Program analysis and constraint solvers

Edgar Barbosa
SyScan360
Beijing - 2014

Who am I?

- Senior Security Researcher - COSEINC
- Reverse Engineering of Windows kernel, device drivers and hypervisors
- One of the creators of the BluePill hardware virtualisation root kit
- Focus now on automation of bug finding

Topics

- Program analysis
- Bug finding
- SAT solvers
- SMT solvers
- Intermediate languages

Objective

- The objective of the presentation is to show how to use constraint solvers, including SMT solvers for program analysis applications like reverse engineering and bug finding.

Program analysis

Bug finding

Bug finding

- Program analysis and reverse engineering these days are mostly dedicated to one specific goal: finding software vulnerabilities.
- Like it or not, this is true. Reverse engineering main use is no more only on understanding and modifying applications, but as a support tool to find bugs.

How to find vulnerabilities?

- How to find vulnerabilities in closed-source applications?
- Black box testing is a method of evaluating a software system by manipulating only its exposed interfaces.
- The most known black box testing tools are fuzzers.

Fuzzing

- Fuzz test, or fuzzing is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program.

Fuzzing Phases

1. Identify targets
2. Identify inputs
3. Generate fuzzed data
4. Execute with fuzzed data
5. Monitoring for exceptions
6. Determine exploitability

Fuzzing – input type knowledge

- If we are going to create a fuzzer for a FTP server, we can’t just generate random data and send it to the tested server. It will be very ineffective (with rare exceptions).
- It is necessary to create a fuzzer able to understand the FTP protocol. The same applies to the any other protocol or file formats like .pdf or .doc.

Fuzzing - formats

- The problem is that the knowledge about the format must be inserted by the programmer.
- What if the protocol/format is unknown?
- What if there is a unknown checksum algorithm?
- Even when the protocol is open, some implementations don't respect the protocol specification.

Reverse Engineering

- With reverse engineering we can extract protocol/format information.
- Some high-level information is lost in the compilation process but all the information necessary to understand how the application works is coded inside the executable file.
- This includes the protocol and file parsers.

Fuzzing

- Fuzzers are still highly effective bug finders.
- It generates so many crashes that the problem now has become Root Cause Analysis to determine the exploitability of the crash.
- Does exist methods that could automate the fuzzing process without requiring the programmer to learn a new protocol or file format specification?
- We are lazy and learning new formats and protocols is time consuming.

Automated bug finding

- We want a system able to:
  - Understand how the input data is able to affect the execution of software
  - audit the program functions without any previous knowledge about protocols or file formats
  - reports bugs with immediate root cause analysis results
  - do not generate false positives
  - increase code/path coverage automatically

Constraint Solvers

SyScan360

Constraint Solvers

- Constraint solvers to the rescue.
- Can help us to learn file formats and protocols and to automatically increase code/ path coverage

- The idea is to translate program analysis problems to be solved by constraint solvers.

- What are constraint solvers?

Constraint programming

"Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." [E. Freuder]

Constraint solvers

- The user specifies the constraints of the objects (variables) using some specific language and the solver will try to find values for each variable able to satisfy each constraint.

Boolean satisfiability

- The most famous satisfiability problem is the Boolean Satisfiability Problem (SAT)
- NP-complete problem!
- Even with this complexity it has been used to solve problems in model checking, formal verification and other areas consisting of thousands of variables and millions of constraints!

Propositional formulas

- SAT problems are encoded as formulas
- In propositional logic, a propositional formula is a type of syntactic formula which is well formed and has a truth value.

SAT solving

- Find satisfying assignment to a propositional logic formula
- Is it possible to satisfy this problem?

$$(x_1 \vee x_3 \vee \bar{x_4}) \wedge (x_4) \wedge (x_2 \vee \bar{x_3})$$

- If you want to use a SAT solver to solve your problem, you need to translate your problem to a boolean formula using the CNF form.

CNF

- Conjunctive Normal Form
- It is common to require that the boolean expression be written in conjunction normal form or "CNF". A formula in conjunctive normal form consists:
  - clauses joined by AND;
  - each clause, in turn, consists of literals joined by OR;
  - each literal is either the name of a variable (a positive literal, or the name of a variable preceded by NOT (a negative literal).

