SMT Solving in security applications

Dr. Morton Swimmer
Forward-Looking Threat Research
Trend Micro, Inc
2018-01-25 for MUC:SEC Meetup

SAT Solvers

- The SAT problem
  - “Is there a solution to a boolean expression making it true”
  - $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (x_3 \vee x_4 \vee x_1) \wedge (x_4 \vee \neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4 \vee \neg x_1) \wedge (\neg x_4 \vee x_1 \vee \neg x_2)$
- If you can express a problem as a boolean expression, you can solve it this way

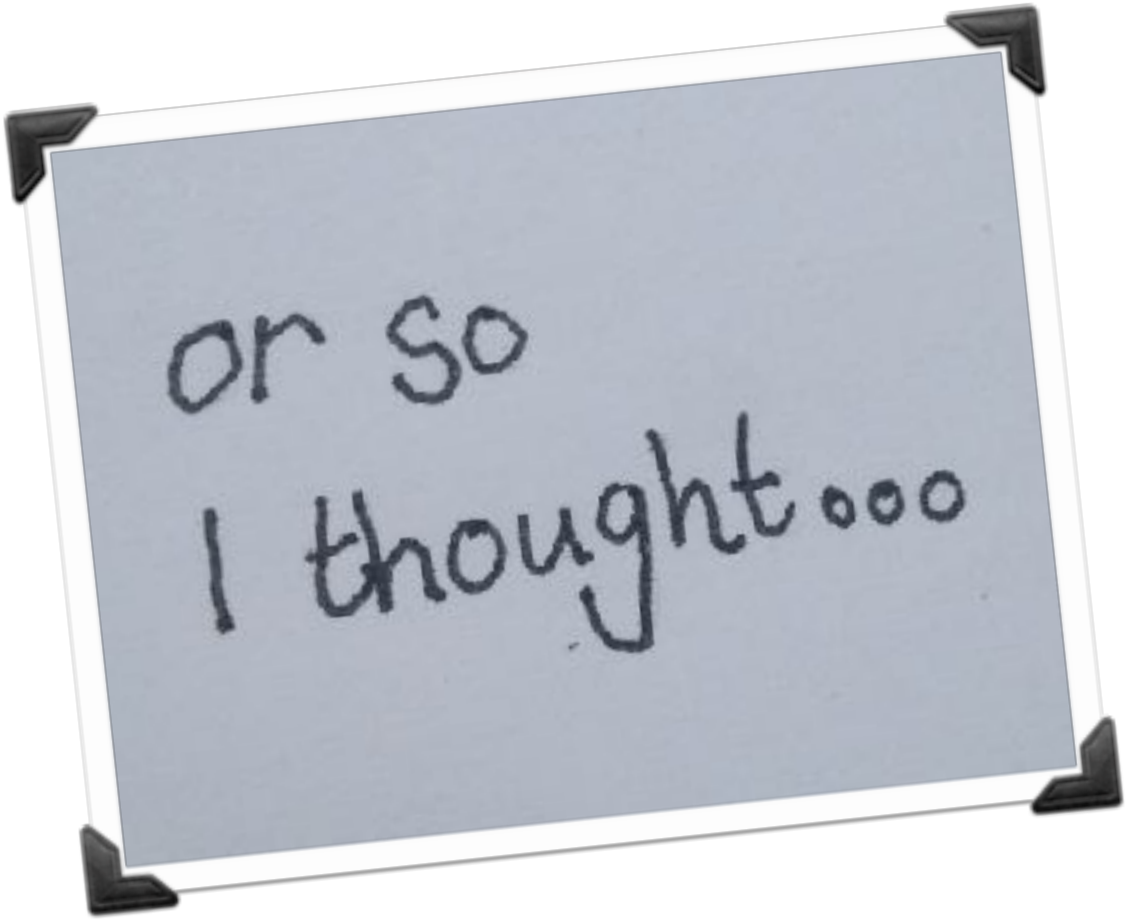
Security applications

- Security settings consistent with requirements
  - Configuring IPTables, ACL rules, and SDN
  - AWS settings insanity
- Reversing:
  - Scoping register values at key system calls
- etc

Sounds great! Why aren’t we using it?

The SAT Problem

- Is a combinatorial problem
  - and is NP-Complete
- So, there is no useful algorithm for meaningful-sized problems
- Bummer



While I wasn’t paying attention

- Tremendous progress was made using heuristics
  - CVC3, CVC4, STP, Alt-Ergo, Yices, **Z3**, …
- Solutions with millions of variables now possible
- Bring in the actual science of security!

Um. What was that about ‘SMT Solvers’?

