_2$	$\rightsquigarrow e_1 \% 2$	[R-And-LBS]
$e_2 = 0$	$\vdash_E$	$e_1 \mid e_2$	$\rightsquigarrow e_1$	[R-Or-0]
$(e_1 = 0 \vee e_1 = 1) \wedge e_2 = 1$	$\vdash_E$	$e_1 \mid e_2$	$\rightsquigarrow 1$	[R-Or-1]
$e_2 = 0$	$\vdash_E$	$e_1 \hat{} e_2$	$\rightsquigarrow e_1$	[R-Xor-0]
$e_1 = e_2 = 0 \vee e_1 = e_2 = 1$	$\vdash_E$	$e_1 \hat{} e_2$	$\rightsquigarrow 0$	[R-Xor-Eq]
$(e_1 = 1 \wedge e_2 = 0) \vee (e_1 = 0 \wedge e_2 = 1)$	$\vdash_E$	$e_1 \hat{} e_2$	$\rightsquigarrow 1$	[R-Xor-Neq]
$e_1 \geq 0 \wedge e_2 = \texttt{CHAR\_BIT} * \texttt{sizeof}(e_1) - 1$	$\vdash_E$	$e_1 >> e_2$	$\rightsquigarrow 0$	[R-RightShift-Pos]
$e_1 < 0 \wedge e_2 = \texttt{CHAR\_BIT} * \texttt{sizeof}(e_1) - 1$	$\vdash_E$	$e_1 >> e_2$	$\rightsquigarrow -1$	[R-RightShift-Neg]

- Judgment $\mathcal{C} \vdash_E e_{bv} \rightsquigarrow e_{int}$ means under **condition** $\mathcal{C}$ bitvector expression e_{bv} can be approximated with linear expression e_{int}.
- Then, apply a substitution δ, and replace e_{bv} with `if-then-else` expression $\mathcal{C}\delta \ ? \ e_{int}\delta : e_{bv}$

Weakening Rules

Table: Weakening rules for **Relational Expressions** & **Assignment Statements**.

Linear Condition		Statement	Linear Approximation	
$e_1 \geq 0 \wedge e_2 \geq 0$	$\vdash_S$	r $\mathtt{op}_{le}$ e_1`&`e_2	$\rightsquigarrow$ r`<=`e_1 `&&` r`<=`e_2	[W-And-Pos]
$e_1 < 0 \wedge e_2 < 0$	$\vdash_S$	r $\mathtt{op}_{le}$ e_1`&`e_2	$\rightsquigarrow$ r`<=`e_1 `&&` r`<=`e_2 `&&` r`<0`	[W-And-Neg]
$e_1 \geq 0 \wedge e_2 < 0$	$\vdash_S$	r $\mathtt{op}_{eq}$ e_1`&`e_2	$\rightsquigarrow$ `0<=`r `&&` r`<=`e_1	[W-And-Mix]
$(e_1 = 0 \vee e_1 = 1) \wedge (e_2 = 0 \vee e_2 = 1)$	$\vdash_S$	$(e_1$`|`$e_2)$`==0`	$\rightsquigarrow$ e_1`==0` `&&` e_2`==0`	[R-Or-LOG]
$e_1 \geq 0 \wedge$ `is_const`(e_2)	$\vdash_S$	r $\mathtt{op}_{ge}$ e_1`|`e_2	$\rightsquigarrow$ r`>=`e_2	[W-Or-Const]
$e_1 \geq 0 \wedge e_2 \geq 0$	$\vdash_S$	r $\mathtt{op}_{ge}$ e_1`|`e_2	$\rightsquigarrow$ r`>=`e_1 `&&` r`>=`e_2	[W-Or-Pos]
$e_1 < 0 \wedge e_2 < 0$	$\vdash_S$	r $\mathtt{op}_{eq}$ e_1`|`e_2	$\rightsquigarrow$ r`>=`e_1 `&&` r`>=`e_2 `&&` r`<0`	[W-Or-Neg]
$e_1 \geq 0 \wedge e_2 < 0$	$\vdash_S$	r $\mathtt{op}_{eq}$ e_1`|`e_2	$\rightsquigarrow$ e_2`<=`r `&&` r`<0`	[W-Or-Mix]
$e_1 \geq 0 \wedge e_2 \geq 0$	$\vdash_S$	r $\mathtt{op}_{ge}$ e_1`^`e_2	$\rightsquigarrow$ r`>=0`	[W-XOr-Pos]
$e_1 < 0 \wedge e_2 < 0$	$\vdash_S$	r $\mathtt{op}_{ge}$ e_1`^`e_2	$\rightsquigarrow$ r`>=0`	[W-XOr-Neg]
$e_1 \geq 0 \wedge e_2 < 0$	$\vdash_S$	r $\mathtt{op}_{le}$ e_1`^`e_2	$\rightsquigarrow$ r`<0`	[W-XOr-Mix]
$e_1 \geq 0$	$\vdash_S$	r $\mathtt{op}_{le}$ `~`e_1	$\rightsquigarrow$ r`<0`	[W-Cpl-Pos]
$e_1 < 0$	$\vdash_S$	r $\mathtt{op}_{ge}$ `~`e_1	$\rightsquigarrow$ r`>=0`	[W-Cpl-Neg]

