DarkSea: A Toolchain for Automatic Verification of Lifted Binaries

Cyrus Liu

Stevens Institute of Technology
Cypress Lab

Empire Hacking • 12th April 2023

CYPRESS
CYBERSECURITY
PROGRAMMING LANGUAGES
AND SYSTEMS AT STEVENS

Verification of Lifted Binaries

Why binary verification & challenges

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

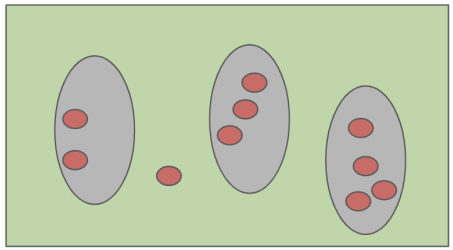
Verification of lifted code needs bitvector reasoning

`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
```

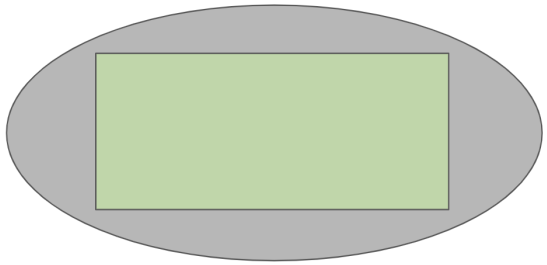
[a] http://github.com/ultimate-pa/ultimate/blob/dev/trunk/examples/LTL/simple/PotentialMinimizeSEVPABug.c

Verification Background



Testing

- Fuzzing.
- Symbolic execution.
- Constraint solving.
- Taint analysis.



Formal Verification

- Model checking / automaton.
- Abstract interpretation.
- Mathematical Logics.
- SMT Solving.

Formal Verification

Automated software verification: $P \vDash \varphi$

Challenges:

- Types of program P (Decompiled binaries).
- Assertion logic of φ (Temporal logics).

Program properties (φ):

- **Reachability:** program never reaches a bad state (err).
- **Termination:** program always exits ($exit$).

Linear Temporal Logic (LTL)

A generalization that describes how program behaviors change over time.

Benchmarks Repositories (SVCOMP [1] and ULTIMATE [2])

Programs with Linear Arithmetic

```
if ( (b - c >= 1) && (a == c)) {
        int r =  random();
        b = 10;
        c = c + 1 + r;
```

```
unsigned int y = w + 1;
unsigned int z = x + 1;
```

```
if(nondet()) {
    c--; curr_serv--;
    resp++;
```

```
while (j < m) {
    j = j + 1;
    k = i;
    while (k < N - 1) {
        k = k + 1;
```

```
if( WarmPollPeriod > 20 ) {
    WarmPollPeriod = 20;
}
```

Programs with Non-linear Arithmetic (NLA)

Bitvectors

```
for (v >>= 1; v; v >>= 1) {
    r <<= 1;
    r |= v & 1;
    s--;
  }
r <<= s; // shift when v's highest bits are zero
```

De-compiled Binaries

```
if (((((((tmp__18 == 0u)&1)) && ( tmp__15 != 0u) )&1))) {
tmp__14__PHI_TEMPORARY = tmp__15;   /* for PHI node */
goto block_401135;
```

Polynomials

```
while (1) {
    __VERIFIER_assert(4*x - y*y*y*y - 2*y*y*y - y*y == 0);
    if (!(c < k))
        break;
    c = c + 1;
    y = y + 1;
    x = y * y * y + x;    }
__VERIFIER_assert(k*y - (y*y) == 0);
__VERIFIER_assert(4*x - y*y*y*y - 2*y*y*y - y*y == 0);
```

Others (logarithm, square root etc.)

[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

[2] https://github.com/ultimate-pa/ultimate/tree/dev/trunk/examples/LTL

Problems in Temporal Verification of Lifted Binaries

Why can't we currently verify these Lifted Binaries?

Static verification tools are limited for NLA programs:

- Complexity in bitvector SMT solving.
- Unable to find invariants and rank functions (needed for termination, LTL) for NLA programs.
- Some bitvector reachability/termination techniques
- **But not for LTL**.

Solutions

LTL Verification of Bitvector Programs with Bitwise Branching[a]

[a] Yuandong Cyrus Liu, Chengbin Pang, Daniel Dietsch, Eric Koskinen, Ton-Chanh Le, Georgios Portokalidis, and Jun Xu. Proving LTL Properties of Bitvector Programs and Decompiled Binaries, APLAS 2021.

Automatic Verification with DarkSea[b,c,d,e]

[b] Yuandong Cyrus Liu, Poster Session, FMCAD-SF 2021.

[c] Open Source, https://github.com/cyruliu/darksea

[d] APLAS'21 Extended, arXiv: 2105.05159

Rewriting Rules & Weakening Rules for Bitwise Branching

Formalized Bitwise Branching Rules [3]

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|e_2	$\rightsquigarrow e_1$	[R-Or-0]
$(e_1 = 0 \vee e_1 = 1) \wedge e_2 = 1$	$\vdash_E$	e_1|e_2	$\rightsquigarrow 1$	[R-Or-1]
$e_2 = 0$	$\vdash_E$	e_1^e_2	$\rightsquigarrow e_1$	[R-Xor-0]
$e_1 = e_2 = 0 \vee e_1 = e_2 = 1$	$\vdash_E$	e_1^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^e_2	$\rightsquigarrow 1$	[R-Xor-Neq]
$e_1 \geq 0 \wedge e_2 =$ `CHAR_BIT * sizeof`$(e_1) - 1$	$\vdash_E$	e_1>>e_2	$\rightsquigarrow 0$	[R-RightShift-Pos]
$e_1 < 0 \wedge e_2 =$ `CHAR_BIT * sizeof`$(e_1) - 1$	$\vdash_E$	e_1>>e_2	$\rightsquigarrow -1$	[R-RightShift-Neg]

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 $\mathtt{op}_{ge}$ 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]

- 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 $\mathcal{C}\delta \ ? \ e_{int}\delta : e_{bv}$