- SMT = Satisfiability Modulo Theories
- A generalisation of SAT
- This way, we can use maths formulas
  - $2x = 7$ AND $x < 10$
- Basically, this is first-order logic
- In the background SMT translated to SAT

Z3 <3 <3 <3

- Dr. Nicolaj Bjørner, Leonardo de Moura, et al
  - Microsoft Research
- Comprehensive ‘theorems’
- Many language bindings
- https://github.com/Z3Prover/z3
  - MIT License

How to engineer an SMT problem?

Problem statement

- How to allocate n Heralds (session chairs) to m time slots
- Let’s build this systematically (if not really completely) as a toy example

Preamble: we start with boilerplate

```
from z3 import *

s = Solver()
n_shifts = 4
n_heralds = 2
n_shift_max = 2

herald_shifts = []
for i in range(n_heralds):
    herald_shifts.append(IntVector('herald_'+str(i)+'_shifts', n_shifts))

for i_herald in range(n_heralds):
    for i_shift in range(n_shifts):
        s.add(Or(herald_shifts[i_herald][i_shift] == 0, herald_shifts[i_herald][i_shift] == 1))
```

Define the Herald matrix

Make sure that a Herald can have a max of one shift at a time

for every shift,
count the number of
heralds assigned to it and
ensure it's one

```
for i_shift in range(n_shifts):
    s.add(Sum([herald_shifts[i_herald][i_shift] for i_herald in range(n_heralds)]) == 1)

for i_herald in range(n_heralds):
    s.add(Sum(herald_shifts[i_herald]) == 2)
```

for every herald, we
need to make sure s/he has
exactly 2 shifts

```
[Or(herald_0_shifts__0 == 0, herald_0_shifts__0 == 1),
 Or(herald_0_shifts__1 == 0, herald_0_shifts__1 == 1),
 Or(herald_0_shifts__2 == 0, herald_0_shifts__2 == 1),
 Or(herald_0_shifts__3 == 0, herald_0_shifts__3 == 1),
 Or(herald_1_shifts__0 == 0, herald_1_shifts__0 == 1),
 Or(herald_1_shifts__1 == 0, herald_1_shifts__1 == 1),
 Or(herald_1_shifts__2 == 0, herald_1_shifts__2 == 1),
 Or(herald_1_shifts__3 == 0, herald_1_shifts__3 == 1),
 herald_0_shifts__0 + herald_1_shifts__0 == 1,
 herald_0_shifts__1 + herald_1_shifts__1 == 1,
 herald_0_shifts__2 + herald_1_shifts__2 == 1,
 herald_0_shifts__3 + herald_1_shifts__3 == 1,
 herald_0_shifts__0 + herald_0_shifts__1 + herald_0_shifts__2 + herald_0_shifts__3 == 2,
 herald_1_shifts__0 + herald_1_shifts__1 + herald_1_shifts__2 + herald_1_shifts__3 == 2]
```

For this simple example, we can still print out the entire expression

Synthesise some Results

```
([[herald_1_shifts__3 = 1,
 herald_1_shifts__2 = 0,
 herald_1_shifts__1 = 0,
 herald_0_shifts__3 = 0,
 herald_0_shifts__2 = 1,
 herald_0_shifts__1 = 1,
 herald_0_shifts__0 = 0,
 herald_1_shifts__0 = 1],
[herald_0_shifts__0 = 0,
 herald_1_shifts__3 = 0,
 herald_0_shifts__2 = 1,
 herald_1_shifts__2 = 0,
 herald_1_shifts__1 = 1,
 herald_0_shifts__1 = 0,
 herald_0_shifts__3 = 1,
 herald_1_shifts__0 = 1], ...
```

And then I discovered Bool (doh!)

```
herald_shift = []
for i in range(n_heralds):
    herald_shift.append(BoolVector('herald_'+str(i)+'_shift', n_shifts))

for i_shift in range(n_shifts):
   s.add(AtMost(*([herald_shift[i_herald][i_shift] for i_herald in range(n_heralds)]), 1))
    s.add(AtLeast(*([herald_shift[i_herald][i_shift] for i_herald in range(n_heralds)]), 1))

for i_herald in range(n_heralds):
    s.add(AtMost(*herald_shift[i_herald], 2))
    s.add(AtLeast(*herald_shift[i_herald], 2))
```