$\mathtt{op}_{le} \in$ {`<`,`<=`,`==`, `:=`}, $\mathtt{op}_{ge} \in$ {`>`,`>=`,`==`, `:=`}, and $\mathtt{op}_{eq} \in$ {`==`, `:=`}

- Judgment $\mathcal{C} \vdash_S s_{bv} \rightsquigarrow s_{int}$ means under **condition** $\mathcal{C}$ bitvector statement s_{bv} can be approximated with linear statement s_{int}.
- Then, apply a substitution δ, and replace s_{bv} with `if-then-else` statement `if` $\mathcal{C}\delta$ `then assume(`$s_{int}\delta$`) else` s_{bv}

Weakening Rules and Termination

$$e_1 \geq 0 \wedge e_2 \geq 0 \quad \vdash_S \quad r\ \mathtt{op}_{le}\ e_1\&e_2 \quad \rightsquigarrow r\texttt{<=}e_1\ \texttt{\&\&}\ r\texttt{<=}e_2 \quad [\text{W-And-Pos}]$$

```
1 a = *;
2 assume(a>0);
3 while(x>0){
4   a--;
5   x = x & a;
6 }
```

Weakening Rules and Termination

$$e_1 \geq 0 \wedge e_2 \geq 0 \quad \vdash_S \quad r \ \mathtt{op}_{le} \ e_1 \& e_2 \quad \rightsquigarrow r <= e_1 \ \&\& \ r <= e_2 \quad [\text{W-AND-POS}]$$

```
1  a = *;
2  assume(a>0);
3  while(x>0){
4     a--;
5     x = x & a;
6  }
```

- $\mathcal{I} : x > 0 \wedge a > 0$
- $T : x{>}0 \wedge a'{=}a{-}1 \wedge x'{=}x\&a'$
- Tools fail to show:
  $\mathcal{I} \wedge T \wedge x'{>}0 \implies \mathcal{I}'$

Weakening Rules and Termination

$$e_1 \geq 0 \wedge e_2 \geq 0 \quad \vdash_S \quad r \ \mathrm{op}_{le} \ e_1 \& e_2 \quad \rightsquigarrow r <= e_1 \ \&\& \ r <= e_2 \quad \text{[W-And-Pos]}$$

```
1  a = *;
2  assume(a>0);
3  while(x>0){
4     a--;
5     x = x & a;
6  }
```

- $\mathcal{I} : x > 0 \wedge a > 0$
- $T : x{>}0 \wedge a'{=}a{-}1 \wedge x'{=}x\&a'$
- Tools fail to show:
  $\mathcal{I} \wedge T \wedge x'{>}0 \implies \mathcal{I}'$

```
1  a = *; assume(a > 0);
2  while (x > 0) {
3     { x > 0 ∧ a > 0 }
4     a--;
5     if (x >= 0 && a >= 0)
6     then { x = *; assume(x <= a); }
7     else { x = x & a; }
8  }
```

Weakening Rules and Termination

$$e_1 \geq 0 \wedge e_2 \geq 0 \quad \vdash_S \quad r \ \mathrm{op}_{le} \ e_1 \& e_2 \quad \rightsquigarrow r <= e_1 \ \&\& \ r <= e_2 \quad \text{[W-And-Pos]}$$

```
1  a = *;
2  assume(a>0);
3  while(x>0){
4     a--;
5     x = x & a;
6  }
```

- $\mathcal{I}: x > 0 \wedge a > 0$
- $T: x{>}0 \wedge a'{=}a{-}1 \wedge x'{=}x\&a'$
- Tools fail to show:
  $\mathcal{I} \wedge T \wedge x'{>}0 \implies \mathcal{I}'$

```
1  a = *; assume(a > 0);
2  while (x > 0) {
3     { x > 0 ∧ a > 0 }
4     a--;
5     if (x >= 0 && a >= 0)
6     then { x = *; assume(x <= a); }
7     else { x = x & a; }
8  }
```

