Creating Human Readable Path Constraints from Symbolic Execution

Tod Amon and Tim Loffredo
Sandia National Laboratories
Email: ttamon@sandia.gov, tjloffr@sandia.gov

***Abstract*—Advances in constraint solving have led to a prosperous time for static analysis. Powerful static analysis techniques like symbolic execution can now approach the scale of analyzing real commercial binaries - partly due to the efficient solving of symbolic constraints, which returns a satisfying variable assignment to those constraints or indicates that no such assignment is possible. While these advances have made automated machine analysis more scalable, the symbolic path constraints extracted from real commercial binaries and from toy problems are often unreadable for human analysts, who play an irreplaceable role in real-world binary analysis today. The work presented in this paper explores the problem of where path-constraints come from and how we might make symbolic path constraints easier for human analysts to digest and manipulate. This paper also presents a novel technique for automatically simplifying constraints based on conversion from the machine-centric *bit-vector* domain to the analyst-centric *mathematical integer* domain.**

In this paper we describe an impediment standing in the way of our building automated tools to assist humans performing binary analysis when using powerful tools like symbolic execution and SMT solvers: human readability of path constraints. Constraint solving is a core component of symbolic execution, and numerous advances in the field of binary analysis rely on these techniques[7]. Quite a lot of research effort is taking place to strengthen solvers, improve their efficiency, and extend the reach of tools upon which they are based. At Sandia National Laboratories, we are using symbolic execution and SMT solvers for a variety of missions in cyber security, such as vulnerability analysis, mitigating security threats, and strengthening application security.

This paper is informed by multiple efforts that aim to lean on skilled humans who interact with static analysis tools. Although we believe that the human analysts that work with our tools may be computer scientists that have taken courses in reverse engineering and will have some knowledge of assembly language, we do not necessarily expect them to understand all of the intricacies of symbolic execution, SMT solvers, and internal representations for different instruction set architectures. We want to enable our users to interact with tools that use these approaches without having to understand how to implement them.

Our key contributions: 1) The presentation of several detailed examples highlighting existing challenges in path constraint readability; 2) The development of a prototype based on pattern matching that illustrates the feasibility of converting bit-vector constraints into the integer domain; 3) A brief study showing the usefulness of logic synthesis algorithms for both simplifying and transforming formulas and the development of software that will allow others to experiment and build upon these ideas; 4) A demonstration showing the value of the information contained in the path constraints and a discussion of the benefits of making them more human readable.

Workshop on Binary Analysis Research (BAR) 2020
23 February 2020, San Diego, CA, USA
ISBN 1-891562-62-2
https://dx.doi.org/10.14722/bar.2020.23006
www.ndss-symposium.org

I. Examples

This paper is organized around three key examples which we present here and reference throughout the paper. The examples are small but are exemplars for the types of problems we have encountered and are attempting to solve with the tools we are building, in order to allow analysts to benefit from symbolic execution. All three examples contain path constraints that contain valuable information that a human analyst would like to know and would also likely believe should be available to them in human readable form.

Our first example: program analysis for a simple function, demonstrates that even for very simple problems, human readability of path constraints is problematic. Our second example: network protocol extraction, demonstrates the value of the path constraints that arise when symbolically executing code whose input consists of symbolic byte arrays. Our last example features a problem we have often encountered, the difficulty of understanding a symbolic path constraint for a string not being equal to a specific value. This example is used later when we discuss why logic synthesis algorithms can be of value.

All of our tools use angr [18] to perform symbolic execution of an X86 binary created from our source-code. We use Z3 [12] as the back-end constraint solver for Claripy [3] which wraps Z3. All of the examples, results, and code described in this paper is publicly available [4].

A. Program Analysis for a Simple Function Example

Consider the simple function shown in in Listing 1 that we analyze using symbolic execution. The corresponding assembly language is shown in Listing 2. We carefully use Claripy to set up "y" as a symbolic variable that is structured as a 32-bit value that represents a 32 bit little-endian integer, see Listing 3.