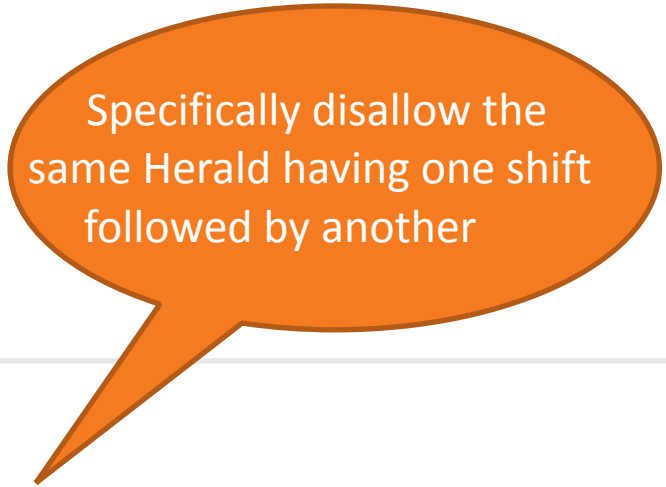
for every shift, count the number of heralds assigned to it and ensure it's one

for every herald, we need to make sure s/he has exactly 2 shifts

[AtMost((herald_0_shift__0, herald_1_shift__0), 1),
at-least(herald_0_shift__0, herald_1_shift__0),
AtMost((herald_0_shift__1, herald_1_shift__1), 1),
at-least(herald_0_shift__1, herald_1_shift__1),
AtMost((herald_0_shift__2, herald_1_shift__2), 1),
at-least(herald_0_shift__2, herald_1_shift__2),
AtMost((herald_0_shift__3, herald_1_shift__3), 1),
at-least(herald_0_shift__3, herald_1_shift__3),
AtMost((herald_0_shift__0, herald_0_shift__1, herald_0_shift__2, herald_0_shift__3), 2),
at-least(herald_0_shift__0, herald_0_shift__1, herald_0_shift__2, herald_0_shift__3),
AtMost((herald_1_shift__0, herald_1_shift__1, herald_1_shift__2, herald_1_shift__3), 2),
at-least(herald_1_shift__0, herald_1_shift__1, herald_1_shift__2, herald_1_shift__3)]

The expression is still manageable for this toy example

Requirement: no consecutive shifts




```
for i_herald in range(n_heralds):
    for i_shift in range(n_shifts-1):
        s.add(Not(And(herald_shift[i_herald][i_shift],herald_shift[i_herald][i_shift+1])))
```

Requirement: shifts on different days

```
n_shifts = 16
n_heralds = 8
n_days = 4
```

```
for i_herald in range(n_heralds):
    for i_day in range(n_days):
        s.add(AtMost(
            *[herald_shift[i_herald][int(n_shifts/n_days)*i_day + i]
                for i in range(int(n_shifts/n_days)) ],
            1))
```

Naively, we assume that each day has the same number of shifts and is exactly divisible

Requirement: A Herald gets certain shifts

We explicitly assign Herald 0 the first and last shift. If this violated any other requirements, we should have had to code up an exception

```
s.add(herald_shift[0][0] == True)
s.add(herald_shift[0][n_shifts-1] == True)
```

Requirement: Herald preferences

```
morning_shifts = [0, 1, 4, 5, 8, 9, 12, 13]
afternoon_shifts = [2, 3, 6, 7, 10, 11, 14, 15]

s.add(Not(Or([herald_shift[1][i_shift] == True for i_shift in morning_shifts])))
s.add(Not(Or([herald_shift[2][i_shift] == True for i_shift in morning_shifts])))

s.add(Not(Or([herald_shift[3][i_shift] == True for i_shift in afternoon_shifts])))
s.add(Not(Or([herald_shift[4][i_shift] == True for i_shift in afternoon_shifts])))

s.add(Not(Or([herald_shift[5][i_shift] == True for i_shift in range(int(n_shifts/2), n_shifts)])))
```

heralds 1 and 2 want to avoid the morning shifts

herald 5 can only make it on the first two days

heralds 3 and 4 want to avoid the afternoon shifts

Example solution:

```
day->       0   0   0   0   1   1   1   1   2   2   2   2   3   3   3   3
       am  am  pm  pm  am  am  pm  pm  am  am  pm  pm  am  am  pm  pm
shift->     0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
herald   --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
0      T                                                           T
1                                   T                   T
2                   T                                           T
3           T                                   T
4                       T                                   T
5               T               T
6                                       T                   T
7                                   T               T
```

I finally wrote a pretty printer :-)

Times

- For all the requirements execution ranged from 0.01109 sec to 0.00050 sec
  - The solution iterator tends to get FASTER because you impose MORE constraints every cycle
- In any case, this is plenty fast, but adding more slots and heralds has penalties
- https://github.com/mswimmer/smt-scheduling-toy

Network security

Simple questions, hard to answer

- Which packets can reach B from A?
- Are subnets A isolated from B?
- Can protocol p be received on A?
- Can we synthesise a network to our policy set?

