ISP RAS

Sydr

Cutting Edge Dynamic Symbolic Execution

Alexey Vishnyakov, Andrey Fedotov, Daniil Kuts, Alexander Novikov, Darya Parygina, Eli Kobrin, Vlada Logunova, Pavel Belecky and Shamil Kurmangaleev

December 11, 2020

ISP RAS

arxiv.org/abs/2011.09269

Motivation

- The security development lifecycle (SDL) is becoming an industry standard
- Dynamic symbolic execution (DSE) reinforce fuzzing, critical defects detection, software certification, etc.

Dynamic Symbolic Execution

- *Dynamic symbolic execution* explore variation of input data on a selected program execution path
  - i.e., we symbolically execute program on a fixed input data
- Each input byte is modeled by a free *symbolic variable*
- Interpreted instructions produce SMT formulas (over constants and symbolic variables) according to corresponding operational semantics
- *Symbolic state* maps registers and memory bytes to SMT formulas
- *Path predicate* contains taken branch constraints
  - i.e., represents the explored path
  - Solution to conjunction of these constraints is an input data to reproduce the same execution path

E. J. Schwartz et al. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)

Exploring New Paths

- We symbolically execute program to invert branches
- Thus, we discover new paths
- Target branch constraint is negated
- **Input:** `"bac"`
- **Path predicate:**
  $\alpha_0 \neq c \wedge \alpha_1 \neq c \wedge \alpha_2 = c$
- **Invert last branch:**
  $\alpha_0 \neq c \wedge \alpha_1 \neq c \wedge \alpha_2 \neq c$
- **Model:** `"bad"`

```
char *
find(const char *s, char c)
{
  // Buffer overrun.
  while (*s != c) ++s;
  return s;
}

int
main(int argc, char **argv)
{
  char *c = find(argv[0], 'c');
  printf("%c\n", *c);
}
```

Contributions

Symbolic execution accuracy and performance improvement:

- Skipping non-symbolic instructions
- Symbolic AST simplification
- Path predicate slicing
- Indirect jumps resolving
- Handling multi-threaded programs

We present Sydr (Symbolic DynamoRIO) – dynamic symbolic execution tool combining DynamoRIO for concrete execution and Triton for symbolic execution.

Skipping Non-Symbolic Instructions

- Path predicate builds faster when skipping non-symbolic instructions
- We retrieve all explicit and implicit instruction operands from DynamoRIO
- Sydr symbolically executes instruction iff any of its read/write registers, memory, or flags are symbolic

AST Simplification

- Triton uses intermediate AST representation that is later translated to SMT
- Kill taint: $A \oplus A \to 0$, $0 * A \to 0$, $A - A \to 0$, etc.
- `((_ extract 11 9) (concat (_ bv1 8) (_ bv2 8) (_ bv3 8) (_ bv4 8)))` $\to$ `((_ extract 3 1) (_ bv3 8))`
  - Triton symbolic context stores AST for each parent register
  - `mov rax, symbolic_variable ; mov al, 0x00 ; test al, al ; jz 0xdeadbeef`
  - `jz` branch won't be symbolic
- `((_ extract 31 0) ((_ zero_extend 32) (_ bv1 32)))` $\to$ `(_ bv1 32)`
  - 32-bit GPR registers on x86-64 are zero-extended
- etc.

Path Predicate Slicing

- Path predicate should conjunct only constraints relevant to inverting the target branch
- Conjuncts contain symbolic variables that transitively depend on variables in the target branch constraint
- Solver consumes less memory and time
- Slicing removes possibly underconstrained symbolic variables
- Solver returns a model for a subset of input bytes
- Other bytes are taken from initial input

C. Cadar et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs

Path Predicate Slicing Algorithm

Input: *cond* – predicate for target branch inversion,
Π – path predicate (path constraints prior to the target branch).

