Advanced Binary Deobfuscation

NTT Secure Platform Laboratories

Yuma Kurogome

% whoami

- Yuma Kurogome
  - Security Researcher @ NTT Secure Platform Laboratories
  - Research Interests: Malware Detection, Analysis, and Anti-Analysis
  - Hobby: Climbing
- Recent Publication
  - Kurogome et al. EIGER: Automated IOC Generation for Accurate and Interpretable Malware Detection. ACSAC'19.
    - Fully-automated behavioral signature generation method
    - Based on combinatorial optimization

Function	Description
$f_1(R) = \lvert S \rvert - size(R)$	Represent the entire dataset with a few rules.
$f_2(R) = \sum_{r \in R} (\max_{r \in S \times C} width(r) - width(r))$	Highly evaluate a concise rule *i.e.*, rule with a few representations. We redefined $width(r)$ to the number of regular expression objects in r.
$f_3(R) = \sum_{\substack{r_i, r_j \in R \\ i \le j \\ c_i = c_j}} (N - \lvert overlap(r_i, r_j) \rvert)$	Reduce the intra-class *overlap* of rules. N is the number of points in the dataset.
$f_5(R) = \lvert C \rvert - \sum_{c' \in C} \mathbb{1}(\exists r = (s,c) \in R \text{ such that } c = c')$	Have at least one rule for each class. We removed this function from our objective. The function $\mathbb{1}$ returns 1 whenever the condition it takes as an argument is true and 0 otherwise.
$f_6(R) = N \cdot \lvert S \rvert - \sum_{r \in R} \lvert incorrect\text{-}cover(r) \rvert$	To encourage precision, reduce the size of *incorrect-cover* sets.
$f_7(R) = N - \sum_{(\mathbf{x},y) \in D} \mathbb{1}(\lvert \{r \mid (\mathbf{x},y) \in correct\text{-}cover(r)\} \rvert \ge 1)$	To encourage recall, have at least one accurate rule describes each data points.

Introduction

Overview

- Abstract
  - Reverse engineering is not easy, especially if a binary code is obfuscated. Once obfuscation performed, the binary would not be analyzed accurately with naive techniques alone. In this course, you will learn obfuscation principles (especially used by malware), theory and practice of obfuscated code analysis, and how to write your own tool for deobfuscation. In particular, we delve into data-flow analysis and SAT/SMT-based binary analysis (e.g., symbolic execution) to render obfuscation ineffective.
- Objective
  - Understand binary obfuscation techniques used in malware
  - Acquire a skill of writing deobfuscation tools

Overview

- At the end of this course, you will be able to:
  - Have an in-depth understanding of theory, practice, and behind insights of obfuscation
  - Build a custom obfuscated payload with state-of-the-art packers
  - Apply compiler optimization techniques to binary analysis tasks
  - Design and implement automated binary analysis tools top on a symbolic execution engine
  - Even analyze obfuscated malware used in the APT campaign

Outline

- Introduction
- Obfuscation Techniques
  - Preliminaries
  - Garbage Code Insertion, Instruction Substitution, …
  - Hands-On
- Deobfuscation Techniques
  - Preliminaries
  - Dataflow Analysis, Symbolic Execution, Equivalence Checking, …
  - Hands-On
- Conclusion

Obfuscation Techniques

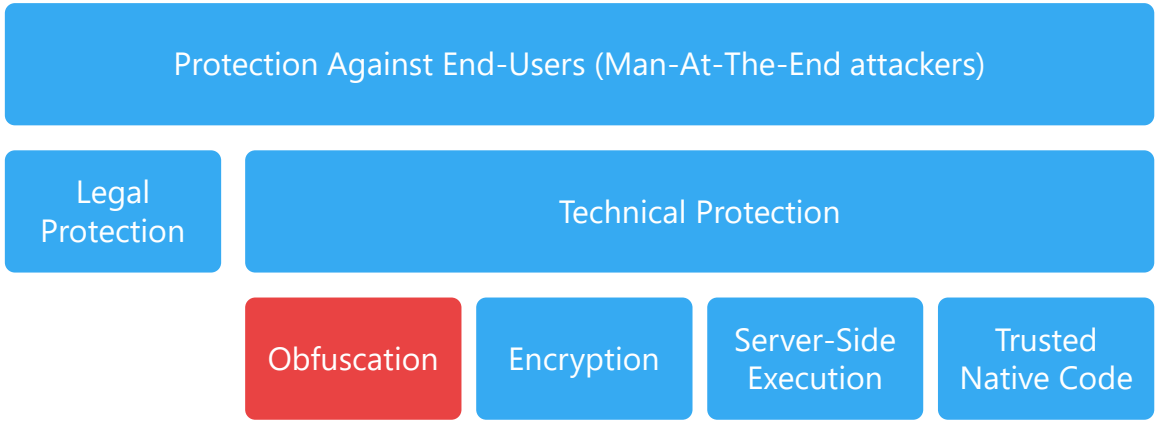
Obfuscation

àbfəskéɪʃən

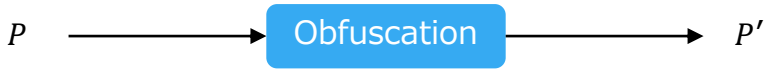
Obfuscation

↕

Deobfuscation



- Definition (Informal)
  - Obfuscation is a transformation from program P to functionally equivalent program P' which is harder to extract information than from P.

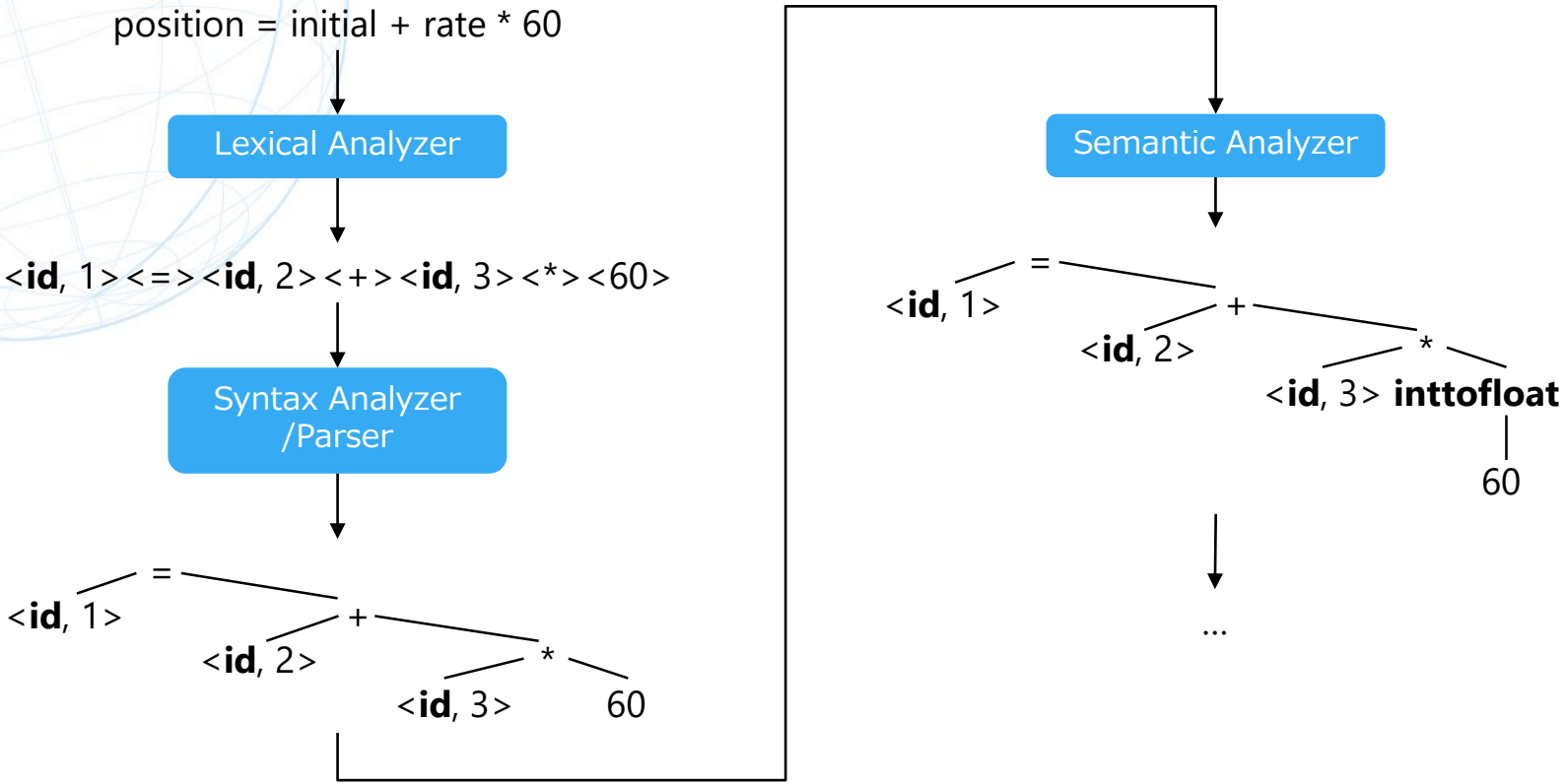


Compiler

- To understand the obfuscation techniques, we delve into an architecture of a compiler e.g., LLVM:
  - Frontend
  - Backend

