Creating Human Readable Path Constraints from Symbolic Execution

Tod Amon (ttamon@sandia.gov)
Tim Loffredo (tjloffr@sandia.gov)
Sandia National Laboratories
2/23/2020

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA0003525. SAND # 2020-2220 C Unlimited Unrestricted Release
Background

- **Path Constraints:**
  - An inherent component of symbolic execution;
  - When execution is conditional upon symbolic variables, multiple states arise, with different path constraints
  - Constraints stored in SMT solver

- **Example:**

```c
int abs(int x) {
    if (x < 0) {
        return -x;
    }
    return x;
}
```

- symbolic execution yields two states, with resulting path-constraints and return values

When \( x < 0 \)
- Result is \(-x\)

When \( x \geq 0 \)
- Result is \(x\)

When \(<\text{Bool} \ x[31:31] \neq 0>\)
- Result is \(<\text{BV32} \ 0xffffffff \times x>\)

When \(<\text{Bool} \ x[31:31] = 0>\)
- Result is \(<\text{BV32} \ x>\)
Readability

• Human-tool cooperation is currently the fastest approach for thoroughly analyzing programs

• Some common questions when symbolically debugging and reverse engineering binaries:
  ▪ What does this function do?
  ▪ Did I set up my symbolic variables correctly?
  ▪ How do I get here? or How did I get here?

• Simple questions should have simple answers
Contributions

• Our paper presents several examples that demonstrate the usefulness of path constraints and the need for them to be human readable

• We demonstrate the feasibility of transforming Boolean bit-vector constraints into the integer domain

• We present several novel ideas
  ▪ Including the use of logic synthesis tools to put constraints into specific forms.
  ▪ Including an alternative approach to type inferencing based simply on finding patterns in path-constraints.
Basics

- We are using “an gr” for symbolic execution
- We are using Z3
- We are using python
- Our artifacts are available here: http://github.com/TodAmon/Bar2020
Example #1:

- Help vulnerability researchers study functions.
  - Access to both source code and binary
  - Leverage SMT solvers to handle complex bit-vector issues
- Toy problem: When does this function return \( y - 2 \)?

```c
int sub1or2(int y)
{
    int x = y;
    x--;
    if (x > 5)
        x--;
    return x;
}
```

- Solution:
  - Two states are obtained from symbolic execution, one has the return value as:
    - **Claripy**: \(<BV32 \text{0xfffffffffe} + y\_text{intle:32} \_13\_32>\)
    - **Z3 sexpr**: \((\text{bvadd} \text{ #0xfffffffffe} \mid y\text{intle_32}_13\_32)\)
  - Print this state’s path-constraint to get the answer
Ugly Path Constraints

- **Claripy:**
  - \[
  \begin{aligned}
  &<\text{Bool} (0xffffffff + y\text{\_intle:32\_13\_32} - 0x5[31:31] \ ^ 0xffffffff + y\text{\_intle:32\_13\_32}[31:31] & (0xffffffff + y\text{\_intle:32\_13\_32}[31:31] ^ 0xffffffff + y\text{\_intle:32\_13\_32} - 0x5[31:31]) \ | (\text{if } 0xffffffff + y\text{\_intle:32\_13\_32} - 0x5 == 0x0 \text{ then } 1 \text{ else } 0)) == 0>
  \end{aligned}
  \]

- **Z3 string (simplified using ctx-solver-simplify):**
  - \[
  \begin{aligned}
  &\text{And}((\text{Extract}(31, 31, 4294967290 + y\text{\_intle:32}) == 1) == \\
  &\neg(\text{Or}(\text{Extract}(31, 31, 4294967290 + y\text{\_intle:32}) == 1, \text{Extract}(31, 31, 4294967295 + y\text{\_intle:32}) == 0)), \neg(y\text{\_intle:32} == 6))
  \end{aligned}
  \]

- **Z3 sexpr:**
  - \[
  \begin{aligned}
  (\text{let } ((a!1 (\text{bv_xor} ((\_extract\ 31\ 31) (\text{bv_add} \ #xxffffffff\ y))
  ((\_extract\ 31\ 31) (\text{bv_sub} (\text{bv_add} \ #xxffffffff\ y) \ #x00000005))))
  (a!3 (\text{ite} (= \ #x00000000 (\text{bv_sub} (\text{bv_add} \ #xxffffffff\ y) \ #x00000005)) \ #b1
  \ #b0)))) (\text{let } ((a!2 (\text{bv_xor} ((\_extract\ 31\ 31) (\text{bv_sub} (\text{bv_add} \ #xxffffffff\ y)
  \ #x00000005)) (\text{bv_and} ((\_extract\ 31\ 31) (\text{bv_add} \ #xxffffffff\ y)) a!1)))))
  (\text{and} (= \ #b0 (\text{bv_or} a!2 a!3))))
  \end{aligned}
  \]
• Path constraints are added when evaluating a conditional branch in the intermediate representation used by symbolic execution.