$vars \leftarrow used\_variables(cond)$ ▷ slicing variables
$change \leftarrow vars$
while $change \neq \varnothing$ **do**
    $change \leftarrow vars$
    for all $c \in \Pi$ **do** ▷ iterate over path constraints
        if $vars \cap used\_variables(c) \neq \varnothing$ **then**
            $vars \leftarrow vars \cup used\_variables(c)$
    $change \leftarrow vars \setminus change$
$\Pi_S \leftarrow cond$ ▷ predicate for branch inversion
for all $c \in \Pi$ **do** ▷ iterate over path constraints
    if $vars \cap used\_variables(c) \neq \varnothing$ **then**
        $\Pi_S \leftarrow \Pi_S \wedge c$
return Π_S

Path Predicate Slicing Example

```c
1  char* syms = "SLICING FIX IT!\n";
2  // b - input data.
3  int len = strlen(syms);
4  if (b[0] < len)
5    if (syms[b[0] % len] == '!')
6      if (b[2] > '@')
7        if (b[5] + b[4] < 'B')
8          if (b[3] + b[5] > '@')
9            if (b[1] + b[3] > '@')
10             if (b[4] < '9')
11               if (b[1] > '@')
12                 // Target branch.
13                 printf("OK\n");
14               else
15                 // Initial path.
16                 printf("FAIL\n");
```

b[0] in line 5 is underconstrained.

Path Predicate Slicing Example

```
1  char* syms = "SLICING FIX IT!\n";
2  // b - input data.
3  int len = strlen(syms);
4  if (b[0] < len)
5    if (syms[b[0] % len] == '!')
6      if (b[2] > '@')
7        if (b[5] + b[4] < 'B')
8          if (b[3] + b[5] > '@')
9            if (b[1] + b[3] > '@')
10             if (b[4] < '9')
11               if (b[1] > '@')
12                 // Target branch.
13                 printf("OK\n");
14               else
15                 // Initial path.
16                 printf("FAIL\n");
```

b[0] in line 5 is underconstrained.

Line	Slicing variables
11	b[1]

Path Predicate Slicing Example

```
1  char* syms = "SLICING FIX IT!\n";
2  // b - input data.
3  int len = strlen(syms);
4  if (b[0] < len)
5    if (syms[b[0] % len] == '!')
6      if (b[2] > '@')
7        if (b[5] + b[4] < 'B')
8          if (b[3] + b[5] > '@')
9            if (b[1] + b[3] > '@')
10             if (b[4] < '9')
11               if (b[1] > '@')
12                 // Target branch.
13                 printf("OK\n");
14               else
15                 // Initial path.
16                 printf("FAIL\n");
```

b[0] in line 5 is underconstrained.

Line	Slicing variables
11	b[1]
9	b[1], b[3]

Path Predicate Slicing Example

```
1  char* syms = "SLICING FIX IT!\n";
2  // b - input data.
3  int len = strlen(syms);
4  if (b[0] < len)
5    if (syms[b[0] % len] == '!')
6      if (b[2] > '@')
7        if (b[5] + b[4] < 'B')
8          if (b[3] + b[5] > '@')
9            if (b[1] + b[3] > '@')
10             if (b[4] < '9')
11               if (b[1] > '@')
12                 // Target branch.
13                 printf("OK\n");
14               else
15                 // Initial path.
16                 printf("FAIL\n");
```

b[0] in line 5 is underconstrained.

Line	Slicing variables
11	b[1]
9	b[1], b[3]
8	b[1], b[3], b[5]

Path Predicate Slicing Example

```c
1  char* syms = "SLICING FIX IT!\n";
2  // b - input data.
3  int len = strlen(syms);
4  if (b[0] < len)
5    if (syms[b[0] % len] == '!')
6      if (b[2] > '@')
7        if (b[5] + b[4] < 'B')
8          if (b[3] + b[5] > '@')
9            if (b[1] + b[3] > '@')
10             if (b[4] < '9')
11               if (b[1] > '@')
12                 // Target branch.
13                 printf("OK\n");
14               else
15                 // Initial path.
16                 printf("FAIL\n");
```

b[0] in line 5 is underconstrained.