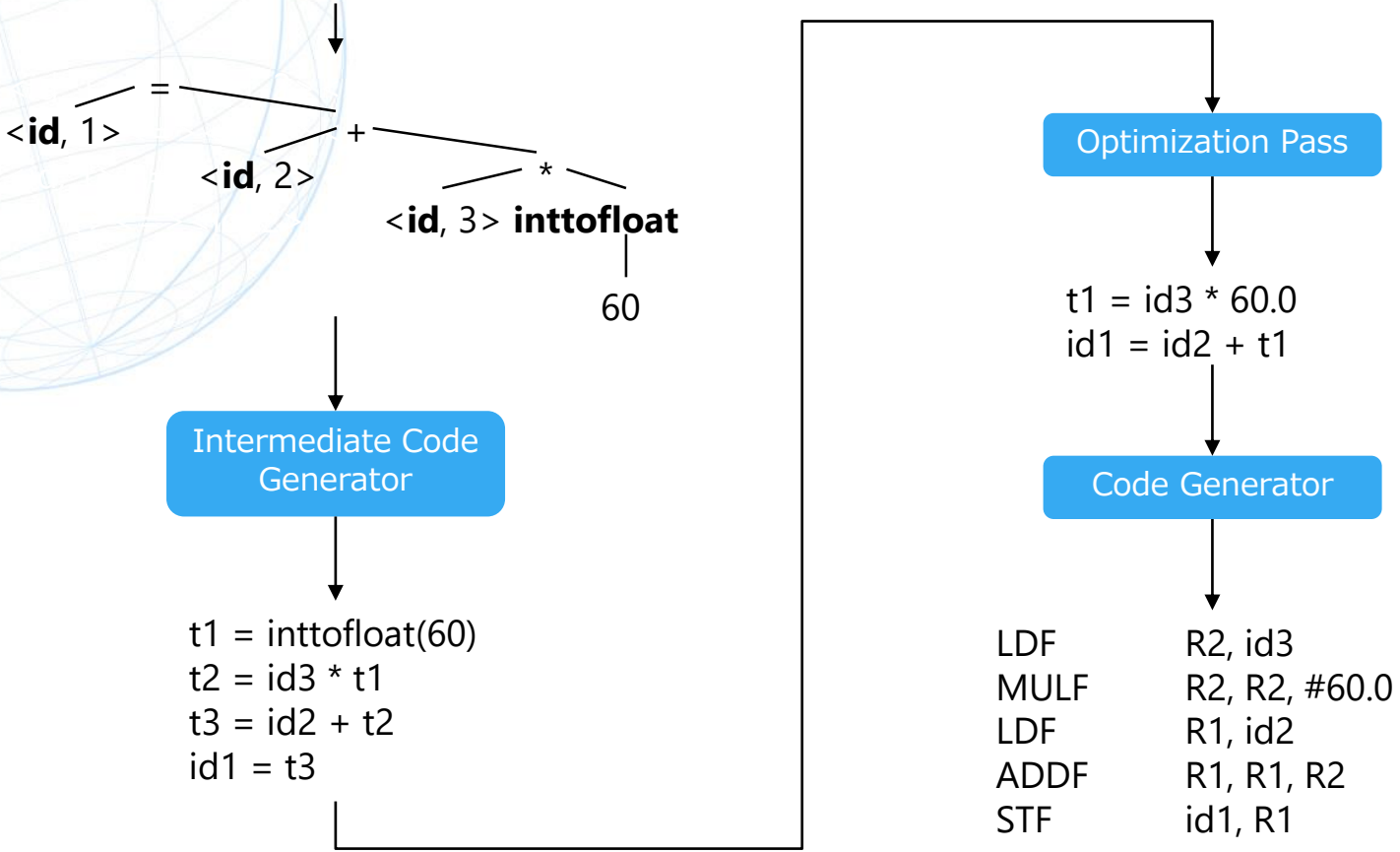
Compiler Frontend

Aho et al. Compilers: Principles, Techniques, and Tools. 1986



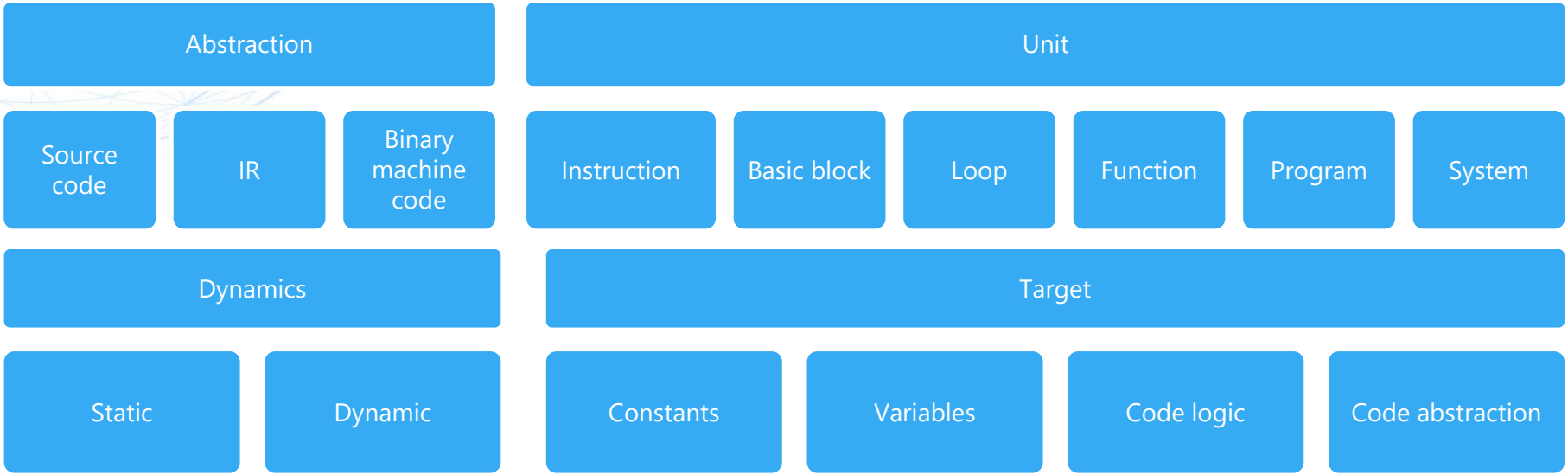
Compiler Backend

Aho et al. Compilers: Principles, Techniques, and Tools. 1986



Taxonomy of Obfuscation

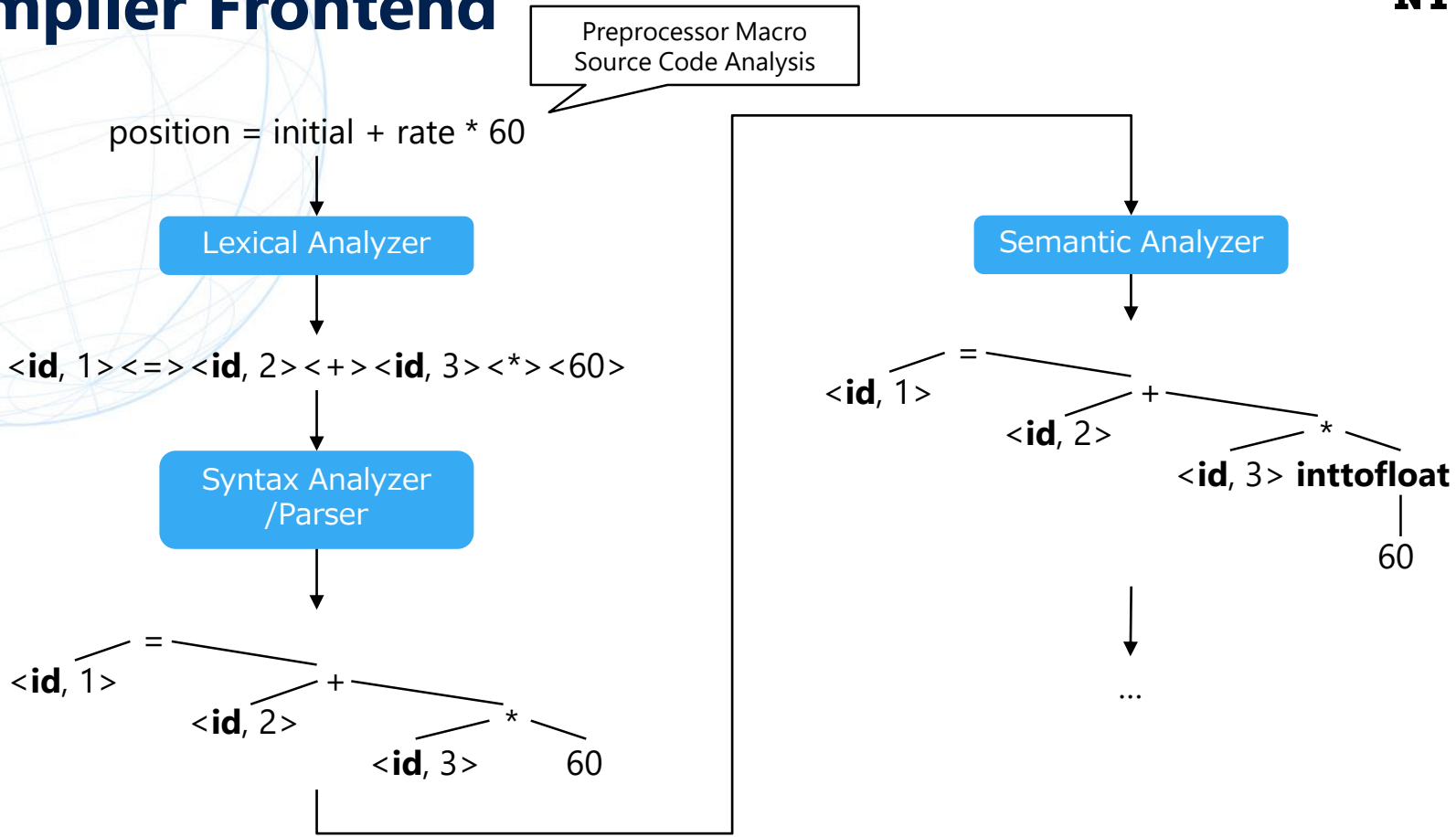
- When, where, and how to apply obfuscation is closely related to such a compiler architecture