DIMACS input format

- The file can start with comments, that is lines beginning with the character 'c'.
- Right after the comments, there is the line "p cnf nbvar nbclauses" indicating that the instance is in CNF format; nbvar is the number of a variables appearing in the file; nbclauses is the exact number of clauses contained in the file.

DIMACS input format

- Then the clauses follow. Each clause is a sequence of distinct non-null numbers between -nbvar and nbvar ending with 0 on the same line. Positive numbers denote the corresponding variables. Negative numbers denote the negations of the corresponding variables.

Law of excluded middle

```
c law-of-excluded-middle
c
p cnf 1 1
1 -1 0
```

SAT - DEMO

$$(x_1 \vee x_3 \vee \bar{x_4}) \wedge (x_4) \wedge (x_2 \vee \bar{x_3})$$

SAT Encoding

(automatically generated from problem specification)

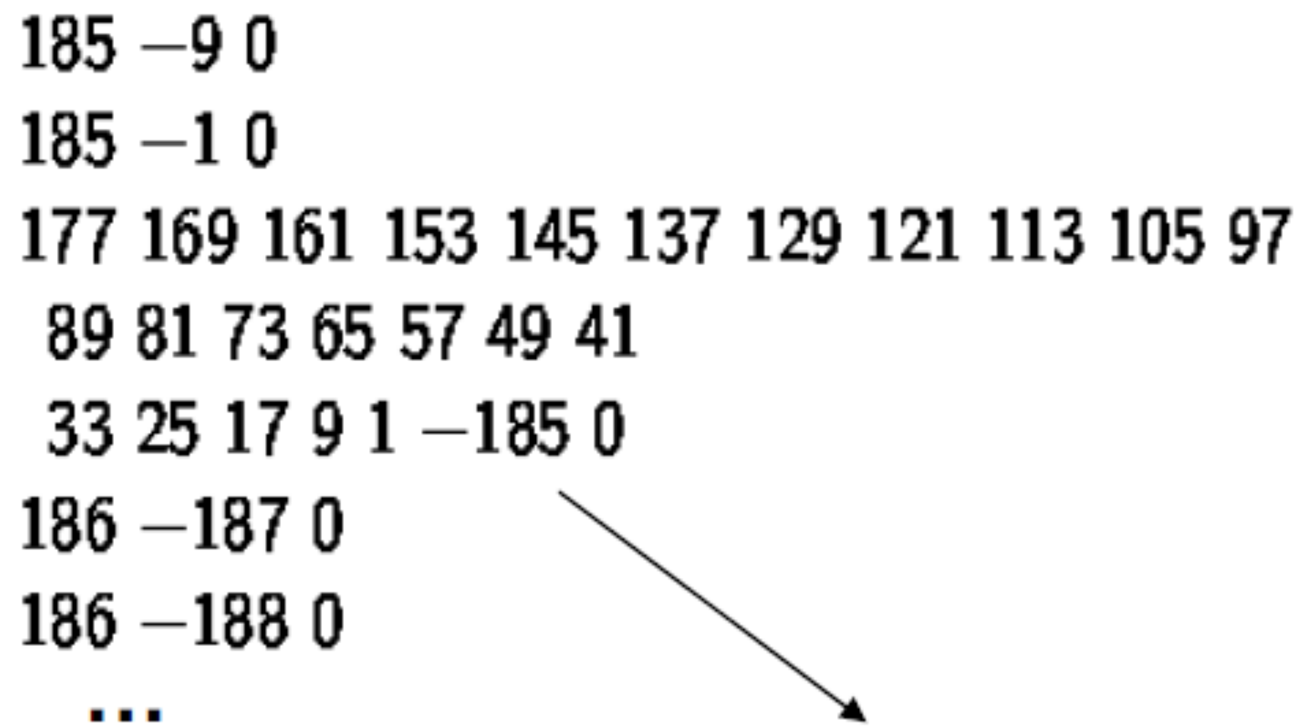
```
p cnf 51639 368352
−1 7 0
−1 6 0
−1 5 0
−1 −4 0
−1 3 0
−1 2 0
−1 −8 0
−9 15 0
−9 14 0
−9 13 0
−9 −12 0
−9 11 0
−9 10 0
−9 −16 0
−17 23 0
−17 22 0
```

i.e., ((not x_1) or x_7)
((not x_1) or x_6)
etc.

x_1, x_2, x_3, etc. are our Boolean variables (to be set to True or False)

Should x_1 be set to False??