40053b: jle 400541 <sub1or2+0x1b>

vex for 0x40053b:
IRSB {
    t0:Ity_I1 t1:Ity_I64 t2:Ity_I64 t3:Ity_I64 t4:Ity_I64 t5:Ity_I64 t6:Ity_I64

    00 | ------ IMark(0x40053b, 2, 0) ------
    01 | t1 = GET:I64(cc_op)
    02 | t2 = GET:I64(cc_dep1)
    03 | t3 = GET:I64(cc_dep2)
    04 | t4 = GET:I64(cc_ndep)
    05 | t5 = amd64g_calculate_condition(0x0000000000000000e,
               t1,t2,t3,t4):Ity_I64
    06 | t0 = 64to1(t5)
    07 | if (t0) { PUT(rip) = 0x400541; Ijk_Boring }
NEXT: PUT(rip) = 0x0000000000040053d; Ijk_Boring
}
Why?

- Path constraints are added when evaluating a conditional branch in the intermediate representation used by symbolic execution.

  ```python
  ULong amd64g_calculate_condition (  
    ...
  return 1 & (inv ^ ((sf ^ of) | zf));
  ```

- Path constraints are simpler if vex is optimized
  - Our tools typically execute a single instruction at a time, for blocks the constraints are simpler
A Better Result

- Using type information and tools that transform patterns in bit-vector-domain to integer-domain

\[
\begin{align*}
\text{(let } \((a!1 \text{ (or } \text{(and } \text{(not } (\leq 1 \text{ } |y\text{-intle:32}|) \text{) } \text{(not } (\leq 6 \text{ } |y\text{-intle:32}|)\text{))} \\
\text{(and } (\geq |y\text{-intle:32}| 1) \text{ } \text{(<= } 6 \text{ } |y\text{-intle:32}|)\text{))} \\
\text{(>= |y\text{-intle:32}| 1)\text{)))})
\end{align*}
\]

\[
\begin{align*}
\text{(let } \((a!2 \text{ (or } \text{=} |y\text{-intle:32}| 6) \\
\text{(and } (\lt (+ \text{(- } 6 \text{ } |y\text{-intle:32}|) 0) \text{ a!1}) \\
\text{(and } (\geq (+ \text{(- } 6 \text{ } |y\text{-intle:32}|) 0) \text{ (not a!1))))\text{)))}
\end{align*}
\]

\[
\text{(not a!2))}
\]

No longer bvand, bvsub, bvadd, etc.

- Then use \texttt{ctx-solver-simplify} (or other approaches):

    \[
    \text{And(Not(|y\text{-intle:32}| == 6), 6 } \text{<= } |y\text{-intle:32}|)
    \]

- We are nearly there! (Z3 avoids strict inequalities)
A Better Result

• A lot of work to discover that when $y > 6$
  our function returns $y - 2$

```c
int sub1or2(int y) {
    int x = y;
    x--;
    if (x > 5) And(Not(y_intle:32 == 6), 6 <= y_intle:32)
        x--;
    return x;
}
```

• The translation into the integer-domain may not be precise,
  due to overflow or other bit-vector effects
  - E.g., if we switch $x--$ to $x++$ the result, that our function returns $y+2$ when $y > 4$
    is not precise in that there are some possible values of $y$ that do not return $y+2$.
  - See our code for methods to check equivalence of statements in the same domain,
    or potentially cross domain, in the presence of constraints
Example #2

• Tools to support network protocol extraction
  ▪ Identify paths from Source (e.g., read) to Sink (e.g., write)
  ▪ Configure Source as a symbolic byte array (network input)
  ▪ Sink deliver bytes to network
  ▪ How is what is written related to what is read?

• Add marshalling to previous example:

```c
read(0, inbuf, 64)
...
int *ri = (int*)&inbuf[0];
int x = *ri;
x--;
if(x > 5) {
    x--;
}
int *wi = (int*)&outbuf[0];
*wi = x;
write(1, outbuf, 4);
```

Configured as array of symbolic bytes:
[sym0, sym1, sym2, sym3, ...]
Example #2

- Users and tools have only the binary (no source)
- Path constraint when we decrement twice:

\[
\begin{align*}
\text{let } ( \text{a!1} = ( \text{extract 31 31} ( \text{bvadd \#fffffff}\text{ff} \text{a (concat sym3 sym2 sym1 sym0)}) ) \#b1)) \\
\text{a!2} = ( \text{extract 31 31} ( \text{bvadd \#fffffff}\text{ff} \text{f (concat sym3 sym2 sym1 sym0)}) ) \#b0)) \\
\text{a!3} = ( \text{extract 31 31} ( \text{bvadd \#fffffff}\text{ff} \text{f (concat sym3 sym2 sym1 sym0)}) ) \#b1)) \\
\text{let } ( (\text{a!4} \text{ or } ( = \text{a!1} \text{ or a!2} = ( \text{a!3 a!1}) )) \\
(\text{and} ( \text{sym0 \#x06} ) ( = \text{sym1 \#x00} ) ( = \text{sym2 \#x00} ) ( = \text{sym3 \#x00}) ) \) (not a!4)) )
\end{align*}
\]