*Basic Block is a straight-line instruction sequence with no branches in except to the entry and no branches out except at the exit

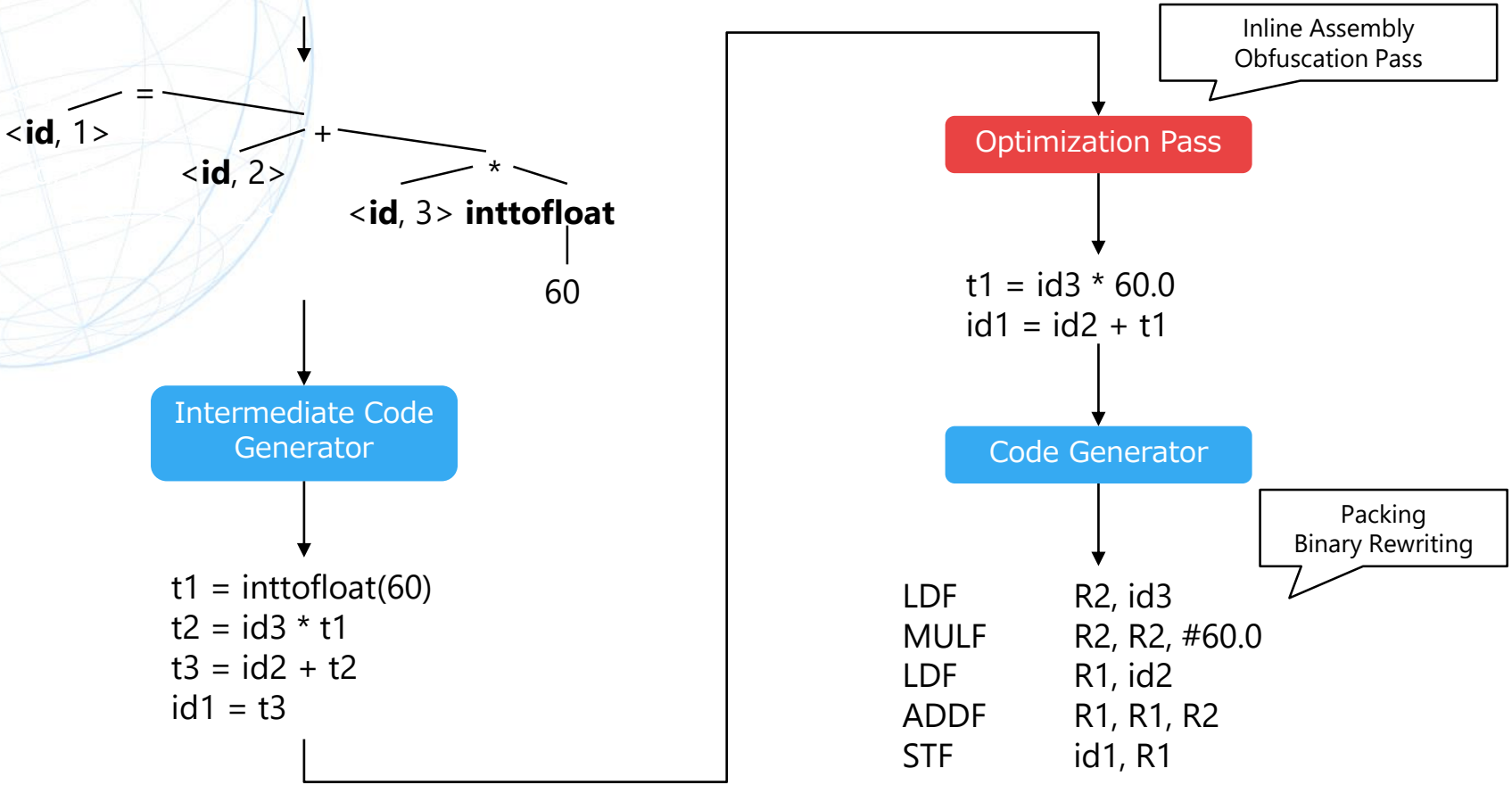
Compiler Frontend

Aho et al. Compilers: Principles, Techniques, and Tools. 1986



Compiler Backend

Aho et al. Compilers: Principles, Techniques, and Tools. 1986



Obfuscation Techniques

- There are 31 known obfuscation transformations
  - Most of them are applicable at the same time

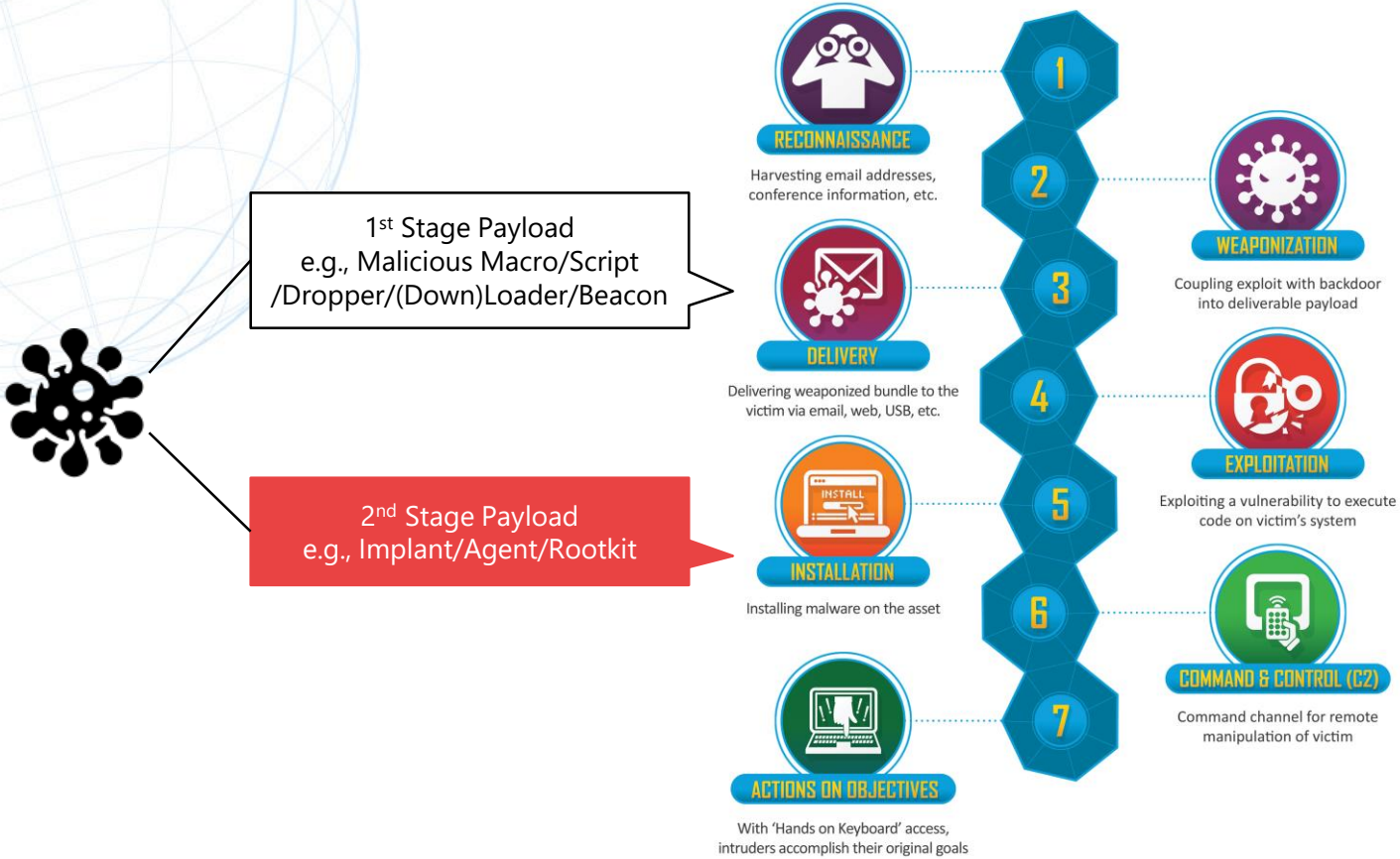
Obfuscation Transformation	Abstraction	Unit	Dynamics	Target
Opaque Predicates	All	Function	Static	Data constant
Convert static data to procedural data	All	Instruction	Static	Data constant
Mixed Boolean Arithmetic	All	Basic block	Static	Data constant
White-box cryptography	All	Function	Static	Data constant
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

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

Obfuscation Tools

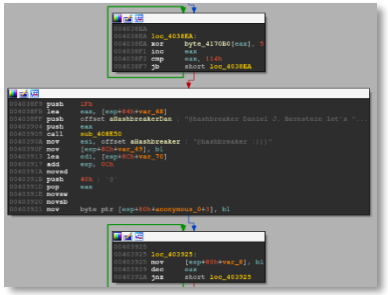
- Commercial
  - Themida
  - Code Virtualizer
  - VMProtect
  - Enigma
  - Epona
  - …
- Academic
  - Tigress
  - Obfuscator-LLVM (O-LLVM)
  - …

