SAND2020-2220C

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 # XXXXXXXXXXX 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:

symbolic execution on the binary for this function with “x” setup as a symbolic 32 bit little endian integer yields two states, with resulting path-constraints and return values

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

```
When <Bool x_intle:32_13_32[31:31] != 0>
Result is <BV32 0xffffffff * x_intle:32_13_32>
```

```
When <Bool x_intle:32_13_32[31:31] == 0>
Result is <BV32 x_intle:32_13_32>
```

Readability

- Human-tool cooperation is currently the fastest approach for thoroughly analyzing programs
- Path constraints often contain useful information
  - For both tools *and* humans
  - Simple questions should have simple answers
  - The bit-vector domain is precise, but often obscures results.
- Some common questions when symbolically debugging and reverse engineering binaries:
  - How do I get here? or How did I get here?
  - Did I set up my symbolic variables correctly?
  - Is the path constraint satisfiable?
  - What does this function do?
  - What is this meaning of this complex bit-vector constraint?

Contributions

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

- Our tools 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 “angr” for symbolic execution
- We are using Z3
- We are using python
- Our artifacts are available here:
  <URL will be inserted later>

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` ?

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

```
400526: push rbp
400527: mov rbp,rsp
40052a: mov DWORD PTR [rbp-0x14],edi
40052d: mov eax,DWORD PTR [rbp-0x14]
400530: mov DWORD PTR [rbp-0x4],eax
400533: sub DWORD PTR [rbp-0x4],0x1
400537: cmp DWORD PTR [rbp-0x4],0x5
40053b: jle 400541 <sub1or2+0x1b>
40053d: sub DWORD PTR [rbp-0x4],0x1
400541: mov eax,DWORD PTR [rbp-0x4]
400544: pop rbp
400545: ret
```

- Solution:
  - Two states are obtained from symbolic execution, one has the return value as

    Claripy: `<BV32 0xfffffffe + y_intle:32_13_32>`
    Z3 sexpr: `(bvadd #xfffffffe |y_intle_32_13_32|)`
  - Print this state's path-constraint to get the answer

Ugly Path Constraints

- Claripy:
  - `[<Bool (0xffffffff + y_intle:32_13_32 - 0x5[31:31] ^ 0xffffffff + y_intle:32_13_32[31:31] & (0xffffffff + y_intle:32_13_32[31:31] ^ 0xffffffff + y_intle:32_13_32 - 0x5[31:31]) | (if 0xffffffff + y_intle:32_13_32 - 0x5 == 0x0 then 1 else 0)) == 0>]`
- Z3 string (simplified using ctx-solver-simplify):
  - `And((Extract(31, 31, 4294967290 + y_intle:32) == 1) == Not(Or(Extract(31, 31, 4294967290 + y_intle:32) == 1, Extract(31, 31, 4294967295 + y_intle:32) == 0)), Not(y_intle:32 == 6))`
- Z3 sexpr:
  - ```
    (let ((a!1 (bvxor ((_ extract 31 31) (bvadd #xffffffff y))
    ((_ extract 31 31) (bvsub (bvadd #xffffffff y) #x00000005))))
    (a!3 (ite (= #x00000000 (bvsub (bvadd #xffffffff y) #x00000005)) #b1
    #b0)))(let ((a!2 (bvxor ((_ extract 31 31) (bvsub (bvadd #xffffffff y)
    #x00000005))(bvand ((_ extract 31 31) (bvadd #xffffffff y)) a!1))))
    (and (= #b0 (bvor a!2 a!3)))))
    ```
    ```

Why?

- 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(0x000000000000000e,t1,t2,t3,t4):Ity_I64
        06 | t0 = 64to1(t5)
        07 | if (t0) { PUT(rip) = 0x400541; Ijk_Boring }
        NEXT: PUT(rip) = 0x000000000040053d; Ijk_Boring
     }
```

Why?

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

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

```
(let ((a!1 (bvxor ((_ extract 31 31) (bvadd #xffffffff y))
((_ extract 31 31) (bvsub (bvadd #xffffffff y) #x00000005))))
(a!3 (ite (= #x00000000 (bvsub (bvadd #xffffffff y) #x00000005))
 #b1 #b0) ))(let ((a!2 (bvxor ((_ extract 31 31) (bvsub (bvadd #xffffffff y)
#x00000005)) (bvand ((_ extract 31 31) (bvadd #xffffffff y)) a!1) )))
(and (= #b0 (bvor a!2 a!3)))))
```

- Path constraints <u>are</u> 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

```
(let ((a!1 (or (and (not (<= 1 |y_intle:32|)) (not (<= 6 |y_intle:32|)))
               (and (>= |y_intle:32| 1) (<= 6 |y_intle:32|))
               (>= |y_intle:32| 1))))
(let ((a!2 (or (= |y_intle:32| 6)
               (and (< (+ (- 6) |y_intle:32|) 0) a!1)
               (and (>= (+ (- 6) |y_intle:32|) 0) (not a!1)))))
  (not a!2)))
```

No longer bvand, bvsub, bvadd, etc.

- Then use ctx-solver-simplify (or other approaches):

```
And(Not(y_intle:32 == 6), 6 <= y_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`

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

```
And(Not(y_intle:32 == 6), 6 <= y_intle:32)
```

- Also, the translation into the integer-domain may not be precise, e.g., in some cases it could ignore 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 tools 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:

```
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);
return outbuf;
```

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:

```
(let ((a!1 (= ((_ extract 31 31)(bvadd #xfffffffa (concat sym3 sym2 sym1 sym0) )) #b1))
(a!2 (= ((_ extract 31 31)(bvadd #xffffffff (concat sym3 sym2 sym1 sym0) )) #b0))
(a!3 (= ((_ extract 31 31)(bvadd #xffffffff (concat sym3 sym2 sym1 sym0) )) #b1)))
(let ((a!4 (or (= a!1 (or a!2 (= a!3 a!1)))
(and (= sym0 #x06) (= sym1 #x00) (= sym2 #x00) (= sym3 #x00))))) (not a!4)))
```

- 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
  - ```
    sym0 -> ((_ extract 31 24) |sym[0-3]-?_intle:32|)
    sym1 -> ((_ extract 23 16) |sym[0-3]-?_intle:32|)
    sym2 -> ((_ extract 15 8) |sym[0-3]-?_intle:32|)
    sym3 -> ((_ extract 7 0) |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:
    - If-then-else checks on a sign-bit gets converted to inequality
    - And-of-equality-on-extracts gets converted to actual value
    - 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

Methodology

- 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

```
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:

```
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:

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

⇒

```
sym[0:3] == "AUTH" and
sym[4:7] != "TOD\0")
```

Results

- Existing tools perform amazing analyses
  but are insufficient with regards to human readability:
    - Z3 __str__ and 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 many 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