- Path constraint suggests that our symbolic byte sequence contains a 32 bit integer in little endian
- Substitute each symbolic byte with an expression showing it as a piece in a hypothesized type
  - \text{sym0} \to ( \text{extract 31 24} | \text{sym[0-3]-?\_intle:32}| )
  - \text{sym1} \to ( \text{extract 23 16} | \text{sym[0-3]-?\_intle:32}| )
  - \text{sym2} \to ( \text{extract 15 8} | \text{sym[0-3]-?\_intle:32}| )
  - \text{sym3} \to ( \text{extract 7 0} | \text{sym[0-3]-?\_intle:32}| )

- Then apply domain conversion, and simplification to obtain:
  - And(6 <= sym[0-3]-?_intle:32, Not(sym[0-3]-?_intle:32 == 6))
Methodology

• Convert from bit-vector domain to integer domain
  ▪ Use examples to discover constraint patterns such as:
    - And-of-equality-on-extracts gets converted to actual value
    - If-then-else checks on a sign-bit gets converted to inequality
    - Concat-with-zero/s gets converted to multiplication
  ▪ Examples that fail suggest more patterns to understand
  ▪ Preliminary results testing on constraints from toy problems that are simplified using different strategies was very promising
Example #3

• Use logic synthesis tools with gate-libraries created for human readability for tailored situations.
  
  ▪ Example – path constraints when symbolic bytes are not equal to a string

```c
char inbuf[64];
num_bytes = read(0, inbuf, 64);
int authreq = (inbuf[0]==’A’ &&
inbuf[1]==’U’ &&
inbuf[2]==’T’ &&
inbuf[3]==’H’);
int good_password = (inbuf[4]==’T’ &&
inbuf[5]==’O’ &&
inbuf[6]==’D’ &&
inbuf[7]==0);
if (authreq && !good_password) {
  ... // send authentication rejection
}
```

If we combine the constraints for the four paths that lead to authentication rejection:

```c
Or(
  And(sym0==65, sym1==85, sym2==84, sym3==72,
      Not(sym4==84)),
  And(sym0==65, sym1==85, sym2==84, sym3==72,
      sym4==84, Not(sym5==79)),
  And(sym0==65, sym1==85, sym2==84, sym3==72,
      sym4==84, sym5==79, Not(sym6==68)),
  And(sym0==65, sym1==85, sym2==84, sym3==72,
      sym4==84, sym5==79, sym6==68, Not(sym7==0))
)
```

We can use SIS on a gate library biased to avoid “Or” gates to obtain:

```c
And(sym0==65, sym1==85, sym2==84, sym3==72,
    Not(And(sym4==84, sym5==79, sym6==68, sym7==0)))  \rightarrow  sym[0:3] == "AUTH" and
               sym[4:7] != "TOD\0"
```
Results

- Existing tools perform amazing analyses but are insufficient with regards to human readability:
  - \texttt{Z3 \_str\_} and \texttt{Z3.sexpr()} are useful at times but often misleading / dense
  - Claripy readability is an improvement over Z3 (and handles end-ness issues quite nicely) but the structure of the constraints are still unwieldy
  - Constraint simplification algorithms exist primarily for efficiency

- There exist promising techniques:
  - Pattern-matching when symbolic variables are annotated with type
  - Logic synthesis algorithms for simplifying and structuring

- Claim: readability of path-constraints is a largely unexplored and important aspect of automated analysis

- See our paper and code / artifacts for more details
A Difficult Task

• “Don’t attempt to understand anything after you’ve given it to an SMT solver”
  ▪ Indeed, the problem does appear challenging
  ▪ So to is the problem of understanding a binary (never meant for consumption by anything other than hardware)

• “Please don’t make me try and understand that”
  ▪ Humans need software to simplify things for their consumption

• “Use something other than symbolic execution”
  ▪ Yes! But we do need multiple approaches, and humans can more easily leverage the power of symbolic execution and SMT solvers
Future Work

• Formalize the notion of human-readability
  ▪ Score answers so we can choose good ones
• Quantitative Evaluation of our ideas
• Analysis on real binaries
• Work further upstream?
• Extend ideas to more data-types
• Extend ideas to other domains
  ▪ E.g., strings
Thank You