SAND2019-15210C

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 *bitvector* domain to the analyst-centric *mathematical integer* domain.**

I. Introduction

In this paper we describe an impedement 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[2]. 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. Additionally, we have found that there are a number of use-cases where an analyst might need to examine symbolic path-constraints and understand them, rather than apply different analysis techniques that can naturally produce more readable output. We also discuss a use case based on analyzing byte arrays to find patterns in symbolic byte sequences.

This paper is organized around key examples, which we present in the next section. We then follow with a description of our approach, a discussion of related work, and then a summary.

II. Examples

We consider several examples derived from tools we are building. All of our tools use angr [11] to perform symbolic execution of an X86 binary created from our source-code. We use Z3 [7] as our constraint solver.

A. Program Analysis for a Simple Function Example

Consider this simple function being subjected to analysis using symbolic execution:

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

```
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
```

We carefully set up y as a symbolic integer that is structured as a 32-bit little-endian bit vector:

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

We then perform our symbolic execution, stepping angr a single instruction at a time. We can ask angr for the values of

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

variables at various points in the execution. For example, after executing the code at 0x400533 we can ask angr the value of ‘x’ using:

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

to see that ‘x’ is ‘y-1’:

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

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 above (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, ‘y-2’ is returned when:

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

We omit the string representations in Claripy and Z3 for sake of brevity and show only the sexpr. 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 (simplify, and ctx-solver-simplify), we can create many different expressions that are equivalent to the expression above, but none of them are easily understood or noticably smaller.

For readers interested in knowing where this path constraint comes from, we give a brief summary. There are no path constraints until angr symbolically executes the instruction at 0x40053b. This is accomplished by executing the VEX statements corresponding to the instruction (which are difficult to understand without context). Those statements are:

```
------ IMark(0x40053b, 2, 0)
t1 = GET:I64(offset=144)
t2 = GET:I64(offset=152)
t3 = GET:I64(offset=160)
t4 = GET:I64(offset=168)
t5 = amd64g_calculate_condition(AMD64CondLE,
       t1,t2,t3,t4):Ity_I64
t0 = 64to1(t5)
if (t0) {PUT(offset=184)=0x400541; Ijk_Boring}
```

From this VEX code we can see 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 register flags arising from a comparison. Although we omit an in-depth explanation, we can see the core elements of our path constraint in the last line of the function:

```
ULong amd64g_calculate_condition (
     ULong/*AMD64Condcode*/ cond,
     ULong cc_op,
     ULong cc_dep1,
     ULong cc_dep2,
     ULong cc_ndep ) {
  ULong 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));
```

Our path-constraint sexpr contains “(bvor a!2 a!3)” and 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 constraints (for paths and values) looking for patterns that suggest types. We present an example that casts II-A as this type of problem:

```
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);
```

When we examine the output buffer, we obtain a symbolic constraint, e.g., for ‘outbuf[3]’:

```
(= write_byte_3 ((_ extract 31 24)
(bvadd #xfffffffe
(concat sym3 sym2 sym1 sym0))))
```

the path constraint as an sexpr is:

```
(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))))
```

As expected, the symbolic values and constraints are muddled by precise but unreadable bitvector operations. However, the presence of four symbolic bytes in a concat sequence that is applied in an arithmetic expression could be viewed as a strong indicator that we we have discovered an integer value, and in little-endian order because the sequence is descending (e.g., from ‘*sym3*’ to ‘*sym*0’).

C. Authentication Example:

In this example we see only a portion of the code:

```
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
}
```

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 paths and the path-constraint becomes (as a Z3 string):

```
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)))
```

III. 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 II-A). For this reason, we have reported most of our results using sexprs.
- 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 II-A. 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., as defined in https://bitstring.readthedocs.io/en/latest/packing.html. 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, etc. Our prototype is by no means complete, but it has proven capable of handling many examples that arise in our problem domains. To some extent, we have invested time in understanding complex bit-vector expressions with the hope that this can save others from having to do so.

We report results from our tool for some of the problems described earlier.

