Hacking Using SAT and SMT Solvers

Or how I managed to spend 8 years of my life optimising an algorithm that I don't understand

Mate Soos
soos.mate@gmail.com

01-09-2018
ABOUT ME

- Engineer
- Hacker
- Threat modeling
- Pentesting
- Security architecture
- Broke a few things (Oyster cards, cars, SIM cards, GPRS, etc)
- Currently privacy engineer at Zalando (Thanks for sponsoring me!)
- My hobby is SAT solving… eats about 20-30h/week
- Hobby project: CryptoMiniSat
- Did some research here and there on SAT, e.g. at NUS
TABLE OF CONTENTS

SAT Solvers
SMT Solvers
Concolic Execution
Use Cases
INPUT TO SAT SOLVERS

- A bunch of variables that can be true/false: x, y, z, etc.
- A set of requirements in a simple form
- Allows you to express all NP-complete problems
- We can model SHA-1
  github.com/vegard/sha1-sat
- We can model a LOT of things
  Examples: MiniSat, lingeling, MapleSAT CryptoMiniSat

\[
\begin{align*}
  x & \lor \neg y \lor \neg z \\
  -x & \lor y \\
  -x & \lor z \\

  \text{This means } x = y \text{ AND } z
\end{align*}
\]

\[
\begin{align*}
  x & \lor y \lor \neg z \\
  x & \lor \neg y \lor z \\
  -x & \lor y \lor z \\
  -x & \lor \neg y \lor \neg z \\

  \text{This means } x = y \text{ XOR } z
\end{align*}
\]
SAT SOLVERS ARE GLORIFIED SEARCH ENGINES

Start

Guess until conflict

Backtrack

Solution Found

First conflict
BUT WHY SO FAST?

BECAUSE IT’S GOT SCIENCE

• 1UIP conflict generation
• VSIDS and LRB variable picking
• Backjumping
• Timestamping
• Binary implication graph minimisation (transitive reduction)
• InTree probing
• ...

BECAUSE IT’S OPTIMIZED

• Lazy data structures
• Hand-rolled memory manager
• Bitstuffed datastructs
• Prefetching
• Variable renumbering
• Optimised on thousands of instances: just last year I burned 300k CPU hours on it
• ...

In 2016 I did a presentation on all the implemented optimisations… it’s 16 pages of acronyms :S
INPUT TO SMT SOLVERS

- Bunch of variables that can be any type. e.g. 32bit signed int, double, float etc
- Set of requirements that can express complicated stuff like addition, division, multiplication, etc.
- Static Single Assignment (SSA) form
- High level, a lot easier to deal with
- Translates these requirements to SAT (in a smart way).

Examples: Z3, Boolector, STP, CVC4

; Modeling sequential code with bitvectors
;; Correct swap with no temp var
; int x, y;
; x = x + y;
; y = x - y;
; x = x - y;
(set-logic QF_BV)
(declare-const x_0 (_ BitVec 32))
(declare-const x_1 (_ BitVec 32))
(declare-const x_2 (_ BitVec 32))
(declare-const y_0 (_ BitVec 32))
(declare-const y_1 (_ BitVec 32))
(assert (= x_1 (bvadd x_0 y_0)))
(assert (= y_1 (bvsub x_1 y_0)))
(assert (= x_2 (bvsub x_1 y_1)))

(assert (not
    (and (= x_2 y_0) (= y_1 x_0))))
(check-sat)
(exit)
SMT SOLVERS’ INNER WORKINGS

2-bit adder without overflow
Z = X + Y

Z_0 = x_0 XOR y_0
Z_1 = x_1 XOR y_1 XOR (x_0 AND y_0)

Define
F = x_0 AND y_0

Then
Z_1 = x_1 XOR y_1 XOR F

This happens to be one way to do it in silicon...

- Bit blasting: translating the problem as if it was e.g. an electric circuit. Lots of variables, but that’s OK
- Abstraction Refinement: Lazy translation. If part of the formula is unsatisfiable, that may come out early.
- Constant propagation
- Some high-level reasoning
- Reminds me of LLVM bitcode transformations
from z3 import *
x, y, z = BitVecs("x y z", 32)

s = Solver()
s.add(x % 7 == 0)
s.add((x | y) & 0xFF == 123)
s.add(x > 1337337)
s.add(z == (x + y)/2)
s.add(z < 42424242)

if s.check():
    print(s.model())

$ time python crack.py
[y = 3625989961, x = 719564090, z = 25293377]
real 0m0,255s

(shamefully copied from
https://0x00sec.org/t/black-magic-using-z3-for-automated-crackme-solving/6889)
CONCOLIC EXECUTION - KLEE

1. Provide source code, compile to LLVM bc
2. Set dynamic input, goal
3. ???
4. Profit!!!

- It works well for uncomplicated C code
- It sucks on anything more complicated: network, I/O, timing, C++, etc.
- But the potential is there

Maze demo of KLEE

image by Felipe Andres Manzanom

https://feliam.wordpress.com/
CONCOLIC EXECUTION IN A VIRTUAL MACHINE - S2E

1. Provide (complicated) code
2. Set dynamic input, recompile
3. ???
4. Profit!!!

• Runs a virtual machine with extra instructions to track variables
• It’s actually usable
• Needs some love
• Needs some time and patience
WHY USE IT?

BECAUSE IT’S LESS HASSLE

• Nobody wants to solve equations on paper
• Breaking crypto by hand is boring. Automate it!
• Rolling your own package dependency resolver is a pain
• Doing brute-force search is slow
• Until KLEE it’s about 2-300k LoC

BECAUSE IT’S FUN

• We all like doing weird stuff
• This is the very definition of weird stuff
• You never know when it will come useful
• Automation is the new buzzword, this thing automates
USE CASES

Breaking encryption -- e.g. used to break the early Oyster cards

Solving dependency graphs -- e.g. conda, but also apt-get (Mancoosi project)

Checking HW correctness -- e.g. the floating point bug in early Pentiums

Checking SW correctness -- e.g. Ariane 5 blowing up due to overflow

Checking for function equivalence -- e.g. optimised bitwise manipulation to get hamming weight

Proving optimality -- e.g. finding optimal differential characteristics for ciphers

Test case generation -- through evenly sampling the solution space

Complicated probability distribution checking -- through solution counting

Checking for liveness --- in fail-operational environments (e.g. airplanes, healthcare)

Fuzzing -- through KLEE/S2E, experimental but very interesting
Questions/Remarks/etc?
Don’t spare me :)

THANKS FOR YOUR TIME