We then perform our symbolic execution, stepping angr a single instruction at a time. We can ask angr for the values of variables at various points in the execution. For example,

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

Listing 1: A simple function for analysis

```
0000000000400526 <sub1or2>:
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
```

Listing 2: Assembly language for Listing 1

```
y = claripy.BVS("y", 32)
symbolic_integer_le = claripy.Concat(
  claripy.Extract(7,0,y),
  claripy.Extract(15,8,y),
  claripy.Extract(23,16,y),
  claripy.Extract(31,24,y))
call_state = project.factory.call_state(
  0x400526, symbolic_integer_le, ...
```

Listing 3: Python code for symbolic "y" parameter

after stepping past the instruction at 0x400533 we can ask angr the value of "x" using the code in Listing 4 to obtain the symbolic results shown in Expression 1 to see that "x" is "y-1":

```
state.memory.load(
  state.solver.eval(state.regs.rbp) -4, 4,
  endness=archinfo.Endness.LE)
```

Listing 4: Code to examine "x" after first decrement

```
in Claripy: <BV32 0xffffffff + y>
as Z3 s-expr: (bvadd #xffffffff y)
as Z3 __str__:  4294967295 + y
```

Expression 1: Symbolic value of "x" using Listing 4 printed in three different formats.

The three expressions shown in Expression 1 are equivalent but were printed using three different methods. For all three, and throughout this paper, we first remove angr's name mangling of symbolic variables. Both Claripy and Z3 provide Python string methods (i.e. they implement "__str__") for printing symbolic values and constraints. Claripy is formatted nicely and does an especially good job of handling issues involving endness when symbolic values are not properly structured (e.g., it supports "Reverse"). Both methods use infix notation and at times hide some of the details in the expression, for example the difference between bit vector addition and integer addition. Both also truncate extremely lengthy outputs. In this paper we report most results using the Z3 formatter that creates s-expressions. An s-expression is a general purpose way to represent a nested list of data [15]. S-expressions use prefix notation and in Z3 often contain substitutions, as can be seen in Expression 2 in its declaration of "a!1", "a!2", and "a!3". For humans readability an s-expression may be the least human readable but in this paper the expressions are shorter and more precise. In cases where an equivalent expression is noticeably more readable when formatted using a string method, we have done so using Z3.

If we run until function exit (i.e., 0x400544) and examine the two symbolic states that result from symbolic execution, we can examine the EAX register to obtain results similar to those shown in Expression 1 (e.g., showing that the function returns "y-1" or "y-2"), and we can examine the path constraints to see under what conditions these values are returned. For example, the path constraint when the function returns "y-2" is shown as a Z3 s-expression in Expression 2.

Note that the path-constraint uses bit operations, an if-then-else, and several extracts of the sign bit of bit-vector arithmetic results. As such, this path constraint is very difficult to read. How long might it take someone to manually verify that "y > 6" is a simpler equivalent expression for this path constraint? Could the authors have even intentionally added a mistake to this representation of the condition, just to prove that no one would notice? Using two simplification tactics that are built into Z3 (i.e., "simplify", and "ctx-solver-simplify"), we can create many different expressions that are equivalent to Expression 2, but none of them are easily understood or noticeably smaller.

For readers interested in knowing where this path constraint comes from, we give a brief summary. The path constraint is "True" until angr symbolically executes the instruction at 0x40053b . The instruction is symbolically executed using angr's intermediate representation VEX [17] (which allows angr to support multiple architectures). The VEX statements shown in Listing 5 are difficult to fully understand without context, but show that the path constraint originates from the check on t0, which is derived from t5, which is obtained by calculating the less-than-or-equal conditions for the jump instruction. This in turn is based on an evaluation of the

```
(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)))))
```

Expression 2: Path Constraint when "y-2" is returned

```
------ 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(
              AMD64CondLE,
              t1,t2,t3,t4):Ity_I64
   06 | t0 = 64to1(t5)
   07 | if (t0) { PUT(rip) = 0x400541;
                  Ijk_Boring }
```