10 Pages Later:



i.e., (x_{177} or x_{169} or x_{161} or x_{153} …
x_{33} or x_{25} or x_{17} or x_9 or x_1 or (not x_{185}))

clauses / constraints are getting more interesting…

Note x_1 …

4,000 Pages Later:

```
10236 −10050 0
10236 −10051 0
10236 −10235 0
10008 10009 10010 10011 10012 10013 10014
 10015 10016 10017 10018 10019 10020 10021
 10022 10023 10024 10025 10026 10027 10028
 10029 10030 10031 10032 10033 10034 10035
 10036 10037 10086 10087 10088 10089 10090
 10091 10092 10093 10094 10095 10096 10097
 10098 10099 10100 10101 10102 10103 10104
 10105 10106 10107 10108 −55 −54 53 −52 −51 50
 10047 10048 10049 10050 10051 10235 −10236 0
10237 −10008 0
10237 −10009 0
10237 −10010 0
```

...

Finally, 15,000 Pages Later:

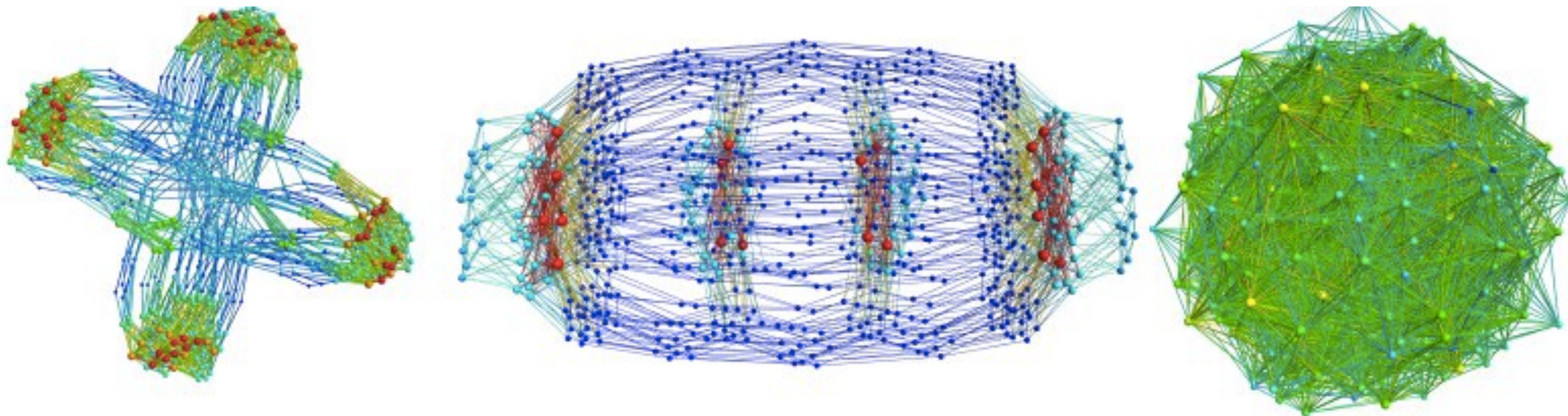
-7 260 0
7 -260 0
1072 1070 0
-15 -14 -13 -12 -11 -10 0
-15 -14 -13 -12 -11 10 0
-15 -14 -13 -12 11 -10 0
-15 -14 -13 -12 11 10 0
-7 -6 -5 -4 -3 -2 0
-7 -6 -5 -4 -3 2 0
-7 -6 -5 -4 3 -2 0
-7 -6 -5 -4 3 2 0
185 0

Search space of truth assignments: $2^{50000} \approx 3.160699437 \cdot 10^{15051}$

Current SAT solvers solve this instance in just a few seconds!