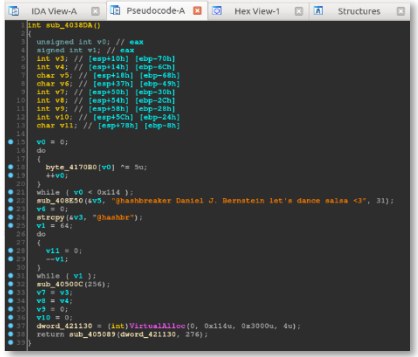
Obfuscation in Malware



https://icons8.com/

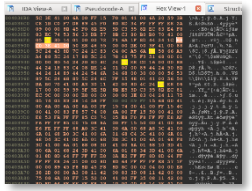
Obfuscation in Malware





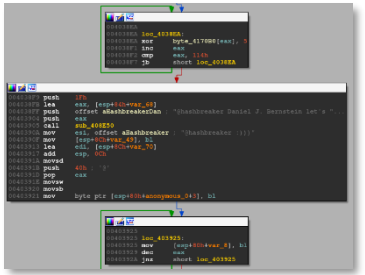
https://icons8.com/

Obfuscation in Malware



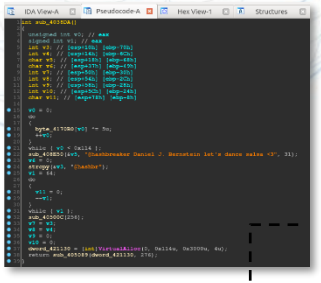
Source Code → Intermediate Representation → Binary Machine Code

Binary Machine Code → Assembly Code → Intermediate Representation → Source Code



https://icons8.com/

Obfuscation in Malware



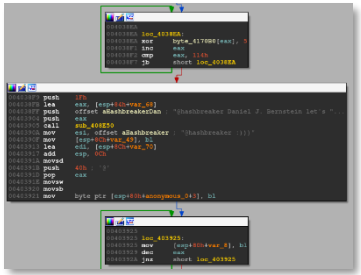
Preprocessor Macro
Source Code Analysis

Inline Assembly
Obfuscation Pass

Packing
Binary Rewriting

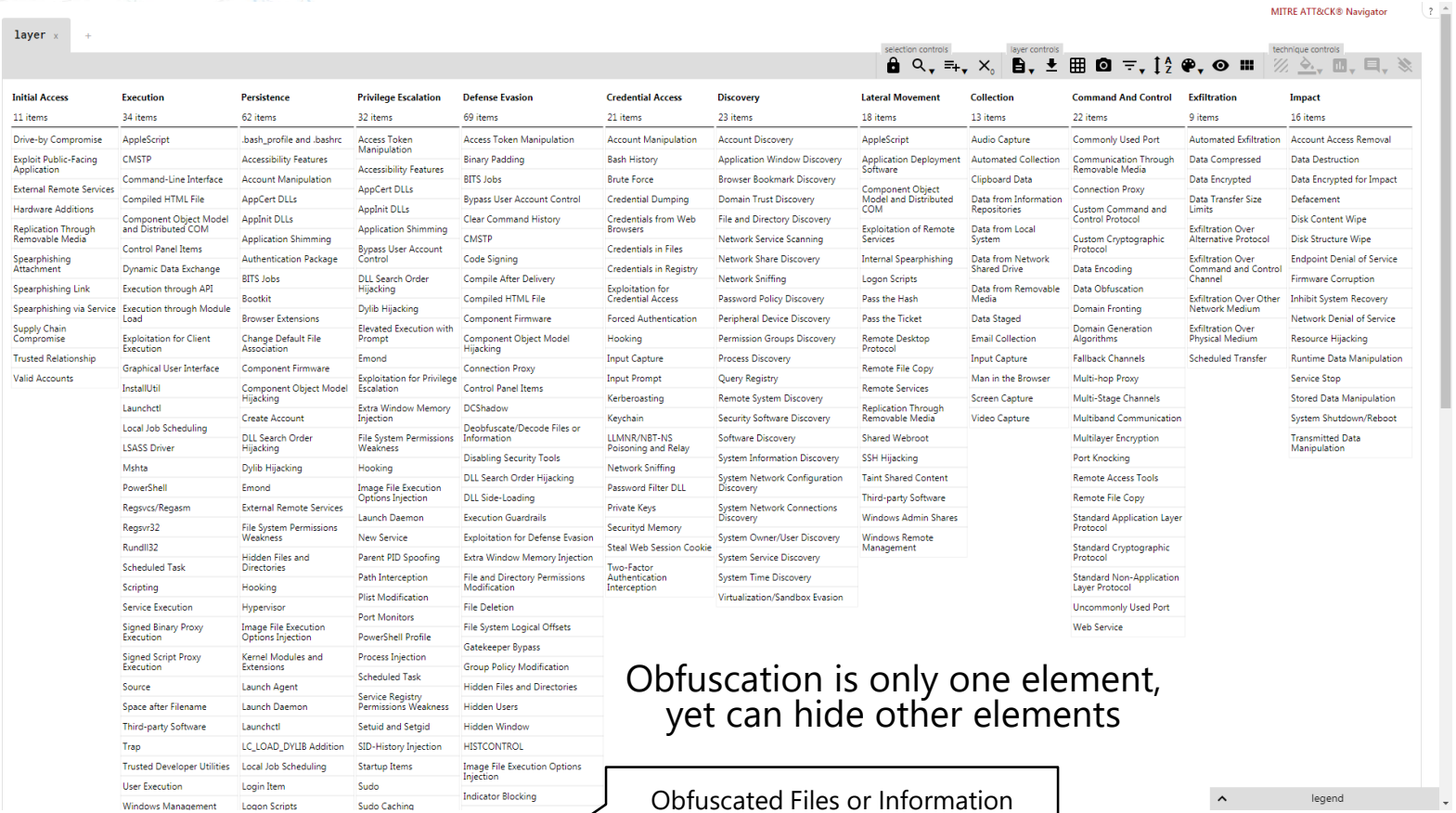
Source Code → Intermediate Representation → Binary Machine Code

Assembly Code → Intermediate Representation → Source Code



https://icons8.com/

Obfuscation in Malware



Obfuscation is only one element, yet can hide other elements

Obfuscated Files or Information

...

Obfuscation Techniques

- Different techniques, common ideas:
  - Do useless things
    - Garbage/Dead Code Insertion
  - Change syntax
    - Instruction Substitution
    - Encode Literals
    - Encode Arithmetic
  - Change not only syntax but also semantics
    - Opaque Predicate
    - Control Flow Flattening
    - Virtualization Obfuscation

Garbage/Dead Code Insertion

- Insert code which does not affect an intended result
  - If the code always executes, it's called garbage code;
  - If the code never executes, it's called dead code
- Often combined with other obfuscations

```
3 uint32 div5 (uint32 a, b )
4 {
5               uint32 x = 0;
6               x = (a + b)/5;
7               return x;
8 }
```

```
...
mov eax, [ebp+arg_4]
mov ecx, [ebp+arg_0]
mov edx, 5
mov [ebp+var_4], ecx
mov [ebp+var_8], eax
mov eax, [ebp+var_4]
add eax, [ebp+var_8]
mov ecx, 0
mov [ebp+var_10], edx
mov edx, ecx
mov ecx, [ebp+var_10]
div ecx
mov [ebp+var_C], eax
...
```

```
...
mov edx, 0xdeadc00d
mov eax, [ebp+arg_4]
mov ecx, [ebp+arg_0]
mov edx, 5
mov [ebp+var_4], ecx
mov [ebp+var_8], eax
mov eax, [ebp+var_4]
add eax, [ebp+var_8]
mov ecx, [ebp+var_8]
mov ecx, 0
mov [ebp+var_10], edx
mov edx, ecx
mov ecx, [ebp+var_10]
div ecx
mov [ebp+var_C], eax
...
```

Instruction Substitution

- Replace code with complex, yet equivalent code

```
...
mov eax, [ebp+arg_0]
mov [ebp+var_4], eax
mov [ebp+var_8], 0Ch
mov [ebp+var_C], 38h ; '8'
mov [ebp+var_10], 7Fh
mov eax, [ebp+var_8]
add eax, [ebp+var_C]
add eax, [ebp+var_10]
add eax, [ebp+var_4]
mov [ebp+var_14], eax
mov eax, [ebp+var_14]
...
```

```
...
mov eax, [ebp+arg_0]
xor ecx, ecx
mov [ebp+var_8], eax
mov [ebp+var_C], 0Ch
mov [ebp+var_10], 38h ; '8'
mov [ebp+var_14], 7Fh
mov eax, [ebp+var_C]
mov edx, [ebp+var_10]
sub eax, 2598A32Bh
add eax, edx
add eax, 2598A32Bh
mov edx, [ebp+var_14]
```

```
mov esi, ecx
sub esi, eax
mov eax, ecx
sub eax, edx
add esi, eax
mov eax, ecx
sub eax, esi
mov edx,
[ebp+var_8]
sub ecx, edx
sub eax, ecx
mov [ebp+var_18],
eax
mov eax,
[ebp+var_18]
...
```