Listing 5: VEX code used when symbolically executing the code in Listing 2 at 0x40053b

```
ULong amd64g_calculate_condition (
     ULong/*AMD64Condcode*/ cond,
     ULong cc_op,
     ULong cc_dep1,
     ULong cc_dep2,
     ULong cc_ndep ) {
 ULong rflags;
 rflags = amd64g_calculate_rflags_all_WRK(
   cc_op, cc_dep1, cc_dep2, cc_ndep);
 ULong of,sf,zf,cf,pf;
 ULong inv = cond & 1;
...
 sf = rflags >> AMD64G_CC_SHIFT_S;
 of = rflags >> AMD64G_CC_SHIFT_O;
 zf = rflags >> AMD64G_CC_SHIFT_Z;
 return 1 & (inv ^ ((sf ^ of) | zf)); }
```

Listing 6: Calculation of t5 in Listing 5 is dependent upon three register flags

register flags arising from a comparison. The return value of the function shown in Listing 6 is the path constraint and a comparison of the code structure and the structure of the path constraint in Expression 2 shows a direct correspondence. The path-constraint s-expression contains "`(bvor a!2 a!3)`" and the code contains "`(sf ^ of) | zf)`" To summarize, "`a!3`" corresponds to the zero-flag, and "`a!2`" is the xor of the sign-flag and overflow flag.

B. Read-to-Write Analysis Example

One of our applications for symbolic execution involves analyzing execution paths from buffer reads to buffer writes to support network protocol extraction. Our approach to this problem isolates individual paths (or sets of paths) and attempts to describe how, and under what conditions, what is written is related to what is read. Our buffers are initialized using sequences of symbolic byte variables (e.g., "`sym0`", "`sym1`", "`sym2`" ,...). For this problem, we do not have source code (unless we create our own examples), and one of our goals is to assist an analyst in creating a succinct summary of the path-constraint, which will often depend on elements of the read buffer (e.g., the message format). To do this, we can at times infer types for some elements of the symbolic byte sequences by looking at known function signatures. We believe a valuable technique may be to inspect the resultant expressions (i.e., path constraints and values) looking for patterns that suggest types.

We present an example that casts I-A as this type of problem, i.e., in Listing 7 we introduce marshalling and buffers into Listing 1.

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

Listing 7: Similar to Listing 1 with marshalling

When we symbolically execute the binary for Listing 7 the symbolic expressions for the output buffer are based on the input buffer. For example, one of the symbolic states has "`outbuf[3]`" equal to "`((_ extract 31 24) (bvadd #xfffffffe (concat sym3 sym2 sym1 sym0)))`" and this symbolic value weakly hints that "`inbuf`" and "`outbuf`" may both be little-endian. The path constraint, shown in Expression 3 and simplified in Expression 4 is, as we saw with Expression 2, muddled by precise but unreadable bit vector operations. However, the presence of four symbolic bytes in a concat sequence that is applied in an arithmetic expression provides stronger evidence that we may have discovered an integer value, and in little-endian order because the sequence is descending (e.g., from "`sym3`" to "`sym0`").

```
(let ((a!1 (bvadd #xffffffff (concat
(concat (concat sym3 sym2) sym1) sym0))))
(let ((a!2 (bvand ((_ extract 31 31) a!1)
  (bvxor ((_ extract 31 31) a!1)
  ((_ extract 31 31)
  (bvsub a!1 #x00000005))))))
(let ((a!3 (bvor (bvxor ((_ extract 31 31)
(bvsub a!1 #x00000005)) a!2)
(ite (= #x00000000 (bvsub a!1 #x00000005))
#b1 #b0)))) (= #b0 a!3))))
```

Expression 3: Path Constraint for Listing 7

```
(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)))
(and (= a!1 (not (or a!1 a!2)))
(not (and (= sym0 #x06) (= sym1 #x00)
(= sym2 #x00) (= sym3 #x00)))))
```

Expression 4: Simplified Path Constraint for Listing 7

C. Authentication Example:

The example shown in Listing 8 also comes from network protocol extraction. Due to short-circuiting, there are four distinct paths in the symbolic execution of the binary that lead to an authentication rejection outcome. Our analysis combines these four paths and the resultant path-constraint is shown in Expression 5 formatted as a Z3 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
}
```

Listing 8: Authentication example