SAT solvers

- SAT solvers are very powerful
- There is even an internacional competition of SAT solvers



SAT solvers

- To use SAT solvers for bug finding we would need to translate the semantics of x86 instructions as boolean formulas using the DIMACS format. This would be very, very hard to do.
- There is a very cool project that translates the Bitcoin mining problem to the CNF format to be solved by a SAT solver!!!
- http://jheusser.github.io/2013/02/03/satcoin.html
- Fortunately we have another very powerful type of solver, an evolution of the SAT solvers: SMT solvers!

SyScan360

SMT SOLVERS

SMT solvers

- Are like SAT solvers but supports several theories, not only boolean operators
- Extremely powerful
- Expressiveness
  - Much more easy to express the semantics of the x86-64 instructions

SMT solvers

- Allow us to determine the necessary values to satisfy code constraints
- Microsoft Z3 was used to prove the correctness of the hyper-V hypervisor core code.
- Microsoft SAGE project is reported to have found several bugs on MS products.

Microsoft Z3

- Z3 is a Satisfiability Modulo Theories (SMT) solver. That is, it is an automated satisfiability checker for many sorted (i.e., typed) first-order logic with built-in theories, including support for quantifiers. The currently supported theories are:
  - equality over free (aka uninterpreted) function and predicate symbols,
  - real and integer arithmetic (with limited support for non-linear arithmetic),
  - bit-vectors,
  - arrays,
  - tuple/records/enumeration types and algebraic (recursive) data-types.

Z3 SMT solver

- Microsoft Z3 SMT solver
- Online version at http://rise4fun.com/Z3
- Linux/Mac/Windows
- Free for non-commercial projects
- USD 14,950.00 for commercial license.

Z3 theories

- Basics
- Arithmetic
- Bit-vectors
- Arrays

Z3 theories - Basics

Op	Mnmonics	Description
0	true	the constant true
1	false	the constant false
2	=	equality
3	distinct	distincinctness
4	ite	if-then-else
5	and	n-ary conjunction
6	or	n-ary disjunction
7	iff	bi-impliciation
8	xor	exclusive or
9	not	negation
10	implies	implication

Z3 theories - BitVector

- Op Mnmonics Parameters Description
- 0 bit1 constant comprising of a single bit set to 1
- 1 bit0 constant comprising of a single bit set to 0.
- 2 bvneg Unary subtraction.
- 3 bvadd addition.
- 4 bvsub subtraction.
- 5 bvmul multiplication.
- 6 bvsdiv signed division.
- 7 bvudiv unsigned division. The operands are treated as unsigned numbers.
- 8 bvsrem signed remainder.
- 9 bvurem unsigned remainder.
- 10 bvsmod signed modulus.
- 11 bvule unsigned <=.
- 12 bvsle signed <=.
- 13 bvuge unsigned >=.
- 14 bvsge signed >=.
- 15 bvult unsigned <.
- 16 bvslt signed <.
- 17 bvugt unsigned >.
- 18 bvsgt signed >.

Z3 theories - BitVector

- 19 bvand n-ary (associative/commutative) bit-wise and.
- 20 bvor n-ary (associative/commutative) bit-wise or.
- 21 bvnot bit-wise not.
- 22 bvxor n-ary bit-wise xor.
- 23 bvnand bit-wise nand.
- 24 bvnor bit-wise nor.
- 25 bvxnor bit-wise exclusive nor.
- 26 concat bit-vector concatentaion.
- 27 sign n n-bit sign extension.
- 28 zero n n-bit zero extension.
- 29 extract hi:low hi-low bit-extraction.
- 30 repeat n repeat n times.
- 31 bvredor or-reduction.
- 32 bvredand and-reducdtion.
- 33 bvcomp bit-vector comparison.
- 34 bvshl shift-left.
- 35 bvlshr logical shift-right.
- 36 bvrshr arithmetical shift-right.
- 37 bvrotate n n-bit left rotation.
- 38 bvrotate n n-bit right rotation.