*Obfuscated by O-LLVM –sub option

Encode Literals

- Replace literals (constants, strings) with more complex expressions
  - By dividing/encoding them

```
lea eax, format ; "%s¥n"
lea ecx, aHelloWorld ; "hello world"
mov [ebp+var_4], ecx
mov ecx, [ebp+var_4]
mov [esp], eax ; format
mov [esp+4], ecx
call _printf
mov [ebp+var_8], eax
```

```
mov edx, [ebp+var_4]
mov eax, [ebp+arg_4]
add eax, edx
mov byte ptr [eax], 68h ; 'h'
add [ebp+var_4], 1
mov edx, [ebp+var_4]
mov eax, [ebp+arg_4]
add eax, edx
mov byte ptr [eax], 65h ; 'e'
add [ebp+var_4], 1
mov edx, [ebp+var_4]
mov eax, [ebp+arg_4]
add eax, edx
mov byte ptr [eax], 6Ch ; 'l'
add [ebp+var_4], 1
mov edx, [ebp+var_4]
mov eax, [ebp+arg_4]
add eax, edx
mov byte ptr [eax], 6Ch ; 'l'
add [ebp+var_4], 1
mov edx, [ebp+var_4]
mov eax, [ebp+arg_4]
add eax, edx
mov byte ptr [eax], 6Fh ; 'o'
```

```
add [ebp+var_4], 1
mov edx, [ebp+var_4]
mov eax, [ebp+arg_4]
add eax, edx
mov byte ptr [eax], 20h ; ' '
add [ebp+var_4], 1
mov edx, [ebp+var_4]
mov eax, [ebp+arg_4]
add eax, edx
mov byte ptr [eax], 77h ; 'w'
add [ebp+var_4], 1
mov edx, [ebp+var_4]
mov eax, [ebp+arg_4]
add eax, edx
mov byte ptr [eax], 6Fh ; 'o'
add [ebp+var_4], 1
mov edx, [ebp+var_4]
mov eax, [ebp+arg_4]
add eax, edx
mov byte ptr [eax], 72h ; 'r'
add [ebp+var_4], 1
mov edx, [ebp+var_4]
mov eax, [ebp+arg_4]
add eax, edx
mov byte ptr [eax], 6Ch ; 'l'
...
```

*Obfuscated by Tigress EncodeLiterals option

Encode Arithmetic/Mixed Boolean Arithmetic

- Replace arithmetic/Boolean operations with more complex expressions

```
...
mov eax, [ebp+arg_0]
mov [ebp+var_4], eax
mov [ebp+var_8], 0Ch
mov [ebp+var_C], 38h ; '8'
mov [ebp+var_10], 7Fh
mov eax, [ebp+var_8]
add eax, [ebp+var_C]
add eax, [ebp+var_10]
add eax, [ebp+var_4]
mov [ebp+var_14], eax
mov eax, [ebp+var_14]
...
```

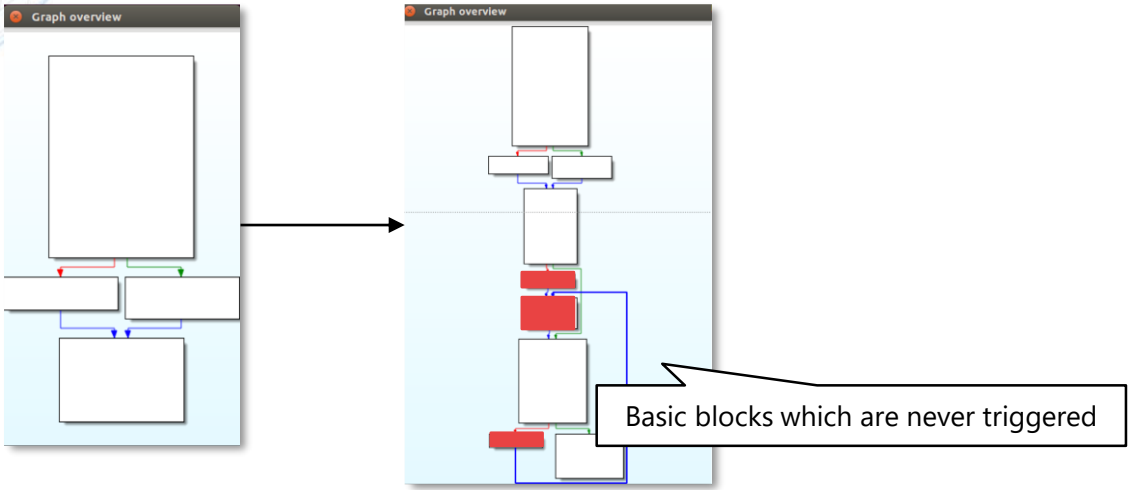
```
...
mov [ebp+var_10], 0Ch
mov [ebp+var_C], 38h ; '8'
mov [ebp+var_8], 7Fh
mov eax, [ebp+var_C]
not eax
mov edx, eax
mov eax, [ebp+var_10]
sub eax, edx
lea edx, [eax-1]
mov eax, [ebp+var_8]
not eax
sub edx, eax
mov eax, edx
sub eax, 1
or eax, [ebp+arg_0]
```

```
mov edx, eax
mov eax, [ebp+var_C]
not eax
mov ecx, eax
mov eax, [ebp+var_10]
sub eax, ecx
lea ecx, [eax-1]
mov eax, [ebp+var_8]
not eax
sub ecx, eax
mov eax, ecx
sub eax, 1
and eax, [ebp+arg_0]
add eax, edx
mov [ebp+var_4], eax
mov eax, [ebp+var_4]
...
```

*Obfuscated by Tigress EncodeArithmetic option

Opaque Predicate

- Insert a conditional branch (predicate) which is never or always be triggered
- Make an unconditional branch conditional



*Obfuscated by O-LLVM –bcf option

Opaque Predicate

- Achieved by inserting deterministic operations:

$7y^2 - 1 \neq x^2$
$2 \mid x(x+1)$
$3 \mid x(x+1)(x+2)$
$x^2 > 0$
$7x^2 + 1 \not\equiv 0 \mod 7$
$x^2 + x + 7 \not\equiv 0 \mod 81$
$x > 0$ for $x \in I$ random where $I \subset X \setminus \{0\}$ a random interval

Arithmetic

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

```
__always_taken:
...
```

```
__never_taken:
...
```

Pseudo-Handle

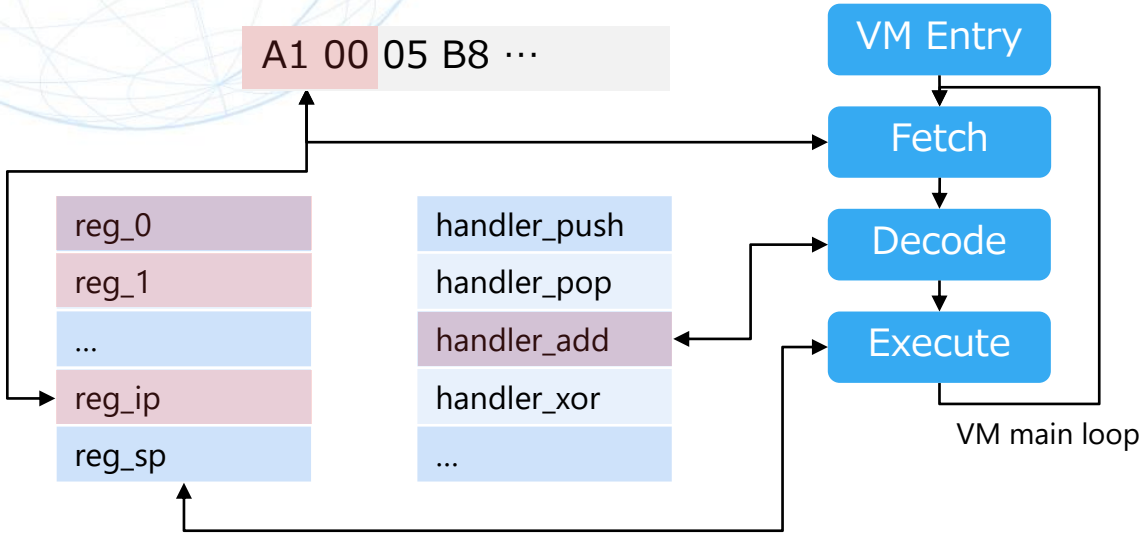
$$f(n) = \begin{cases} \frac{n}{2} & if\ n\%2 = 0 \\ 3n+1 & if\ n\%2 = 1 \end{cases} \longrightarrow 1$$

Collatz Conjecture

Zobernig et al. Indistinguishable Predicates: A New Tool for Obfuscation. 2017.
Wang et al. Linear Obfuscation to Combat Symbolic Execution. ESORICS'11.

Virtualization Obfuscation

- Replace code with unique bytecode
  - Execute it on a virtual machine
- Bytecode format is independent to the host ISA

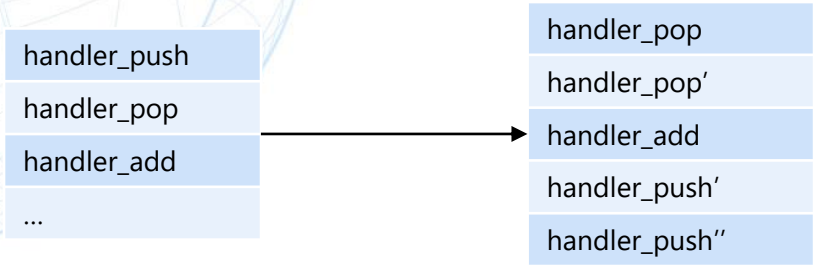


*Obfuscated by Tigress –Virtualize option

```
unsigned int target_function(int n )
{
  char _1_target_function_$locals[8] ;
  union _1_target_function_$node _1_target_function_$stack[1][32] ;
  union _1_target_function_$node *_1_target_function_$sp[1] ;
  unsigned char *_1_target_function_$pc[1] ;
  {
  _1_target_function_$sp[0] = _1_target_function_$stack[0];
  _1_target_function_$pc[0] = _1_target_function_$array[0];
  while (1) {
    switch (*(_1_target_function_$pc[0])) {
    case _1_target_function__load_int$left_STA_0$result_STA_0:
    (_1_target_function_$pc[0]) ++;
    (_1_target_function_$sp[0] + 0)->_int
          = *((int *)(_1_target_function_$sp[0] + 0)->_void_star);
    break;
    case _1_target_function__branchIfTrue$expr_STA_0$label_LAB_0:
    (_1_target_function_$pc[0]) ++;
    if ((_1_target_function_$sp[0] + 0)->_int) {
      _1_target_function_$pc[0] += *((int *)_1_target_function_$pc[0]);
    } else {
      _1_target_function_$pc[0] += 4;
    }
...
```