Line	Slicing variables
11	b[1]
9	b[1], b[3]
8	b[1], b[3], b[5]
7	b[1], b[3], b[4], b[5]

Path Predicate Slicing Example

```
1  char* syms = "SLICING FIX IT!\n";
2  // b - input data.
3  int len = strlen(syms);
4  if (b[0] < len)
5    if (syms[b[0] % len] == '!')
6      if (b[2] > '@')
7        if (b[5] + b[4] < 'B')
8          if (b[3] + b[5] > '@')
9            if (b[1] + b[3] > '@')
10             if (b[4] < '9')
11               if (b[1] > '@')
12                 // Target branch.
13                 printf("OK\n");
14               else
15                 // Initial path.
16                 printf("FAIL\n");
```

b[0] in line 5 is underconstrained.

Line	Slicing variables
11	b[1]
9	b[1], b[3]
8	b[1], b[3], b[5]
7	b[1], b[3], b[4], b[5]
10	b[1], b[3], b[4], b[5]

Jump Tables

`switch` statements may produce jump tables

Address tables:

```
...
mov    rax, [rax * 8 + 0x400688]
jmp    rax
...
```

```
...
lea    rdx, [rax * 8]
lea    rax, [rip + 0x200872]
mov    rax, [rdx + rax]
call   rax
...
```

Offset table:

```
...
mov    eax, [rdx + rax]
movsxd rdx, eax
lea    rax, [rip + 0x110]
add    rax, rdx
jmp    rax
...
```

Indirect Jumps Resolving

- We perform backward slicing from indirect jump within a current basic block
- Thus, we locate an instruction that reads the target address from memory
- We create path constraints for the indirect jump
- A condition for each branch is an equality of the symbolic pointer expression and the corresponding jump table entry address

Indirect Jumps Resolving Example

```
switch (a) {
    case 2:
        <code_c2>
    case 3:
        <code_c3>
    case 6:
        <code_c6>
    default:
        <code_default>
}
```

```
sub    eax, 0x2
cmp    eax, 0x4
ja     _code_default
mov    edx, _table_start
jmp    [edx + eax * 0x4]
```

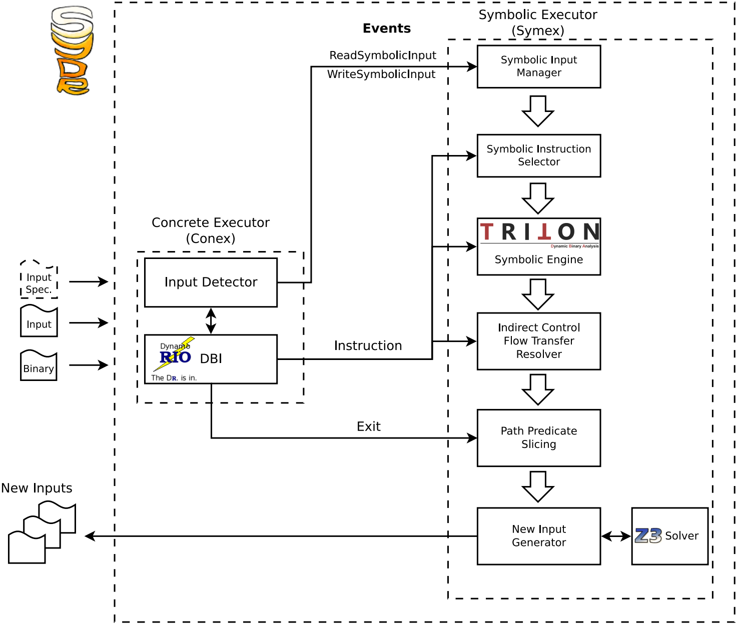
address	value
_table_start + 0x0	_code_c2
_table_start + 0x4	_code_c3
_table_start + 0x8	_code_default
_table_start + 0xc	_code_default
_table_start + 0x10	_code_c6

```
Branch 2:
  (sym_addr = _table_start)
Branch 3:
  (sym_addr = _table_start + 0x4)
Branch 6:
  (sym_addr = _table_start + 0x10)
Branch default:
  (sym_addr = _table_start + 0x8) or
  (sym_addr = _table_start + 0xc)
```