tests

```
(simplify (bvule #x0a #xf0))  ; unsigned less or equal
(simplify (bvult #x0a #xf0))  ; unsigned less than
(simplify (bvuge #x0a #xf0))  ; unsigned greater or equal
(simplify (bvugt #x0a #xf0))  ; unsigned greater than
(simplify (bvsle #x0a #xf0))  ; signed less or equal
(simplify (bvslt #x0a #xf0))  ; signed less than
(simplify (bvsge #x0a #xf0))  ; signed greater or equal
(simplify (bvsgt #x0a #xf0))  ; signed greater than
```

asking questions

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (not (= (bvule a b) (bvsle a b))))
(check-sat)
(get-model)
```

Z3 solver

DEMONSTRATION

Translation and Intermediate languages

SyScan360

Code constraints

```
.text:00863614                 movzx   ecx, word ptr [eax]
.text:00863617                 push    esi
.text:00863618                 xor     esi, esi
.text:0086361A                 test    cx, cx
.text:0086361D                 jz      short loc_863647
.text:0086361F                 movzx   ecx, cx
.text:00863622
.text:00863622 loc_863622:
.text:00863622                 cmp     cx, 20h
.text:00863626                 jz      short loc_863655
.text:00863628                 cmp     cx, 9
.text:0086362C                 jz      short loc_863655
.text:0086362E
.text:0086362E loc_86362E:
.text:0086362E                 cmp     cx, 22h
.text:00863632                 jz      loc_86435A
.text:00863638
.text:00863638 loc_863638:
.text:00863638                 push    eax                ; lpsz
```

Translation

- How to model the following instructions using Z3?
- Suppose we control EBX value. How to use Z3 to find a value for EBX that will evaluate JZ to TRUE?

```
mov eax, ebx
sub eax, 0x50
cmp eax, 0x40
jz  _branch2
```

Translation

- Since we want to use SMT solvers to solve the constraints of x86 code, we need to translate x86 instructions to SMT formulas!

- We have 2 alternatives:

  1. Try to translate x86 directly to SMT formulas

  2. Translate x86 to an Intermediate language (IL) and then translate the IL to SMT

x86 -> IL -> SMT

- Most program analysis tools first translates x86 to some intermediate language and then translate the IL to SMT
- Clear advantage: if you need to support other instruction sets, like ARM for example, you just need to create the ARM to IL translator.

REIL

- There are several intermediate languages available. My first experience is with the REIL language.
- REIL is a Reverse Engineering Intermediate Language
- Developed by Zynamics (now Google)
- Used in the BinNavi product
- Translates x86-64 and ARM to REIL
- There are better intermediate languages than REIL. But REIL is easier to understand.

x86 instruction set

- There are some consideration about IL implementation for x86 isa
- x86 instructions have side effects
- x86 semantics can be very complex

x86 – side effects

- push eax (intrinsic operands)

```
t1        ←  eax
esp       ←  esp – 4
[esp]     ←  t1
```

- add eax, ex

  eax ← eax + ebx

  update(eflags) //OF,SF,ZF,AF,CF,PF

REIL instruction set

- Small number of instructions
- This is great because we just need to create a small number of translators from REIL instructions to SMT.
- Unfortunately REIL has several limitations

REIL- arithmetic

- `add` – addition of 2 values
- `sub` – subtraction of 2 values
- `mul` – unsigned multiplication
- `div` – unsigned division
- `mod` – unsigned module
- `bsh` – logical shift operation

REIL – bitwise instructions

- `and` – Boolean and
- `or` – Boolean or
- `xor` – Boolean exclusive or

REIL – data transfer instructions

- LDM – load a value from memory
- STM – store a value to memory
- STR – store a value in a register

```
0000000010025D300   ldm   eax,  , word t0
```

REIL - conditional

- BISZ – compare a value to zero
- JCC – conditional jump