```
x = x & b;  ⇝  if (x==0 || b==0) x = 0; else x = x & b;
```

[3] Liu, Yuandong Cyrus, et al. "Proving LTL Properties of Bitvector Programs and Decompiled Binaries." Asian Symposium on Programming Languages and Systems. Springer, Cham, 2021.

Implementation and Benchmarks

Bitwise branching implementation

- A fork of ULTIMATE repository (now merged back).[a]
- Recursive AST transformation during ULTIMATE's translation from C to Boogie.

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

Contributed Benchmarks

1. **ReachBitBench, TermBitBench, LTLBitBench** for reachability, termination, LTL verification, respectively.
2. **BitHacks**, online code optimization[b] adapted to termination, LTL verification.

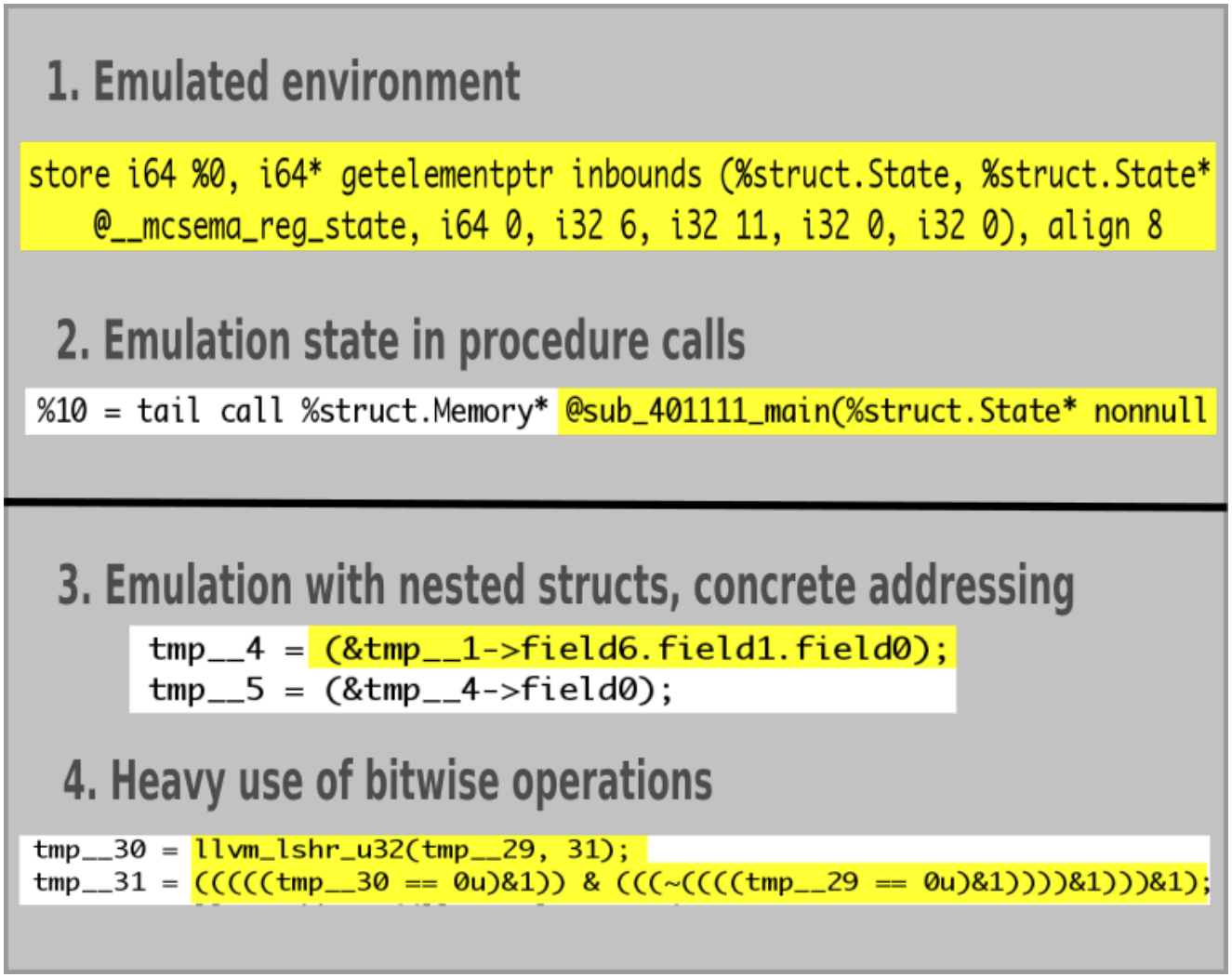
[b] https://graphics.stanford.edu/~seander/bithacks.html

Lifted Binary Code Block