- $T' = x{>}0 \wedge a' = a{-}1 \wedge ((x{\geq}0 \wedge a'{\geq}0 \wedge x'{\leq}a') \vee$ ~~$(\neg(x \geq 0 \wedge a' \geq 0) \wedge x' = x\&a')$~~$)$
- Tools can prove that $\mathcal{I} \wedge T' \wedge x'{>}0 \implies \mathcal{I}'$, ranking function $\mathcal{R}(x, a) = a$

Basic Notations

Standard notions of programs and semantics.

- $\Sigma : Var \to Val$, a state space mapping variables to values.
- $[\![exp]\!] : \Sigma \to Val$, expressions semantics.
- $[\![stmt]\!] : \Sigma \to \mathcal{P}(\Sigma)$, statements semantics.
- $[\![P]\!]$: traces of program P.

Define rewriting rules for expressions and statements.

- $T_E : exp \to exp$, rewriting rules application for expressions.
- $T_S : stmt \to stmt$, weakening rules application for statements.

Soundness Proof

Lemma (Rule correctness)

For every rule $\mathcal{C} \vdash_E e \rightsquigarrow e'$, $\forall \sigma.\ \mathcal{C}(\sigma) \Rightarrow e' = \sigma(e), [\![e]\!] = [\![e']\!]$.
For every rule $\mathcal{C} \vdash_S s \rightsquigarrow s'$, $\forall \sigma.\ \mathcal{C}(\sigma) \Rightarrow s' = \sigma(s), [\![s]\!] \subseteq [\![s']\!]$.

Theorem (Soundness)

For every P, T_E, T_S, $[\![P]\!] \subseteq [\![T_S\{T_E\{P\}\}]\!]$.

Prove by induction on traces, T_E preserves traces equivalence, T_S preserves trace inclusion.

Experiments: Reachability, Termination, LTL

Experiment Benchmarks

Limitations of SV-COMP & Ultimate

- No LTL verification tasks in SV-COMP.
- Rare cases with bit-vector operations.
- Bitvector operations that irrelevant to verification tasks.

Contributed Benchmarks

1. **ReachBitBench, TermBitBench, LTLBitBench** for reachability, termination, LTL verification, respectively.
2. **BitHacks**, online code optimization[a] adapted to termination, LTL verification.
3. **Decompiled**, compile from Ultimate[b] source to binaries, decompile.

[a] https://graphics.stanford.edu/~seander/bithacks.html
[b] https://github.com/ultimate-pa/ultimate/tree/dev/trunk/examples/LTL/

Implementation: UltimateBwB

Bitwise branching implementation

- A fork of Ultimate repository[a]
- Recursive AST transformation during Ultimate's translation from C to Boogie