```
000000010025D60D   bisz  word t10,  , byte ZF
000000010025DA00   jcc   byte ZF,  , 0x10025E6
```

REIL - others

- UNDEF
- UNKN
- NOP

Support

- General purpose x86 instructions
- Doesn’t support:
  - FPU
  - SSE, sse2, sse3
  - MMX

  - Doesn’t support segment selectors :-(
  - FS, GS

Basic block (x86)

```
010025CB   notepad.exe::_SkipBlanks@4

010025D3   movzx        ecx, word ds:[eax]
010025D6   cmp          word cx, word 0x20
010025DA   jz           loc_10025E6
```

Basic block (REIL)

```
000000010025D300: ldm [DWORD eax, EMPTY , WORD t0]
000000010025D301: or [DWORD 0, WORD t0, DWORD ecx]
000000010025D600: and [DWORD ecx, WORD 65535, WORD t1]
000000010025D601: and [WORD t1, WORD 32768, WORD t2]
000000010025D602: and [WORD 32, WORD 32768, WORD t3]
000000010025D603: sub [WORD t1, WORD 32, DWORD t4]
000000010025D604: and [DWORD t4, DWORD 32768, WORD t5]
000000010025D605: bsh [WORD t5, WORD -15, BYTE SF]
000000010025D606: xor [WORD t2, WORD t3, WORD t6]
000000010025D607: xor [WORD t2, WORD t5, WORD t7]
000000010025D608: and [WORD t6, WORD t7, WORD t8]
000000010025D609: bsh [WORD t8, WORD -15, BYTE OF]
000000010025D60A: and [DWORD t4, DWORD 65536, DWORD t9]
000000010025D60B: bsh [DWORD t9, DWORD -16, BYTE CF]
000000010025D60C: and [DWORD t4, DWORD 65535, WORD t10]
000000010025D60D: bisz [WORD t10, EMPTY , BYTE ZF]
000000010025DA00: jcc [BYTE ZF, EMPTY , DWORD 16786918]
```

REIL bb → z3 (1/2)

```
(set-logic QF_BV)
(declare-fun t0 () (_ BitVec 32))
(declare-fun ecx () (_ BitVec 32))
(declare-fun t1 () (_ BitVec 32))
(declare-fun t2 () (_ BitVec 32))
(declare-fun t3 () (_ BitVec 32))
(declare-fun t4 () (_ BitVec 32))
(declare-fun t5 () (_ BitVec 32))
(declare-fun SF () Bool)
(declare-fun t6 () (_ BitVec 32))
(declare-fun t7 () (_ BitVec 32))
(declare-fun t8 () (_ BitVec 32))
(declare-fun OF () Bool)
(declare-fun t9 () (_ BitVec 32))
(declare-fun CF () Bool)
(declare-fun t10 () (_ BitVec 32))
(declare-fun ZF () Bool)
```

REIL bb → z3 (2/2)

```
(assert (= t10 (bvand (bvsub (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x0000FFFF)))
(assert (= t6 (bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand #x00000020 #x00008000))))
(assert (= SF (bvugt (bvlshr (bvand (bvsub (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x00008000) #x0000000F)
#x00000000)))
(assert (= t7 (bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand (bvsub (bvand (bvor #x00000000
t0) #x0000FFFF) #x00000020) #x00008000))))
(assert (= OF (bvugt (bvlshr (bvand (bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand
#x00000020 #x00008000)) (bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand (bvsub (bvand (bvor
#x00000000 t0) #x0000FFFF) #x00000020) #x00008000))) #x0000000F) #x00000000)))
(assert (= t5 (bvand (bvsub (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x00008000)))
(assert (= t8 (bvand (bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand #x00000020 #x00008000))
(bvxor (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000) (bvand (bvsub (bvand (bvor #x00000000 t0)
#x0000FFFF) #x00000020) #x00008000)))))
(assert (= t9 (bvand (bvsub (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x00010000)))
(assert (= ZF (bvugt (ite (= (bvand (bvsub (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x0000FFFF) #x00000000)
#x00000001 #x00000000) #x00000000)))
(assert (= t3 (bvand #x00000020 #x00008000)))
(assert (= t2 (bvand (bvand (bvor #x00000000 t0) #x0000FFFF) #x00008000)))
(assert (= CF (bvugt (bvlshr (bvand (bvsub (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020) #x00010000) #x00000010)
#x00000000)))
(assert (= t4 (bvsub (bvand (bvor #x00000000 t0) #x0000FFFF) #x00000020)))
(assert (= ecx (bvor #x00000000 t0)))
(assert (= t1 (bvand (bvor #x00000000 t0) #x0000FFFF)))
(check-sat)
(get-model)
```

solution

sat

