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
Exploring New Paths

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

```c
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.
• Triton uses intermediate AST representation that is later translated to SMT

• Kill taint: \( A \oplus A \rightarrow 0, 0 \times A \rightarrow 0, A - A \rightarrow 0 \), etc.

• \((( \_ \text{extract} 11 \ 9) \ (\text{concat} \ (\_ \ \text{bv1} \ 8) \ (\_ \ \text{bv2} \ 8) \ (\_ \ \text{bv3} \ 8) \ (\_ \ \text{bv4} \ 8))) \rightarrow ((\_ \text{extract} 3 \ 1) \ (\_ \ \text{bv3} \ 8))\)
  
  • 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

• \(((\_ \text{extract} 31 \ 0)) \ (((\_ \ \text{zero_extend} \ 32) \ (\_ \ \text{bv1} \ 32))) \rightarrow (\_ \ \text{bv1} \ 32)\)
  
  • 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
**Path Predicate Slicing Algorithm**

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

\[
\begin{align*}
\text{vars} & \leftarrow \text{used\_variables}(cond) & \triangleright \text{slicing variables} \\
\text{change} & \leftarrow \text{vars} \\
\text{while } \text{change} \neq \emptyset \text{ do} \\
& \quad \text{change} \leftarrow \text{vars} \\
& \quad \text{for all } c \in \Pi \text{ do} & \triangleright \text{iterate over path constraints} \\
& \quad \quad \text{if } \text{vars} \cap \text{used\_variables}(c) \neq \emptyset \text{ then} \\
& \quad \quad \quad \text{vars} \leftarrow \text{vars} \cup \text{used\_variables}(c) \\
& \quad \quad \quad \text{change} \leftarrow \text{vars} \setminus \text{change} \\
\Pi_S & \leftarrow \text{cond} & \triangleright \text{predicate for branch inversion} \\
\text{for all } c \in \Pi \text{ do} & \triangleright \text{iterate over path constraints} \\
& \quad \text{if } \text{vars} \cap \text{used\_variables}(c) \neq \emptyset \text{ then} \\
& \quad \quad \text{if } \Pi_S \cap \text{used\_variables}(c) \neq \emptyset \text{ then} \\
& \quad \quad \quad \Pi_S \leftarrow \Pi_S \land c \\
\text{return } \Pi_S
\end{align*}
\]
Path Predicate Slicing Example

```c
char* syms = "SLICING FIX IT!\n";

// b - input data.
int len = strlen(syms);
if (b[0] < len)
    if (syms[b[0] % len] == '!')
        if (b[2] > '@')
                    if (b[1] + b[3] > '@')
                        if (b[4] < '9')
                            if (b[1] > '@')
                                // Target branch.
                                printf("OK\n");
                            else
                                // Initial path.
                                printf("FAIL\n");

b[0] in line 5 is underconstrained.
```
Path Predicate Slicing Example

```c
char* syms = "SLICING FIX IT!\n";
// b - input data.
int len = strlen(syms);
if (b[0] < len)
    if (syms[b[0] % len] == '!

if (b[2] > '@


if (b[4] < '9'

if (b[1] > '@

   // Target branch.
   printf("OK\n");
else
   // Initial path.
   printf("FAIL\n");

   // Initial path.
   printf("FAIL\n");
```

b[0] in line 5 is underconstrained.

Line    Slicing variables
  11    b[1]
Path Predicate Slicing Example

```c
char* syms = "SLICING FIX IT!\n";

// b - input data.
int len = strlen(syms);
if (b[0] < len)
    if (syms[b[0] % len] == '!')
        if (b[2] > '@')
                    if (b[1] + b[3] > '@')
                        if (b[4] < '9')
                            if (b[1] > '@')
                                // Target branch.
                                printf("OK\n");
                            else
                                // Initial path.
                                printf("FAIL\n");
```
Path Predicate Slicing Example

```c
char* syms = "SLICING FIX IT!\n";