```
  tmp__4 = globalState;
  tmp__5 = (&tmp__4->field2.field1);
  tmp__6 = (&tmp__4->field2.field3);
  tmp__7 = (&tmp__4->field2.field7);
  tmp__8 = (&tmp__4->field2.field9);
  tmp__9 = (&tmp__4->field2.field13);
  tmp__10 = (&tmp__4->field2.field5);
  goto block_401119;

  do {     /* Syntactic loop 'block_401119' to make GCC happy */
block_401119:
    STATE_REG_RAX = __VERIFIER_nondet_int();

    tmp__11 =  /*tail*/ sub_401106___VERIFIER_nondet_int(/*UNDEF*/((
    struct l_struct_struct_OC_State*)/*NULL*/0), /*UNDEF*/(0UL), tmp__3)
    ;
    tmp__12 = STATE_REG_RAX;
    x = tmp__12;
    y = 1;
    tmp__13 = tmp__12 >> 31;


  /* if (((((((tmp__13 == 0u)&1)) & (((~((((tmp__12 == 0u)&1))))&1)))&1)
    )) { */
    if (((((tmp__13 == 0u)&1)) && (tmp__12 != 0u))&1) {

    tmp_14__PHI_TEMPORARY = tmp__12;   /* for PHI node */
    goto block_401135;
  } else {
    _2e_lcssa3__PHI_TEMPORARY = tmp__12;   /* for PHI node */
    _2e_lcssa2__PHI_TEMPORARY = (((tmp__12 == 0u)&1));   /* for PHI node
     */
    _2e_lcssa1__PHI_TEMPORARY = tmp__13;   /* for PHI node */
    goto block_401163;
  }

  do {     /* Syntactic loop 'block_401135' to make GCC happy */
block_401135:
  tmp__14 = tmp__14__PHI_TEMPORARY;
  tmp__15 = tmp__14-1;
  tmp__16 = tmp__14-2;
  tmp__17 = tmp__16>>31;
  tmp__18 = tmp__15>>31;

  /* if (((((((tmp__16 != 0u)&1)) & ((((((tmp__17 == 0u)&1)) ^ ((((
    llvm_add_u32((tmp__17 ^ tmp__18), tmp__18)) == 2u)&1)))&1)))&1))) {
    */
  if ( ((tmp__16 != 0u)&1) &&(tmp__17 == 0u)) {
    goto block_401159_2e_backedge;
  } else {
    goto block_40114f;
  }
```

Challenges Through An Example De-compiled Binary

Lifting tools often generate programs emulate the machine behaviors.



DarkSea Overview

DarkSea

1. Translations to re-target lifting for verification:
   - Run-time environment.
   - Replace emulation state with a global pointer.
   - Nested structures.
   - Property-directed slicing (DG).
2. Employ Bitwise Branching (&: [R-And-0], [R-And-1]).

Details in extended archive paper[a] and summarized in the APLAS'21 paper

[a] https://arxiv.org/abs/2105.05159

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

LTL Experiments on Lifted Binaries

Fist tool for temporal verification of 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	✗

Summary

Bringing formal methods into de-compilation process, disassembly, lifting.

- IDAPro, Binary Ninja (BNIL $\rightarrow$ MLIL $\rightarrow$ LLVM), Angr (VEX IR).
- Gihdra, RetDec, Radare2.
- DecFox (proof assistant), Smack (Boogie IR), SeaHorn (LLVM IR).

MLIR (hardware specific operations) $\rightarrow$ LLVM IR $\rightarrow$ Boogie IR $\rightarrow$ Verifiers

Temporal Behaviors in Smart Contracts

Flavor of Smart Contracts

"In an auction smart contract, all non-winning bidders should be eventually refunded their bid amount."

- Slither, static analysis framework, detector.
- KEVM, K Framework, formal K Semantics for EVM.
- SmartPulse, temporal verification of smart contract (sodility on EVM), SolVeri tool, can't handle bitwise operations.

Q & A

Thank You!