Multi-Threaded Programs Symbolic Execution

- Threads share memory but have their own register values
- All threads have a shared path predicate storage
- We maintain a thread contexts storage that contains symbolic registers for each thread
- During thread switching we save all symbolic registers and replace them with symbolic registers for the current thread

Sydr Architecture



Future Work

- Security predicates
  - We have already partially developed null pointer dereference, zero division, out of bounds access, and integer overflow checkers
- Modeling function semantics (`tolower`/`toupper` are interesting because they constrain a symbol case)
  - We tried skipping `malloc` function that increases accuracy and significantly reduces number of UNSAT branches
- Symbolic addresses
- Z3-solver tactics

DEMO

1. Inverting branches in readelf
2. Integer overflow security predicate

Evaluation

- Single-threaded 64-bit Linux executables
- Sydr inverts branches from first to last
- Each test is executed up to 2 hours
- We limit path predicate construction time to 20 minutes
- *Accuracy* – percent of generated inputs (SAT) that have the same execution trace as original except the last branch in inverted direction

Path Predicate Construction Time (X1.2–3.5 Speedup)

Application	Input Size	Branch Count	App Time	Path Predicate Time Base	Skip	X
bzip2recover	147b	5131	0.0018s	9s	5s	1.8
cjpeg	12K	8010	0.0017s	39s	16s	2.4
faad	33K	470588	0.0082s	46m35s	18m7s	2.6
foo2lava	34K	910725	0.0045s	22m32s	18m42s	1.2
hdp	530K	67478	0.0021s	1m6s	41s	1.6
jasper	198K	837669	0.0037s	—	14m11s	—
libxml2	453b	53699	0.0024s	1m5s	34s	1.9
minigzip	19K	8977	0.0023s	2m44s	58s	2.8
muraster	887b	7102	0.0024s	7s	3s	2.3
pk2bm	1.7K	3673	0.0018s	4s	2s	2.0
pnmhistmap_pgm	198K	967187	0.0038s	14m37s	7m55s	1.8
pnmhistmap_ppm	12K	8121	0.0021s	29s	11s	2.6
readelf	8.3K	64196	0.0019s	1m19s	36s	2.2
yices-smt2	2K	19543	0.0029s	26s	14s	1.9
yodl	280b	4831	0.0017s	21s	6s	3.5

Path Predicate Slicing

Application	Slicing disabled			Slicing enabled		
	Accuracy	SAT	Queries	Accuracy	SAT	Queries
bzip2recover	100.0%	2101	5131	100.0%	2101	5131
cjpeg	100.0%	50	198	100.0%	50	197
faad	99.23%	389	585	99.07%	430	652
foo2lava	87.1%	31	6252	87.1%	31	6127
hdp	25.0%	464	2427	78.01%	1037	3828
jasper	0.05%	1987	5639	99.53%	6798	18207
libxml2	12.46%	1043	13520	50.98%	1069	17532
minigzip	10.73%	3961	4183	51.47%	7569	8977
muraster	99.97%	3235	4739	99.97%	3228	4726
pk2bm	98.91%	183	3672	99.45%	183	3673
pnmhistmap_pgm	99.97%	3159	4681	99.99%	17089	25446
pnmhistmap_ppm	99.07%	107	8247	99.07%	107	8247
readelf	61.93%	218	2046	86.47%	739	6141
yices-smt2	2.5%	521	2135	78.33%	2699	9647
yodl	8.31%	313	5201	57.51%	313	5201

Parallel Solving

Application	Number of Threads			
	1	2	4	8
bzip2recover	51m3s	25m57s	13m35s	10m12
cjpeg	—	—	63m12s	24m8s
minigzip	29m42s	17m18s	9m13s	6m49s
pk2bm	21m39s	11m21s	5m47s	3m1s
pnmhistmap_ppm	28m52s	14m20s	7m34s	4m14s
yodl	34m59s	16m54s	9m14s	5m23s

Questions?