Virtualization Obfuscation

- Handler Duplication
  - Generate diversified instruction handlers from a handler template



- Direct Threaded Code
  - Hide VM main loop by decentralize the dispatcher
  - Originally a performance optimization technique

```
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*

Virtualization Obfuscation

- Limitations
  - Loop
    - May conflict with loop of the VM itself
  - Switch/Case statements
  - Exception handling

Control Flow Flattening

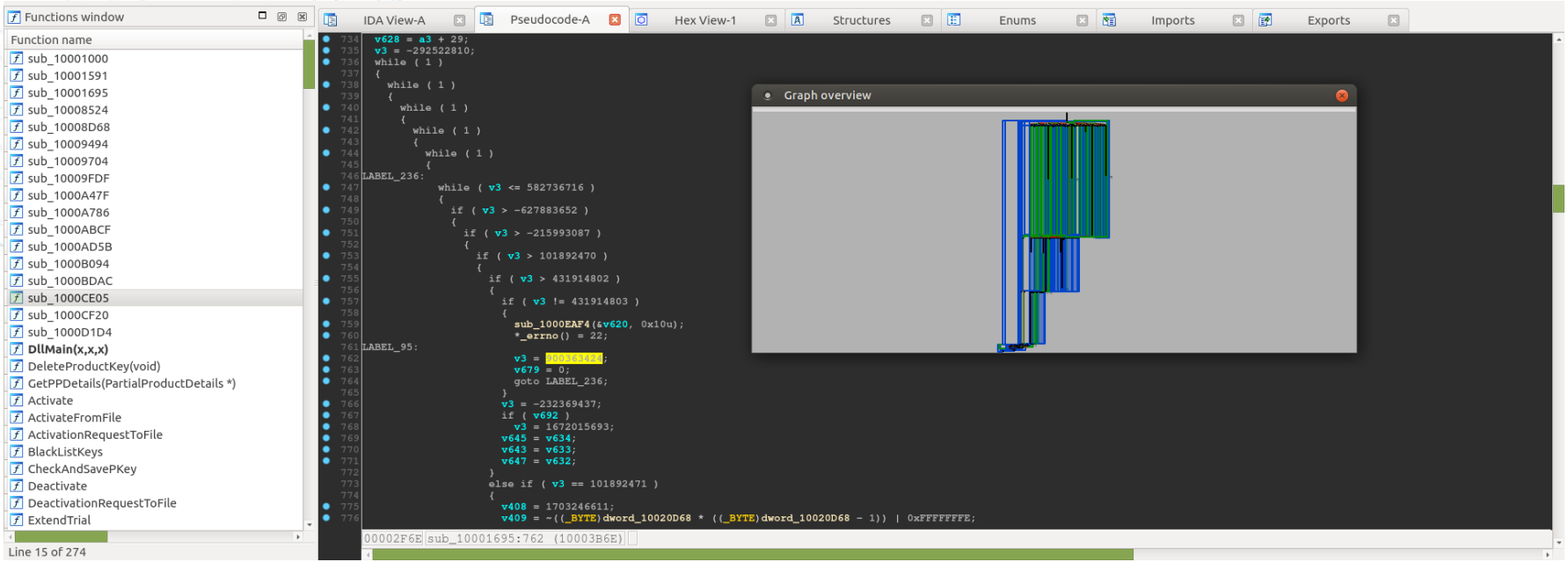
- Flatten each basic block as a case of the switch statement
  - Jump to the next block to be executed based on index in the switch statement
  - Update the index that points to the next block as each block executes

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

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

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

Control Flow Flattening

- Example: ANEL



MD5: a79f59b1b17e8bfa3299e50a8af9cdaf
ANEL is a RAT used by APT10 a.k.a. MenuPass, Stone Panda, or Red Apollo
Haruyama. Defeating APT10 Compiler-Level Obfuscations. VB'19.

Hands-On 1: Obfuscation

- Duration: 20min
- Objective: Obfuscate sample code under the hands-on1 directory
- Step1: Prepare targets
  - test-add.c, test-mod2.c, test-hello.c, test-mod2-add.c
  - Write your own code
- Step2: Apply obfuscation transformations
  - O-LLVM: o-llvm.sh
    - ./o-llvm.sh ./src/ ../obfuscator/build/bin/clang
  - Tigress: tigress.sh
    - ./tigress.sh ./src/
      - Execute Tigress inside Docker container

Hands-On 1: Obfuscation

- Step3: Compare obfuscated binary files and un-obfuscated ones
  - How to build un-obfuscated binary files
    - make
  - Take a look at obfuscated source code generated by Tigress

Sample Code

```
3 unsigned int target_function(int n)
4 {
5           int a, b, c, r;
6
7           a = 12;       //0x0C
8           b = 56;       //0x38
9           c = 127;      //0x7F
10
11          r = a + b + c + n;
12
13          return r;
14 }
```

test-add.c

```
3 void target_function(void)
4 {
5           char* msg = "hello world";
6           printf("%s¥n", msg);
7 }
```

test-hello.c

```
3 unsigned int target_function(int n)
4 {
5           if(n % 2 == 0){
6                       return 0;
7           }else{
8                       return 1;
9           }
10 }
```

test-mod2.c

O-LLVM

- Obfuscates a given intermediate representation of LLVM (LLVM-IR)
  - Implemented as an optimization pass of the compiler toolchain
- Options
  - sub: Instruction Substitution
  - fla: Control Flow Flattening
  - bcf: Opaque Predicate (as it is also referred to as Bogus Control Flow)
- References
  - https://github.com/obfuscator-llvm/obfuscator/wiki
  - Junod et al. Obfuscator-LLVM – Software Protection for the Masses. SPRO'15.
- Running Examples
  - ${OLLVM} –m32 –mllvm –sub ${FILE} –o "${FILE_NO_EXT##*/}-sub.bin"
  - ${OLLVM} –m32 –mllvm –fla ${FILE} –o "${FILE_NO_EXT##*/}-fla.bin"
  - ${OLLVM} –m32 –mllvm –bcf ${FILE} –o "${FILE_NO_EXT##*/}-bcf.bin"

Tigress

- Obfuscates a given C source code
- Maintained by research group at University of Arizona, lead by Christian Collberg
- Advanced obfuscation transformations for which an analysis method has not yet been established is also applicable

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

Tigress

- Options (tigress --options)
  - Transform: Specify obfuscation transformation
    - Sub-Options of AddOpaque
      - AddOpaqueCount: A number of opaque predicates to be added
      - AddOpaqueKinds: Kinds of opaque predicates to be added
  - Functions: Specify function to be obfuscated
  - Environment: Specify architecture, OS, and compiler
  - out: Specify output source code name
  - o: Specify output binary name
- References
  - http://tigress.cs.arizona.edu/index.html

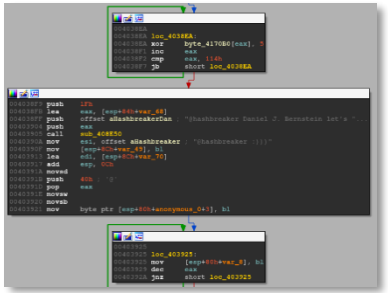
Tigress

- Running Examples
  - Encode Literals
    - tigress --Transform=EncodeLiterals --Functions=target_function --Environment=x86_64:Darwin:Clang:5.1 -m32 --out=${FILE_NO_EXT##*/}-encodeliteral.c ${FILE} -o ${FILE_NO_EXT##*/}-encodeliteral.bin
  - Encode Arithmetic
    - tigress --Transform=EncodeArithmetic --Functions=target_function --Environment=x86_64:Darwin:Clang:5.1 -m32 --out=${FILE_NO_EXT##*/}-encodearith.c ${FILE} -o ${FILE_NO_EXT##*/}-encodearith.bin
  - Opaque Predicate
    - tigress --Transform=InitOpaque --Functions=main --Transform=AddOpaque --Function=target_function --AddOpaqueCount=${NUM} –AddOpaqueKinds=true --Environment=x86_64:Darwin:Clang:5.1 -m32 --out=${FILE_NO_EXT##*/}-opaque.c ${FILE} -o ${FILE_NO_EXT##*/}-opaque.bin
  - Control Flow Flattening
    - tigress --Transform=Flatten --Functions=target_function --Environment=x86_64:Darwin:Clang:5.1 -m32 --out=${FILE_NO_EXT##*/}-flatten.c ${FILE} -o ${FILE_NO_EXT##*/}-flatten.bin
  - Virtualization Obfuscation
    - tigress --Transform=Virtualize --Functions=target_function --Environment=x86_64:Darwin:Clang:5.1 -m32 --out=${FILE_NO_EXT##*/}-virtualized.c ${FILE} -o ${FILE_NO_EXT##*/}-virtualized.bin
- Notes
  - Transform and Functions are stackable:
    - tigress --Transform=Flatten --Functions=target_function --Transform=Virtualize --Functions= …

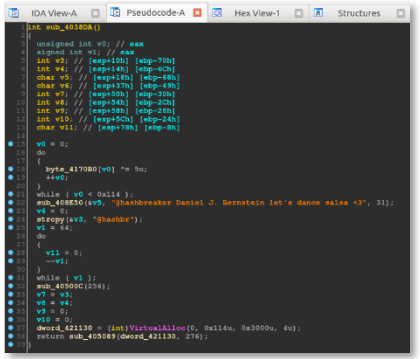
IDA Pro

- Interactive disassembler

Hexdump



Disassembly



Decompiled Code
(Unavailable on Freeware version)

Debugging

```
static CW(off,name,cmt) {
   auto x;
   x = [ 0x40, off ];
   MakeWord(x);
   MakeName(x,name);
   ...
```

IDC Scripting