```
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)))
```

Expression 5: Rejection Path Constraint for Listing 8

II. Our Approach

Today's solvers are powerful highly optimized tools that are very good at answering questions such as "Is this expression satisfiable?" and, if so, "Provide a model assignment". Existing tools leverage these capabilities and put them to good use. However, these solvers are not well suited to having people read and interpret their results. Simplification algorithms exist, but the motivation for these algorithms for the most part is to improve performance. Our examples allow us to illustrate many points:

- When working in the bit-vector domain, negative numbers are ignored, making string representations of Z3 constraints very difficult to read (i.e., 4294967295 instead of -1 using Python Z3's __str__ method as shown in Expression 1). For this reason, we have reported most of our results using s-expressions.
- Even very simple statements in the integer domain, when expressed as bit vectors, become nearly impossible for humans to read. The presence of concat, extract, if-then-else, and complex tests on the sign bit, and other more complex logical constructs, result in statements that are not readable. This is true even for expressions like "`y > 6`" .

Of course binary analysis of real programs will only complicate matters further, and a very reasonable conclusion would simply be that one should never attempt to make sense of bit-vector expressions. As such, humans may interrogate the solver, but not examine what the solver knows. We reject this perspective for a number of reasons:

- For small academic "toy" problems, researchers themselves (i.e., as opposed to other analyst users) need the ability to read constraints.
- When working to build tools that put humans in the loop, we believe strongly that simple problems should have simple answers, even if complex problems do not.
- The expressions held by the solvers are sometimes the precise answers being sought.
- It seems strange for a community dedicated to making sense of a binary (i.e., the byproduct of human engineering that one could argue was never intended for consumption by anything other than hardware) to argue that one should not attempt to make sense of a solver's constraints.
- These are hard problems, and solutions will inevitably rely on multiple approaches and, in many cases, on finding agreement across multiple techniques. Thus, analysis of the constraints themselves is an alternative and potentially fruitful activity.

Thus we believe:

- Humans should have the ability to say to the computer "Spend some time (e.g., as much as 30 minutes) looking at this constraint and see if you can explain it more clearly"
- Working in the bit-vector domain is precise, but humans need to be able to tie the bit-vector statement back to other domains, even if the statement in the other domain is not precise.
- Humans also need to be able to ask solvers to compare domains for equality and/or to test domains for equality.

For some of the problems we are interested in, we have type information, for example when we knew that RBP was a frame pointer that could be used to access a 32-bit little-endian value in Listing 2. For our work in network protocol abstraction, we hope to provide a high level abstraction of the network protocol; and, for this, type information is essential but will have to be inferred because we do not have source code.

A. Rough Prototype for Domain Translation

We have built a rough prototype for domain translation using the Python Z3 library that essentially performs type-influenced pattern matching on Z3 internal expressions. The prototype was built as a proof of concept and consists of a few thousand lines of Python code. Variables are annotated with type information, e.g., using definitions in [2]. We anticipate at some point rewriting this library in Z3 C++ or writing the library as a fully independent tool that takes and produces SMT-LIB 2 expressions. Fundamentally, the rewriter has a simple

goal: replace bit-vector constructs with higher level constructs, e.g., given a bit-vector constraint with type information:

```
(and (= ((_ extract 31 24) |y_intle:32|) #xfe)
     (= ((_ extract 23 16) |y_intle:32|) #xff)
     (= ((_ extract 15 8) |y_intle:32|) #xff)
     (= ((_ extract 7 0) |y_intle:32|) #xff))
```

We can convert from the bit-vector domain to the integer domain (in this case "`y = -2`"). We do this by recognizing concats of extracts and searching for patterns where a conversion is sensical, even if it is not precise. For example, we have patterns that understand that concatenation with 0 is multiplying by two, that checking a sign bit and comparing it to an if-then-else on a condition yielding 0 or 1 is a statement about the condition and an inequality. Some of the patterns are quite simple, such as those that transform arithmetic or inequalities.

Our prototype [4] is at present very simple, but it has proven capable of handling many examples that arise in our problem domains. On occasion constraint simplification can make pattern recognition more difficult (e.g., when only some bits of a larger bit-vector are present in a path constraint). To some extent, we have invested time in understanding complex bit-vector expressions with the hope that at some point we can save others from having to do so. For now, we have demonstrated feasibility, but much more work is needed to handle additional data types and to develop a working tool.

B. Using Logic Synthesis Tools

In order to make path constraints more readable, we intend to take advantage of logic synthesis tools, which can both simplify and manipulate boolean expressions.

We are currently using the logic synthesis tool SIS [16]. Using algorithms that attempt to minimize the number of literals in a solution, as well as algorithms that map solutions to specific component libraries, we can have SIS automatically generate solutions that we believe have been optimized for human readability, or are ready for domain translation. It is interesting that the replacement tool for SIS, ABC [9], may not be as ideally suited because it has less focus on "Advanced combinational logic synthesis (extraction of shared logic, don't-care based optimization, Boolean decomposition, etc)" [1].

The need for "don't care" based optimizations is rooted in work done over twenty years ago by one of the authors on simplifying symbolic timing constraints for human readability [6]. In combinational logic, "don't cares" allow a user to specify input values that will never occur (in Karnaugh maps one marks the output with an "X").

We provide a simple example to demonstrate how logic synthesis can minimize boolean expressions containing integer inequalities. Given "`(y > 6 or y = 6) and not(y = 6)`" we can rewrite this expression as a simple combinational logic expression over two literals: "`(a+b)b'`" where "`a`" is label for "`y > 6`" and "`b`" is a label for "`y = 6`" . Furthermore, we can supplement our knowledge by having the solver prove unsatisfiability for the product "`ab`" which is thus a "don't care", i.e., it is not possible for "`y > 6`" and "`y = 6`" to both be true. We can then formulate the constraint simplification problem as choosing the best implementation for the truth table:

a	b	F
0	0	0
0	1	0
1	0	1
1	1	X

Which, in this case, is just "`a`" , i.e., our expression can be simplified to just "`y > 6`" . The formal underpinnings of this approach are described in [14]. We have recently implemented this algorithm in Python using Z3 and are evaluating its potential to simplify complex path constraints (in both the bit-vector and integer domains). The constraint solver is used to find don't cares and thus the approach can be computationally expensive if one examines all combinations of literals in an expression, but in practice most don't cares involve just a few literals.

Logic synthesis tools can also be used to manipulate boolean expressions into specific forms using gate libraries that bias solutions towards the use of specific logic gates. We intend to implement multiple mapping strategies but at present have tested a mapping designed to expose constraints that contain two components – a string being equal to some value, and a string being not equal to some value, like we see in Listing 7 and Expressions 3 and 4.

Using logic synthesis tools as a base allows us to construct algorithms with the property that humans can adjust how much time is spent performing simplifications and transformation looking for better answers. This is an important principle for our approach.

III. Preliminary Results

For example I-A, our tools are able to transform the bit-vector path constraint shown in Expression 2 into the integer domain path constraint shown in Expression 6. This expression can be simplified using Z3 tactics (e.g., repeat and ctx-solver-simplify) to "`And(6 <= y, Not(y == 6))`" . This result is Z3's preferred answer because it avoids strict inequalities, viewing them as a detriment to performance. This expression can be further simplified using either additional pattern recognition or simplification using logic synthesis to what a human would consider a readable answer "`y > 6`".