B. Results for II-A and II-B

Using our tool that performs pattern matching (e.g., it understands that comparing the sign-bit for 0 or 1 in the bit-vector domain is expressed as an inequality in the integer domain), we obtain an integer domain expression corresponding to the bit-vector domain expression shown in II-A:

```
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))
```

Which can be simplified using Z3 tactics (e.g., repeat and ctx-solver-simplify), yielding:

```
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. We are able to obtain this result for example II-A and for many variants of the bit-vector path constraint we constructed by changing the order in which we repeatedly apply the Z3 simplify and ctx-solver-simplify tactics. 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 $y \leq 6$ the output is y-1” 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 II-B, a structural analysis of the path-constraint detects a concat sequence from ‘*sym3*’ to ‘*sym*0’; 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 other analysis 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. Consider an example:

```
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)
```

The solver will report that “variablesP and not valuesP” is solvable, with one model being i_y =4294967295, bv_y = #xffffffff.

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 our path-constraint example in II-A, 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.

C. Constraint simplification using SIS

In order to simplify complex boolean expressions, we intend to take advantage of logic synthesis tools. Here we show how a logic synthesis tool designed to minimize numbers of literals and map solutions to preferred gate libraries can be used to construct solutions that are more human readable. This approach may also prove useful when looking for specific patterns when attempting to map from one domain to another.

In the next section we describe a technique that utilizes logic synthesis tools and don’t cares to simplify expressions like the one in III-B to get better results than those obtained using ctx-solver-simplify.

We used the logic synthesis tool SIS [10]. 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. It is interesting that the replacement tool for SIS, ABC [4], may not be as ideally suited, e.g., it has less focus on “Advanced combinational logic synthesis (extraction of shared logic, don’t-care based optimization, Boolean decomposition, etc)” per http://vlsicad.eecs.umich.edu/BK/Slots/cache/www-cad.eecs.berkeley.edu/~alanmi/abc/. In order to have SIS operate on example II-C we simply created 8 labels for the assertions that a symbolic byte has a specific (ASCII) value. We then asked SIS to simplify the .eqn file:

```
#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’);
```

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:

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

As a Z3 expression, the solution is:

```
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”), and we hope that in many circumstances when analyzing protocols the domain will result in constraints where the symbolic byte variables that represent the protocol message are checked for equality or inequality to a string (possibly null terminated).

IV. Previous Work

The SMT logics we are most interested in include closed quantifier-free formulas over the theory of fixed-size bitvectors (QF-BV) and quantifier-free integer arithmetic (QF-NIA). We refer readers to [3], and specifically the description of logics (http://smtlib.cs.uiowa.edu/logics.shtml). Most of our work is based on Z3 [7]. 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” [8], and thus support for simplification is provided in order for the solver to be more performant. Some solvers mention human readability, e.g., KLEE [5] 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” [6].

Twenty years ago when working with symbolic timing constraints, one of the authors discovered a technique well suited to our stated goal of asking the computer to spend significant resources trying to simplify a constraint for human readability[1]. The technique can be applied to boolean expressions and is best understood through a simple example. Consider the simplified Z3 expression from III-B : ‘$y \geq 6$ and not $y = 6$’. We can rewrite this expression as a Boolean expression: $(a + b)\ b'$ where a is label for ‘$y > 6$’ and b is a label for ‘$y = 6$’. Furthermore, we can supplement our knowledge by noticing that the product ab is a “don’t care”, i.e., it is not possible for ‘$y > 6$’ and ‘$y = 6$’ to both be true. As such, we are free to include this term in our solution if we so choose. 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 [9]. At present, we are in the process of re-implementing this code for use in simplifying complex Z3 boolean expressions. This technique is an example of a general purpose approach to simplifying complex solver constraints that involve boolean operations. It can be computationally expensive (we use the solver to discover “don’t cares”) but is an example of the type of analysis we believe should be generally available to individuals trying create human readable path constraints.

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 (such as those we described in sections III-A and IV) 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 example II-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] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.