```
from idc import *
from idaapi import *

def main():
   ea = ScreenEA()
   ...
```

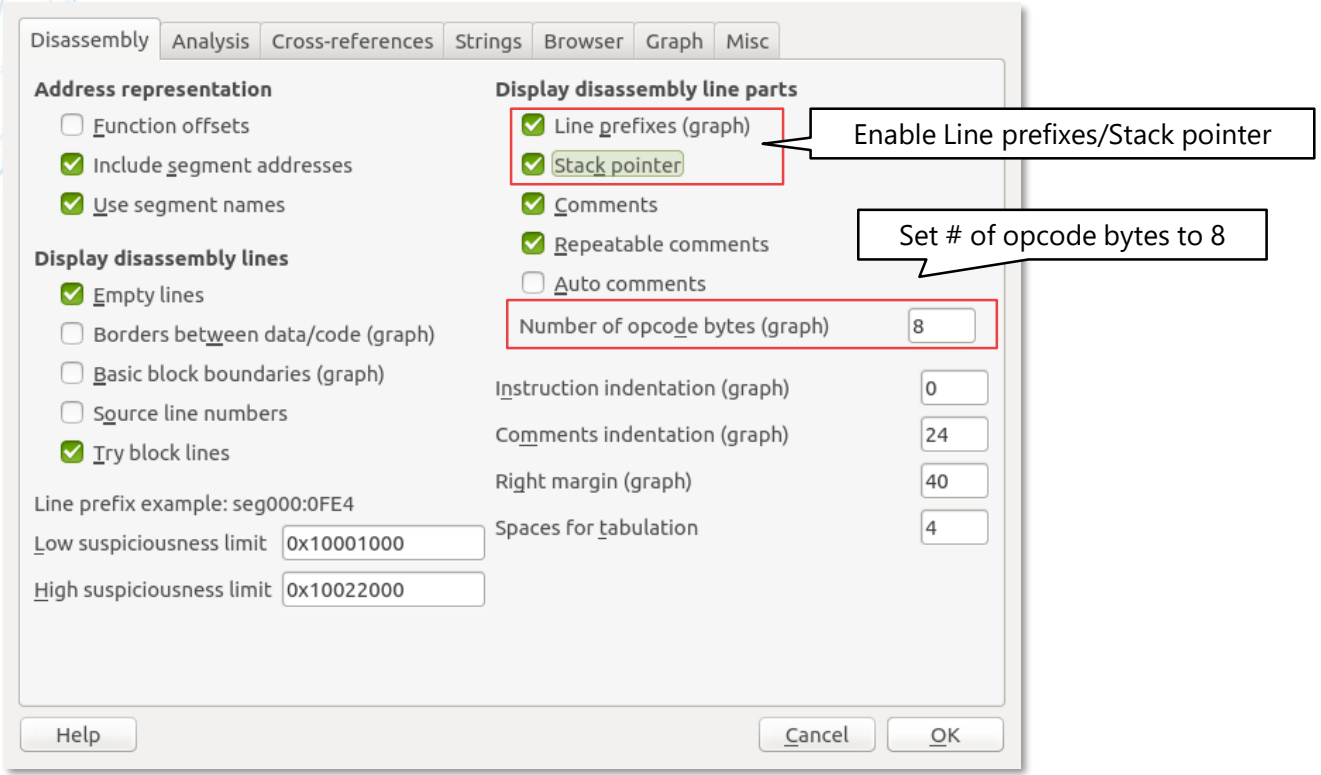
IDAPython Scripting
(Unavailable on Freeware version)

IDA Pro

- IDB Database
  - Store byte sequence, modified labels, comments, etc.
- Subviews (View → Open subviews)
  - Imports
  - Exports
  - Strings
  - Hex
  - Functions
  - Structures

IDA Pro Tips

- Options → General



IDA Pro Cheat Sheet

Shortcut Key	Description
X	Cross Reference (xref)
Enter	Jump to address
Esc	Jump to previous position
Space	Switch views
U	Un-define selected region
C	Interpret selected region as code
D	Interpret selected region as data
P	Interpret selected region as function
A	Interpret selected region as string
N	Rename function or variable
;	Add repeatable (xref-able) comment

- Reference
  - https://www.hex-rays.com/products/ida/support/freefiles/IDA_Pro_Shortcuts.pdf

Applicability of Obfuscation Techniques

- Not all functions can be obfuscated properly
  - The applicability depends on a program structure and a transformation

✓ – Success

	O-LLVM			Tigress				
	sub	bcf	fla	AddOpaque (w/ -q option)	EncodeLiterals	EncodeArithmetic	Virtualize	Flatten
test-add.c	✓	✓	*1	✓ (✓)	*3	✓	✓	✓
test-hello.c	*2	✓	*1	✓ (✓)	✓	*2	✓	✓
test-mod2.c	*2	✓	✓	✓ (*4)	*3	✓	✓	✓

*1 – # of Basic Blocks is not enough
*2 – No substitutable operations
*3 – No literals
*4 – No room for inserting opaque predicate as it only contains conditional branch

Deobfuscation Techniques

- Approach
  - Simplify: Transform code into readable form
  - Elimination: Remove redundant code
  - Dynamic Analysis: Avoid reading obfuscated code
- Technology Stack
  - Dataflow Analysis
  - Symbolic Execution
  - Equivalence Checking
  - Abstract Interpretation
  - Program Synthesis
  - Taint Analysis
  - …

Deobfuscation Techniques

(De-)Obfuscation Techniques

- Different techniques, common ideas:
  - Do useless things
    - Garbage/Dead Code Insertion
  - Change syntax
    - Instruction Substitution
    - Encode Literals
    - Encode Arithmetic
  - Change not only syntax but also semantics
    - Opaque Predicate
    - Virtualization Obfuscation
    - Control Flow Flattening

Dataflow Analysis
(Liveness Analysis)

Dataflow Analysis
(Reachable Definition Analysis)

Symbolic Execution
Equivalence Checking

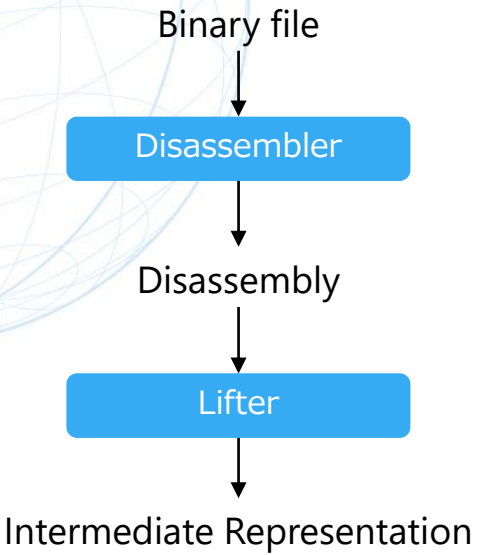
VMHunt
Program Synthesis

Graph Pattern Matching
Dynamic Symbolic Execution

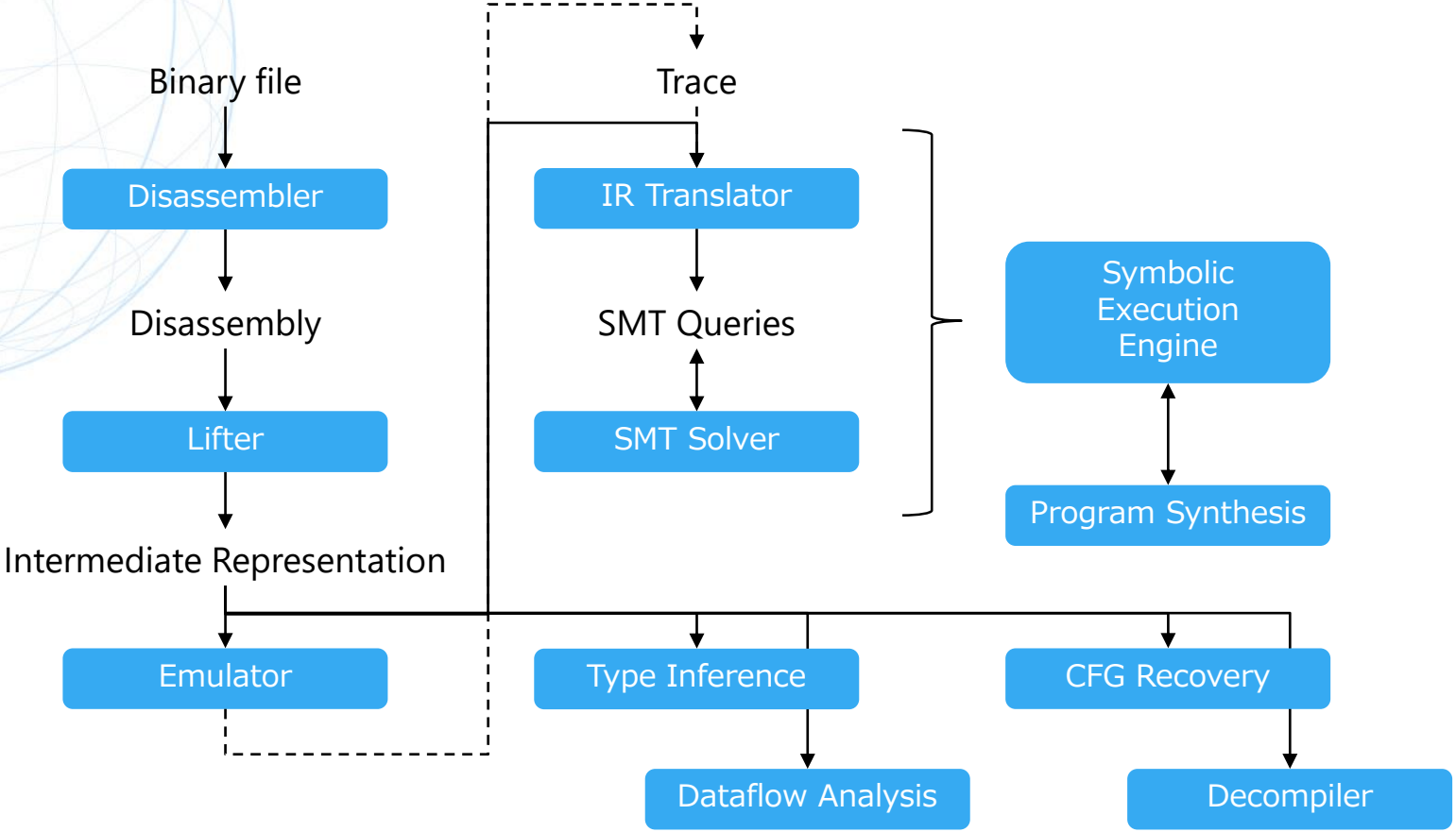
Binary Analysis Tools

- These techniques cannot be established without a modern binary analysis tools
  - IDA, Ghidra, radare2, Binary Ninja, angr, BINSEC, Triton, Miasm, McSema, etc.
- As well as a compiler, binary analysis tools typically consist of two major components:
  - Frontend
  - Backend

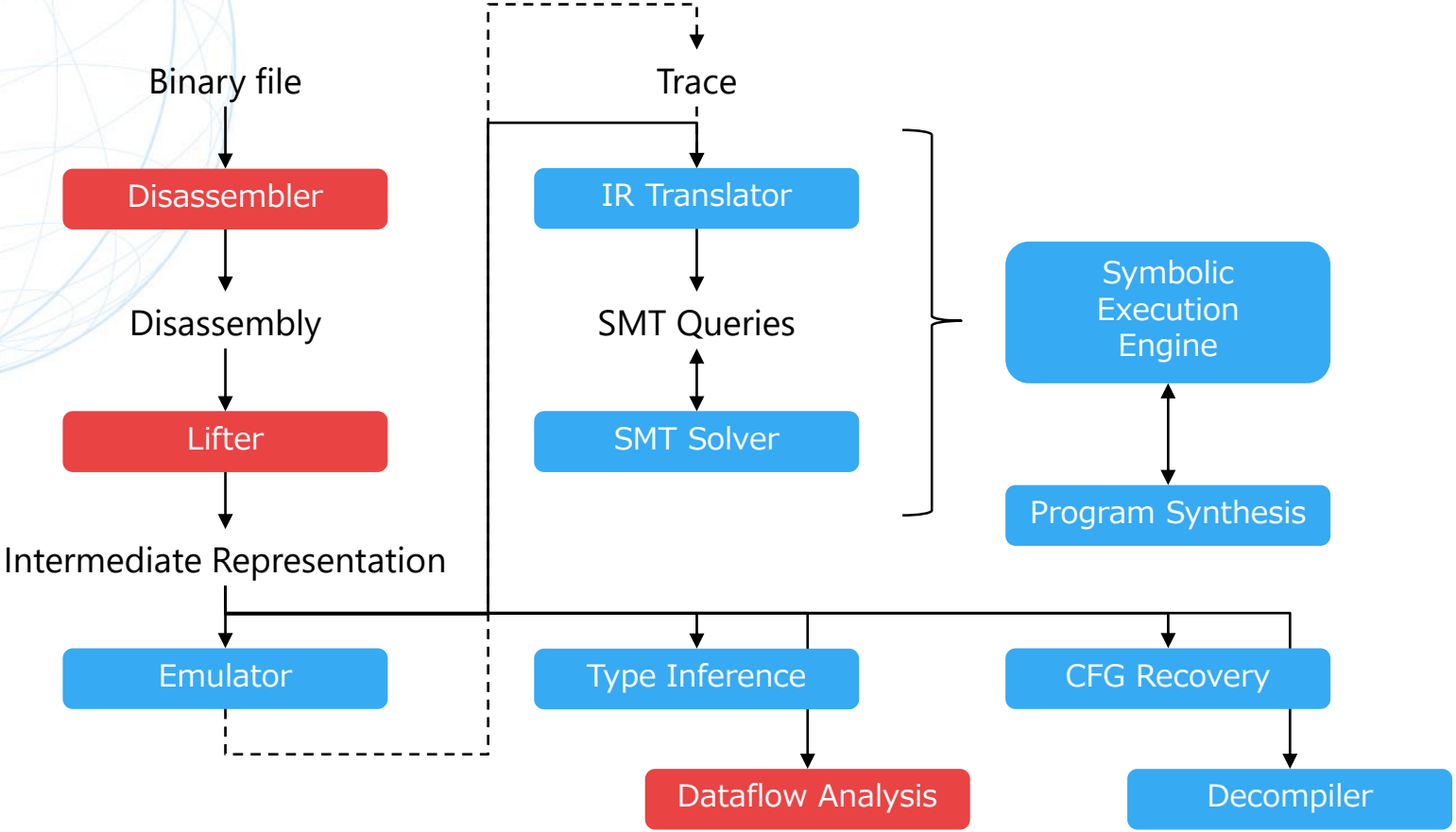
Binary Analysis Frontend



Binary Analysis Backend



Binary Analysis Backend



Intermediate Representation

- A glue between binary code and analysis methods
  - Compiler optimization, symbolic execution, program synthesis, etc.
- IR enables us to handle code from different architectures in a single interface
  - x86, x64, ARM, etc.
- Regardless of a compiler or a binary analysis tool, IR is typically a Static Single Assignment (SSA) form
  - Each variable is assigned exactly once; it is defined before it is used
  - This property facilitates optimization or transformation