```
(model
 (define-fun t0 () (_ BitVec 32)
  #x00000000)
 (define-fun t1 () (_ BitVec 32)
  #x00000000)
 (define-fun ecx () (_ BitVec 32)
  #x00000000)
 (define-fun t4 () (_ BitVec 32)
  #xffffffe0)
 (define-fun CF () Bool
  true)
 (define-fun t2 () (_ BitVec 32)
  #x00000000)
 (define-fun t3 () (_ BitVec 32)
  #x00000000)
 (define-fun ZF () Bool
  false)
 (define-fun t9 () (_ BitVec 32)
  #x00010000)
 (define-fun t8 () (_ BitVec 32)
  #x00000000)
 (define-fun t5 () (_ BitVec 32)
  #x00008000)
 (define-fun OF () Bool
  false)
 (define-fun t7 () (_ BitVec 32)
  #x00008000)
 (define-fun SF () Bool
  true)
 (define-fun t6 () (_ BitVec 32)
  #x00000000)
 (define-fun t10 () (_ BitVec 32)
  #x0000ffe0)
)
```

Automation

Program Analysis and Constraint Solvers

Bug finding automation

- There are several methods to evaluate in the attempt to automate bug finding.
- Some prefer static analysis and other dynamic analysis.
- Intermediate language preference is very personal.
- You have to try some methods and check which one is better for your purpose.

Automation

- The method I propose is based on the incredible Microsoft SAGE project.
- Dynamic analysis
- Has found several bugs on Microsoft products

Process

- Execute target application with an initial seed file
- Trace the execution
- Taint analysis
- Translation of x86 code to SMT formulas
- SMT used to generate new inputs
- Increases code/path coverage

Execution trace

- Great tools for execution trace
- Binary instrumentation: PIN, DynamoRio
- Debuggers (slower)

Taint analysis

- You don't want to translate an entire trace to SMT formulas
- You filter only the instructions affected by user input data (file)
- Taint analysis can be implemented on top of the intermediate language or directly from x86 disassembly
- Biggest problem: system calls!

Taint analysis - syscalls

- How to apply taint analysis in a system where some of the system calls aren't documented? (Windows)

- How can we know what is tainted when an undocumented syscall uses a tainted value? How do we know if the syscall returned information is/isn't tainted? Do we need to trace kernel code?

- Most systems will just consider a very small subset of the syscalls: read, write, open, ..., and create hardcoded rules for taint propagation

Translation

- This is one of the points where it is fundamental to decide if you want or not to use an intermediate language.
- It is possible to create a direct x86 to SMT translator.
- Most basic solution involves the use of some template engine where you will encode most of the translations inside a template.
- Since SMT-LIB doesn't accept multiple assignments to the same variable, you will have to create some kind of versioning system for variables (similar to SSA)
- You can also translate directly to Python code using the awesome Z3Py interface.

Strategy

- After getting results from the Z3 solver, you have new input values.
- What search strategy do you want to use? BFS? DFS? Random?
- You can also give priority to traces that contains some interesting patterns like loops, memory allocation size calculations and others.

Demo

Conclusion

SyScan360

Conclusion

- This is just an introductory presentation about the potencial of constraint solvers for the reverse engineering tasks.
- Program analysis is hard. There are lots of corner cases. Challenging.
- Translation of instruction sets is hard and very time consuming.
- There are a lot of things that can and need to be automated in reverse engineering and program analysis.
- SMT solvers are very powerful and the way to go. However do not use it for everything
- We need more (open source) tools.

Thank you