[a] https://github.com/ultimate-pa/ultimate

Experiments: Reachability with various solvers

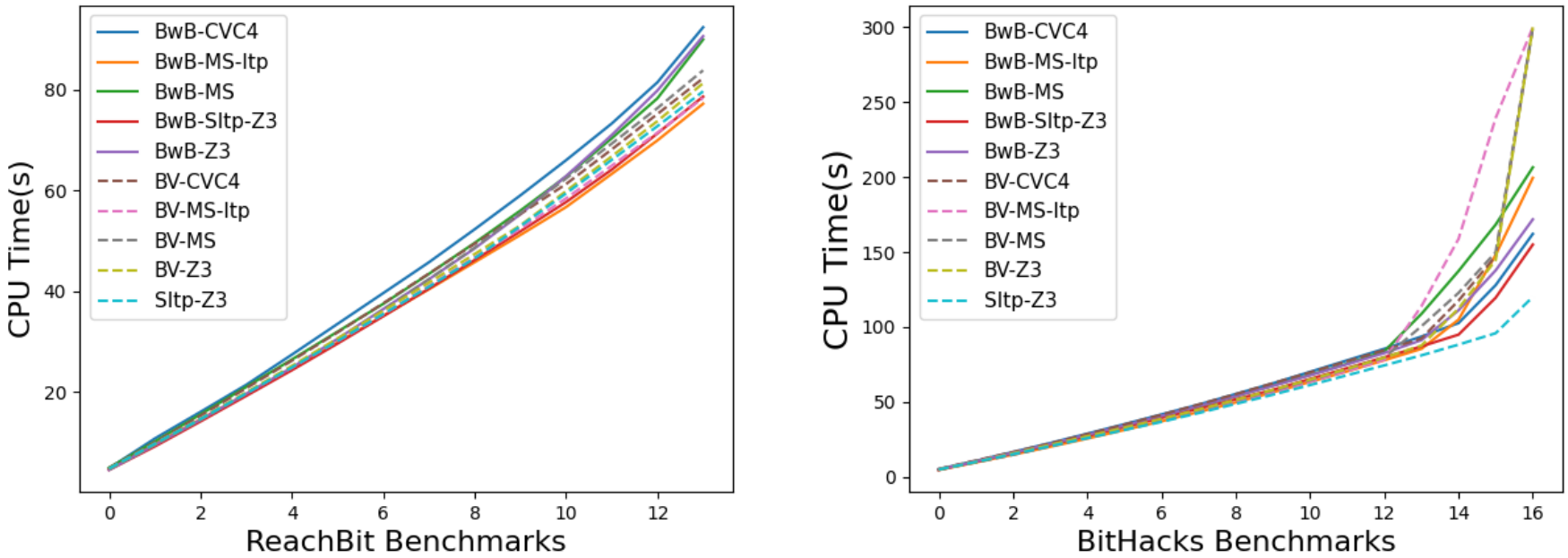


Figure: Performance of UltimateBwB with bitwise branching "BwB" in *integer mode* (solid lines) versus Ultimate (dashed lines, "BV" indicating *bitvector mode*) on bitvector programs, using various SMT solvers.
Default Ultimate (integer mode SItp-Z3) returns *Unknown* for 10/12 "ReachBit", 16/16 "BitHacks".

Experiments: Termination and LTL

State-of-the-art verification tools

Tool	BitVec.	Term.	LTL
ULTIMATE	Limited	Yes	Yes
APROVE	Yes	Yes	No
KITTEL	Yes	Yes	No
CPACHECKER	Limited	Yes	No
2LS	Yes	Yes	No
ULTIMATEBWB	Yes	Yes	Yes

Experiments: Termination

Termination Results Overview

	(ii) TermBitBench						(i) AproveBench					
	AProVE	CPAchecker	KITTeL	2LS	Ultimate	UltimateBwB	AProVE	CPAchecker	KITTeL	2LS	Ultimate	UltimateBwB
✔ (Terminating)	5	1	7	8	2	18	1	3	3	14	2	2
↯✔ (FN)	1	-	-	-	-	-	-	-	-	-	-	-
✗ (Nonterminating)	6	10	-	8	-	13	-	-	-	-	-	-
↯✗ (FP)	2	7	-	3	-	-	-	10	-	-	2	6
?(Unknown)	14	13	-	-	29	-	10	3	-	1	14	8
T (Time Out)	3	-	19	12	-	-	7	-	10	2	-	1
M (Out of Memory)	-	-	-	-	-	-	-	-	-	1	-	1
☢ (Crash)	-	-	5	-	-	-	-	2	5	-	-	-

- TermBitBench, 18 terminating, 13 non-terminating.
- AproveBench, 18/118 or 15% are bitvector programs.

Experiments: LTL

LTL Results Overview

	(iv) Bithacks		(iii) LTLBit Bench	
	ULTIMATE	w. BwB	ULTIMATE	w. BwB
✔ (Satisfied)	3	10	-	21
✗ (Unsatisfied)	-	7	-	20
?(Unknown)	21	5	42	-
T (Time Out)	1	1	-	1
M (Out of Memory)	1	3	-	-

- BitHacks, 18 satisfied, 8 violated.
- LTLBitBench, 22 programs satisfying LTL property, 20 programs are violated.

Case Study: Binary Decompilation

Binary Decompilation

Why binary verification & challenges

- Why: Executable is the one runs on machine, compiler errors, optimizations, proprietary software, malware etc..
- Challenges: Disassembly (function boundaries, symbol table, stack frame), control flow recovery etc..

Decompiled code needs bitwise branching

`if(x <= 1)`[a] in de-compiled code:

```
(((tmp_42!=0u)&1)&((((tmp_44 == 0u)&
1^(((((tmp_44^tmp_45)+tmp_45)==2u)&1)))&1)))&1
```

Decompilation is an approach for binary verification.