// b - input data.
int len = strlen(syms);
if (b[0] < len)
  if (syms[b[0] % len] == '!
    if (b[2] > '@
            if (b[4] < '9'
              if (b[1] > '@'
                // Target branch.
                printf("OK\n");
              else
                // Initial path.
                printf("FAIL\n");
            endif
          endif
        endif
      endif
    endif
  endif
endif
```

b[0] in line 5 is underconstrained.

<table>
<thead>
<tr>
<th>Line</th>
<th>Slicing variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>b[1]</td>
</tr>
<tr>
<td>9</td>
<td>b[1], b[3]</td>
</tr>
<tr>
<td>8</td>
<td>b[1], b[3], b[5]</td>
</tr>
</tbody>
</table>
char* syms = "SLICING FIX IT!\n";
// b - input data.
int len = strlen(syms);
if (b[0] < len)
  if (syms[b[0] % len] == '!')
    if (b[2] > '@')
          if (b[1] + b[3] > '@')
            if (b[4] < '9')
              if (b[1] > '@')
                // Target branch.
                printf("OK\n");
          else
            // Initial path.
            printf("FAIL\n");
Path Predicate Slicing Example

```c
char* syms = "SLICING FIX IT!\n";

// b - input data.
int len = strlen(syms);
if (b[0] < len)
    if (syms[b[0] % len] == '!
')
        if (b[2] > '@
')
')
')
                        if (b[4] < '9'
)
                            if (b[1] > '@'
)
                    // Target branch.
                    printf("OK\n");
            else
                // Initial path.
                printf("FAIL\n");
```

b[0] in line 5 is underconstrained.

<table>
<thead>
<tr>
<th>Line</th>
<th>Slicing variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>b[1]</td>
</tr>
<tr>
<td>9</td>
<td>b[1], b[3]</td>
</tr>
<tr>
<td>8</td>
<td>b[1], b[3], b[5]</td>
</tr>
<tr>
<td>7</td>
<td>b[1], b[3], b[4], b[5]</td>
</tr>
<tr>
<td>10</td>
<td>b[1], b[3], b[4], b[5]</td>
</tr>
</tbody>
</table>
Jump Tables

switch statements may produce jump tables

Address tables:

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

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

Offset table:

```plaintext
...  mov  eax, [rdx + rax]
    movsx  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

```c
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]
```

<table>
<thead>
<tr>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>_table_start + 0x0</td>
<td>_code_c2</td>
</tr>
<tr>
<td>_table_start + 0x4</td>
<td>_code_c3</td>
</tr>
<tr>
<td>_table_start + 0x8</td>
<td>_code_default</td>
</tr>
<tr>
<td>_table_start + 0xc</td>
<td>_code_default</td>
</tr>
<tr>
<td>_table_start + 0x10</td>
<td>_code_c6</td>
</tr>
</tbody>
</table>

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)
• 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
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
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

17/20
<table>
<thead>
<tr>
<th>Application</th>
<th>Input Size</th>
<th>Branch Count</th>
<th>App Time</th>
<th>Path Predicate Time</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2recover</td>
<td>147b</td>
<td>5131</td>
<td>0.0018s</td>
<td>9s</td>
<td>5s</td>
</tr>
<tr>
<td>cjjpeg</td>
<td>12K</td>
<td>8010</td>
<td>0.0017s</td>
<td>39s</td>
<td>16s</td>
</tr>
<tr>
<td>faad</td>
<td>33K</td>
<td>470588</td>
<td>0.0082s</td>
<td>46m35s</td>
<td>18m7s</td>
</tr>
<tr>
<td>foo2lava</td>
<td>34K</td>
<td>910725</td>
<td>0.0045s</td>
<td>22m32s</td>
<td>18m42s</td>
</tr>
<tr>
<td>hdp</td>
<td>530K</td>
<td>67478</td>
<td>0.0021s</td>
<td>1m6s</td>
<td>41s</td>
</tr>
<tr>
<td>jasper</td>
<td>198K</td>
<td>837669</td>
<td>0.0037s</td>
<td>—</td>
<td>14m11s</td>
</tr>
<tr>
<td>libxml2</td>
<td>453b</td>
<td>53699</td>
<td>0.0024s</td>
<td>1m5s</td>
<td>34s</td>
</tr>
<tr>
<td>minigzip</td>
<td>19K</td>
<td>8977</td>
<td>0.0023s</td>
<td>2m44s</td>
<td>58s</td>
</tr>
<tr>
<td>muraster</td>
<td>887b</td>
<td>7102</td>
<td>0.0024s</td>
<td>7s</td>
<td>3s</td>
</tr>
<tr>
<td>pk2bm</td>
<td>1.7K</td>
<td>3673</td>
<td>0.0018s</td>
<td>4s</td>
<td>2s</td>
</tr>
<tr>
<td>pnmhistmap_pgm</td>
<td>198K</td>
<td>967187</td>
<td>0.0038s</td>
<td>14m37s</td>
<td>7m55s</td>
</tr>
<tr>
<td>pnmhistmap_ppm</td>
<td>12K</td>
<td>8121</td>
<td>0.0021s</td>
<td>29s</td>
<td>11s</td>
</tr>
<tr>
<td>readelf</td>
<td>8.3K</td>
<td>64196</td>
<td>0.0019s</td>
<td>1m19s</td>
<td>36s</td>
</tr>
<tr>
<td>yices-smt2</td>
<td>2K</td>
<td>19543</td>
<td>0.0029s</td>
<td>26s</td>
<td>14s</td>
</tr>
<tr>
<td>yodl</td>
<td>280b</td>
<td>4831</td>
<td>0.0017s</td>
<td>21s</td>
<td>6s</td>
</tr>
</tbody>
</table>
## Path Predicate Slicing