Inspiration:

Automated Analysis and Debugging of Network Connectivity Policies

Karthick Jayaraman
Microsoft Azure
karjay@microsoft.com

Nikolaj Bjørner
Microsoft Research
nbjorner@microsoft.com

Geoff Outhred
Microsoft Azure
geoffo@microsoft.com

Charlie Kaufman
charliekaufman@outlook.com

ABSTRACT

Network connectivity policies are crucial for assuring the security and availability of large-scale datacenter. Managing these policies is fraught with complexity and operator errors. The difficulties are exacerbated when deploying large scale offerings of public cloud services where multiple tenants are hosted within customized isolation boundaries. In these large-scale settings it is impractical to depend on hu-

routers, and they enforce an access-control list (ACL) to enforce restrictions on traffic coming from the Internet. We will refer to it as the Edge ACL in this paper. Figure 1 provides a canonical example of an Edge ACL and typical maintenance operations done on it. The ACL in this example is authored in the Cisco IOS language. It is basically a set of rules that filter IP packets. They inspect header information of the packets and the rules determine whether

Basic idea

- We have a Contract C and a Policy P
- We are interested in
  - Is C preserved by P?
  - Or not
  - Is a subset of C preserved by P
- Network wide

My little toy: Check IPTables

- E.g. on INPUT chain

ACCEPT icmp -- 10.0.0.0/8 anywhere icmp echo-request

- can be expressed as

r_1: 10.0.0.0 <= srcIP <= 10.255.255.255 and
0.0.0.0 <= dstIP <= 255.255.255.255 and
protocol == 1 and dst_icmp_port == 8
(Status: Allow)

Combining rules

- IPTables is a First Applicable rule set
- We chain each rule r_i together logically
- Terminate with False (deny unmatched)

$$
\begin{aligned}
P(\vec{x}) &= P_1(\vec{x}) \\
P_i(\vec{x}) &= r_i(\vec{x}) \vee P_{i+1}(\vec{x}) && \textbf{if } r_i.status = Allow \\
P_i(\vec{x}) &= \neg r_i(\vec{x}) \wedge P_{i+1}(\vec{x}) && \textbf{if } r_i.status = Deny \\
P_n(\vec{x}) &= false
\end{aligned}
$$

Value

- Once all IPTables in a cluster can be so verified, bad configurations can be more easily found
- Network ACL rules can also be incorporated
- Mainly challenging because of variety of formats to parse

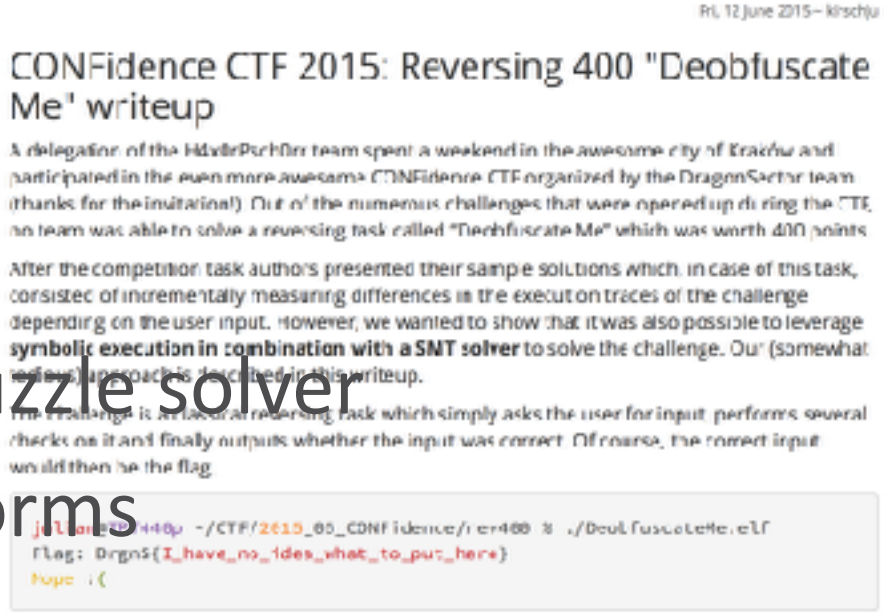
But we can do more

- As the scheduling example showed: we can synthesise the IPTables and ACL rules
- Introspecting a codebase for a cluster, we can generate the Contract and use the solver to generate the Policies
- This would be possible today

What else?

Other applications

- Win CtFs!
  - SMT, the ultimate puzzle solver
- Used in reversing platforms
  - Klee
  - Angr
- Program synthesis from specification



Caveats

- Don’t forget to look for special-purpose algorithms
- Real World™ → SMT expression
  - A little mathematical logics background is a good idea
- Not always an answer
  - SAT, UNSAT and I-Don’t-Know

Pointers

- More on Z3: https://rise4fun.com/Z3
- Bjørner, et al. Automated Analysis and Debugging of Network Connectivity Policies. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/secguru.pdf
- Dennis Yurichev: https://yurichev.com/blog/index.html
- Cornelius Diekmann. Verified iptables Firewall Analysis and Verification

Solve your security, don’t guess it

MORTON.SWIMMER@TRENDMICRO.DE

HTTPS://GITHUB.COM/MSWIMMER/SMT-SCHEDULING-TOY