[a] http://github.com/ultimate-pa/ultimate/blob/dev/trunk/examples/LTL/simple/PotentialMinimizeSEVPABug.c

DarkSea Overview

Preparing decompiled programs for verification

- *Run-time environment.*
- *Passing emulation state through procedures.*
- *Nested structures.*
- *Property-directed slicing.* (See paper for details)

DarkSea: `https://github.com/cyruliu/darksea`

LTL Experiments on Decompiled Binaries

Table: ULTIMATE vs. DARKSEA on decompiled programs with LTL properties.

Benchmark	Property	Exp.	ULTIMATE Time	ULTIMATE Result	DARKSEA Time	DARKSEA Result
01-exsec2.s.c	$\Diamond(\Box x = 1)$	✔	4.45s	☢	11.23s	✔
01-exsec2.s.f.c.c	$\Diamond(\Box x \neq 1)$	✗	6.31s	☢	10.36s	✗
SEVPA_gccO0.s.c	$\Box(x > 0 \Rightarrow \Diamond y = 0)$	✔	6.31s	☢	22.92s	✔
SEVPA_gccO0.s.f.c	$\Box(x > 0 \Rightarrow \Diamond y = 2)$	✗	5.16s	?	14.92s	✗
acqrel.simplify.s.c	$\Box(x = 0 \Rightarrow \Diamond y = 0)$	✔	5.17s	☢	9.00s	✔
acqrel.simplify.s.f.c.c	$\Box(x = 0 \Rightarrow \Diamond y = 1)$	✗	6.06s	☢	17.60s	✗
exsec2.simplify.s.c	$\Box\Diamond x = 1$	✔	4.92s	☢	5.60s	✔
exsec2.simplify.s.f.c.c	$\Box\Diamond x \neq 1$	✗	4.55s	☢	6.28s	✗

Reachability, Termination and LTL

Contributions and findings

1. Source level bitwise branching.
2. Bitwise branching incurs negligible overhead.
3. Enlarge the domain of Termination and LTL verification.
4. Rich set of bitvector benchmarks for various verification tasks.
5. DarkSea tool chain prepares decompiled programs for verification.

Q & A

Thank You!

Proving LTL Properties of Bitvector Programs and Decompiled Binaries

Verify Lifted Code

Tools applied to lifted code

- APROVE: Errors in conversion from LLVM IR to internal representation.
- KITTEL: Parsing (from C to KITTEL's format via LLVM bitcode with LLVM2KITTEL) succeeded, but then KITTEL silently hung until timeout.
- CPACHECKER: Crashes on all benchmarks, while parsing system headers.
- ULTIMATE: Crashes on 3 benchmarks, due to inconsistent type exceptions.

Weakening Rules and Termination

A termination example:

```
1  a = *;
2  assume (a >0);
3  while (x >0){
4     a--;
5   x = x & a;
6  }
```

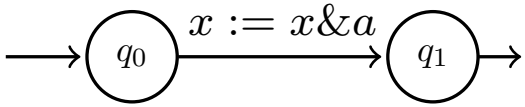
$\mathcal{I} : x > 0 \wedge a > 0$

$T : x{>}0 \wedge a'{=}a{-}1 \wedge x'{=}x\&a'$

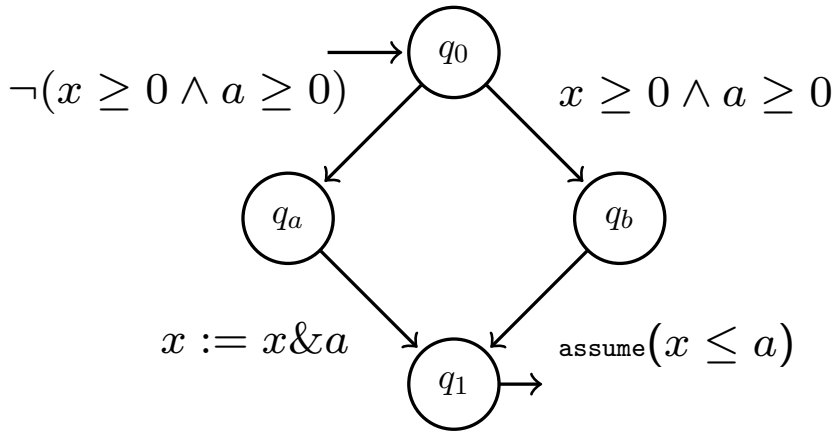
Tools fail to show

$\mathcal{I} \wedge T \wedge x'{>}0 \implies \mathcal{I}'$

Control flow automaton transformation ([W-And-Pos]):



translated into:



$T' : \mathcal{I} \wedge T' \wedge x'{>}0 \implies \mathcal{I}', \mathcal{R}(x, a) = a$