<table>
<thead>
<tr>
<th>Application</th>
<th>Slicing disabled</th>
<th></th>
<th>Slicing enabled</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Accuracy</td>
<td>SAT</td>
<td>Queries</td>
<td>Accuracy</td>
</tr>
<tr>
<td>bzip2recover</td>
<td>100.0%</td>
<td>2101</td>
<td>5131</td>
<td>100.0%</td>
</tr>
<tr>
<td>jpeg</td>
<td>100.0%</td>
<td>50</td>
<td>198</td>
<td>100.0%</td>
</tr>
<tr>
<td>faad</td>
<td>99.23%</td>
<td>389</td>
<td>585</td>
<td>99.07%</td>
</tr>
<tr>
<td>foo2lava</td>
<td>87.1%</td>
<td>31</td>
<td>6252</td>
<td>87.1%</td>
</tr>
<tr>
<td>hdp</td>
<td>25.0%</td>
<td>464</td>
<td>2427</td>
<td>78.01%</td>
</tr>
<tr>
<td>jasper</td>
<td>0.05%</td>
<td>1987</td>
<td>5639</td>
<td>99.53%</td>
</tr>
<tr>
<td>libxml2</td>
<td>12.46%</td>
<td>1043</td>
<td>13520</td>
<td>50.98%</td>
</tr>
<tr>
<td>minigzip</td>
<td>10.73%</td>
<td>3961</td>
<td>4183</td>
<td>51.47%</td>
</tr>
<tr>
<td>muraster</td>
<td>99.97%</td>
<td>3235</td>
<td>4739</td>
<td>99.97%</td>
</tr>
<tr>
<td>pk2bm</td>
<td>98.91%</td>
<td>183</td>
<td>3672</td>
<td>99.45%</td>
</tr>
<tr>
<td>pnmhistmap_pgm</td>
<td>99.97%</td>
<td>3159</td>
<td>4681</td>
<td>99.99%</td>
</tr>
<tr>
<td>pnmhistmap_ppm</td>
<td>99.07%</td>
<td>107</td>
<td>8247</td>
<td>99.07%</td>
</tr>
<tr>
<td>readelf</td>
<td>61.93%</td>
<td>218</td>
<td>2046</td>
<td>86.47%</td>
</tr>
<tr>
<td>yices-smt2</td>
<td>2.5%</td>
<td>521</td>
<td>2135</td>
<td>78.33%</td>
</tr>
<tr>
<td>yodl</td>
<td>8.31%</td>
<td>313</td>
<td>5201</td>
<td>57.51%</td>
</tr>
</tbody>
</table>
## Parallel Solving

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>bzip2recover</td>
<td>51m3s</td>
</tr>
<tr>
<td>cjpeg</td>
<td>—</td>
</tr>
<tr>
<td>minigzip</td>
<td>29m42s</td>
</tr>
<tr>
<td>pk2bm</td>
<td>21m39s</td>
</tr>
<tr>
<td>pnmhistmap_ppm</td>
<td>28m52s</td>
</tr>
<tr>
<td>yodl</td>
<td>34m59s</td>
</tr>
</tbody>
</table>
Questions?