```
Not(Or(Or(And(-6 + y < 0, Or(y >= 1,
And(Not(1 <= y), Not(6 <= y)),
And(y >= 1, 6 <= y))), And(-6 + y >= 0,
Not(Or(y >= 1, And(Not(1 <= y),
Not(6 <= y)), And(y >= 1, 6 <= y))))),
y == 6))
```

Expression 6: Expression 2 translated to integer domain

We are able to obtain this result for many different variants of Expression 2 that we constructed by changing the order in which we repeatedly apply Z3 simplify and the ctx-solver-simplify tactic.

Combining this with our analysis of the contents of EAX, we achieve a totally correct statement: "when `y > 6` the output is `y - 2`" . For the other path, "when `y ≤ 6`" the output is `y - 1`" . This statement is correct in the integer domain, but it is not totally sound due to the possibility of underflow. Our perspective is that analysts will want this answer to get a general sense of the constraint, as well as a more detailed and simplified bit-vector answer if they are investigating bit-vector effects and potential vulnerabilities that might arise as a result.

For example I-B, a structural analysis of the path-constraint detects a concat sequence from "`sym3`" to "`sym0`" . When we see this expression used in an arithmetic expression, we record a strong hypothesis that it represents a 32-bit little-endian integer. We then replace individual symbolic bytes with an extract of a new 32-bit symbolic variable that we create, and see if our tools can find a possible domain translation. For this example we report:

```
And(6 <= sym_[0-3]-?_intle:32,
    Not(sym_[0-3]-?_intle:32 == 6))
```

One thing we can do with some success is use the solver to see if our conversion from one domain to another is sound. Given a variable relationship and a value relationship, we ask the solver to see if it can satisfy the variable relationship while breaking the value relationship. Using the code in Listing 9 we can have the solver check the satisfiability of "`variablesP and not valuesP`" and because this is solvable, we know there are values in the bit-vector domain where the result is not equivalent to the integer domain result. The solver can provide a model, i.e., "`i_y =4294967295, bv_y = #xffffffff`".

```
i_y = z3.Int('y')
bv_y = z3.BitVec('ybe32',32)
variablesP = (z3.BV2Int(bv_y) == i_y)
i_value = i_y + 1
bv_value = bv_y + z3.BitVecVal(1,32)
valuesP = (z3.BV2Int(bv_value) == i_value)
```

Listing 9: Python code for checking domain equivalence for an increment in bit vector and integer domains.

Domain equivalence may not in general hold, but we often have constraints upon our variables (e.g., "`y > 6 or y ≤ 6`"), and these constraints can be applied when checking for domain equivalence. Thus, for Listing 1 for any "`y`" input satisfying "`y > 6`" ,we know that "`y - 2`" is the result. Unfortunately, checking these equivalences is very computationally expensive. Thus, we provide the ability to check using values, e.g., the user supplies a value to see if it can be used to prove that the expressions in different domains are not precisely equivalent.

For example I-C, we created 8 labels for each equality expression in Expression 5 that states that a symbolic byte has a specific (ASCII) value. We then asked SIS to simplify the .eqn file shown in Listing 10

We ran a simple SIS script that did a "full_simplify", a "decomp -g" followed by a "map" operation on a gate library we constructed that biased the solution to not use any OR gates. SIS was able to find a solution involving only "`And`" and "`Not`" as shown in Listing 11. As a Z3 expression, the solution is:

```
#a is a label for 'sym0==65' (A),
#b is a label for 'sym1==85' (U),
#c is a label for 'sym2==84' (T),
#d is a label for 'sym3==72' (H),
#e is a label for 'sym4==84' (T),
#f is a label for 'sym5==79' (O),
#g is a label for 'sym6==68' (D),
#h is a label for 'sym7==0'
INORDER = a b c d e f g h;
OUTORDER = f1;
f1 = (a b c d e') + (a b c d e f') +
(a b c d e f g') + (a b c d e f g h');
```

Listing 10: Input file for SIS for Expression 5

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

This representation seems amenable to string conversion (e.g., "`sym[0:3]='AUTH' and sym[4:7]!='TOD\0'`"). We believe that in many circumstances when analyzing protocols the path constraints will contain expressions for message fields that are checked for string equality or inequality.

```
[152] = e f g h
[220] = [152]'
{f1} = [220] a b c d
```

Listing 11: Output from SIS script on Listing 10

IV. Previous Work

The SMT logics we are most interested in include closed quantifier-free formulas over the theory of fixed-size bit-vectors (QF-BV) and quantifier-free integer arithmetic (QF-NIA). We refer readers to [8], and specifically the description of logics [5].

Most of our work is based on Z3 [12]. Simplification routines are present in most SMT libraries, however, "the scalability of many static analysis techniques requires controlling the size of the generated formulas throughout the analysis" [13], and thus support for simplification is provided in order for the solver to be more performant. Some solvers mention human readability, e.g., KLEE [10] has a few options to make SMT-LIB 2 statements easier to read (e.g., '-smtlib-human-readable') but no existing tools provide support for deep analysis aimed at simplifying constraints for human readability. We have come across online posts that discuss the notion of converting between domains (e.g., from QF_ABV to AUFNIRA) but are aware of no tools that attempt to perform these types of conversions.

The selective symbolic execution tool S2E is supported by a bitfield-theory expression simplifier that performs limited types of conversion using both bottom-up and top-down analysis of the expression trees and "is an example of applying domain-specific logic to reduce constraint solving time" [11].

V. Summary

In this paper we presented several examples to demonstrate our belief that tools to create human readable path-constraints would be very valuable contributions to the static analysis community. We also presented evidence suggesting that such tools can be created. We can do better than having every individual project contemplate the implementation of "domain-specific" simplification strategies (while possible, we have found it difficult to find individual projects that have undertaken such an effort). Instead, we believe that many general purpose techniques can be developed for broad use, and that, for many problems in static binary analysis, it will be possible to create human readable path constraints that can be used for many different mission problems.

We believe that when simple solutions are available, it is imperative that tools present simple answers. We do not advocate that humans become experts at interpreting complex solver constraints – rather we want to make available algorithms that humans can choose to use to analyze constraints to expose simple facts when they have been accumulated (e.g., using tools like symbolic execution). When answers are complex, we anticipate humans not wanting closed-form solutions, but rather the ability to query a complex solution in order to facilitate additional analysis.

There are many future research directions suggested by our initial investigations into creating human readable path-constraints. A formal definition of "human readability" and the development of metrics would allow us to score path-constraint expressions and penalize ones that are deemed less readable. The score could be based on the elements of the expression (e.g., a penalty for use of 'bvxor'), the depth and complexity of the expression, etc. Instead of working with the patterns that arise from symbolic execution of the constraints, we could work further upstream and alter the constraints that are created within angr when symbolically executing a VEX instruction, e.g., as shown in the origin of the path constraint in section I-A.

It may be that by injecting constraints that are easier for humans to read for a given instruction, we can create more readable path constraints and trade off some efficiency to achieve this goal. It is also quite likely that by examining the constraints that are added we can enrich our understanding of the patterns we need to recognize. It is also possible that focusing on the readability of the bit-vector path-constraints would be another viable approach for most of our tools and allow us to stay in the bit-vector domain.

References

[1] Abc: A system for sequential synthesis and verification. what is not in the current release? https://github.com/angr/claripyhttp://vlsicad.eecs.umich.edu/BK/Slots/cache/www-cad.eecs.berkeley.edu/~alanmi/abc/.

[2] Bitstring is a pure python module that makes the creation, manipulation, and analysis of binary data as simple as possible. https://bitstring.readthedocs.io/en/latest/packing.html.

[3] Claripy is an abstracted constraint-solving wrapper. https://github.com/angr/claripy.

[4] Repository for bar2020 creating human readable path constraints from symbolic execution. https://github.com/TodAmon/BAR2020.

[5] Smt logics. http://smtlib.cs.uiowa.edu/logics.shtml.

[6] T. Amon, G. Borriello, and Jiwen Liu. Making complex timing relationships readable: Presburger formula simplification using don't cares. In *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*, pages 586–590, June 1998.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[9] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 24–40, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, February 2012.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Isil Dillig, Thomas Dillig, and Alex Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, pages 236–252, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[14] Hoon Hong. Simple solution formula construction in cylindrical algebraic decomposition based quantifier elimination. In *Papers from the International Symposium on Symbolic and Algebraic Computation*, ISSAC '92, pages 177–188, New York, NY, USA, 1992. ACM.

[15] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[16] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[17] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[18] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 138–157. IEEE, 2016.