```
reg_01 = 5                          reg_01₁ = 5
reg_02 = reg_01 − 3     ⟶          reg_02₁ = reg_01₁ − 3
reg_01 = reg_01 * 2                 reg_01₂ = reg_01₁ * 2
```

Intermediate Representation

- As with other languages, IR consists of:
  - Syntax: Which opcodes and operands can be combined
  - Operational Semantics: How operands are updated by each opcode

program	::=	*stmt**
stmt s	::=	*var* := *exp* | store(*exp*, *exp*) | goto *exp* | assert *exp* | if *exp* then goto *exp* else goto *exp*
exp e	::=	load(*exp*) | *exp* $\Diamond_b$ *exp* | $\Diamond_u$ *exp* | *var* | get_input(*src*) | *v*
$\Diamond_b$	::=	typical binary operators
$\Diamond_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

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

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

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad v' = \Diamond_u v}{\mu, \Delta \vdash \Diamond_u e \Downarrow v'}\ \textsc{Unop}$$

$$\frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad v' = v_1 \Diamond_b v_2}{\mu, \Delta \vdash e_1 \Diamond_b e_2 \Downarrow v'}\ \textsc{Binop}$$

Schwartz et al. All You Ever Wanted to Know About Dynamic Taint Analysis. Oakland'10.

Intermediate Representation

- Limitations
  - Flag registers, floating points, SIMD, etc. are difficult to model;
  - Thus IR in binary analysis tool would not be equivalent to the semantics of the original code, but only approximates it

Kim et al. Testing Intermediate Representations for Binary Analysis. ASE'17.

Intermediate Representation

TABLE I: Existing open-source tools[1] for binary analysis.

		Lifter Design				IR Characteristics						Architecture Support						
Tool	IR Name	Programming Language	Pure Parallelism	IR Optimization	AST Construction	Metadata Embedding	Explicit[5]	Self-Contained[5]	Hash-consed IR Support	x86 SIMD Support	Big Integer Splitting	x86	x86-64	ARMv7	ARMv8	Thumb	MIPS32	MIPS64
angr [40]	VEX[2]	C & Python	✗	✓	✓	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
BAP [13]	BIL	OCaml[4]	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
BINSEC [18]	DBA	OCaml	✗	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗
BinNavi [19]	REIL	Java	✓	✗	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
BitBlaze [41]	Vine	C & OCaml	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗
Insight [21]	Microcode[3]	C++	✓	✗	✓	✗	✓	✓	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗
Jakstab [30]	SSL	Java	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗
Miasm [15]	Miasm IR	C & Python	✗	✗	✓	✗	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✗
radare2 [4]	ESIL [1]	C	✓	✗	✗	✗	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
rev.ng [17]	LLVM	C++	✓	✗	✓	✗	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓
B2R2★	LowUIR	F#	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

[1] We intentionally omit MCSema [45] as it relies on IDA Pro for disassembling binaries.
[2] angr internally uses VEX IR, and it lifts binaries using a module called PyVEX [3], which is a Python wrapper of the VEX lifter [34] originally written in C.
[3] This should not be confused with IDA Pro's IR, which is also referred to as Microcode.
[4] The ocaml-multicore project [28] aims to support multicore parallelism, but it is not yet integrated into the upstream.
[5] We follow the definition of explicitness and self-containment in [29].
★ This is our work.

Jung et al. B2R2: Building an Efficient Front-End for Binary Analysis. BAR'19.

Miasm

- Binary analysis tool
  - Provides Python interface; much easier than OCaml
  - Supports multiple file formats and architectures
- User can lift binary code to Miasm IR and apply various analysis
  - Symbolic Execution
  - Concolic Execution
  - Program Slicing
  - Emulation (JIT)
  - Simplification

Miasm

- Strengths
  - More backward compatibility than Angr
  - Rich functionalities than Triton
- Weaknesses
  - The simplification for the value in the memory is poor
  - Once lifted to Miasm IR, it is difficult to back it to the x86 code
    - x86 → Miasm IR → LLVM IR → x86
  - Automatic analysis would not be scale to the entire binary

Miasm

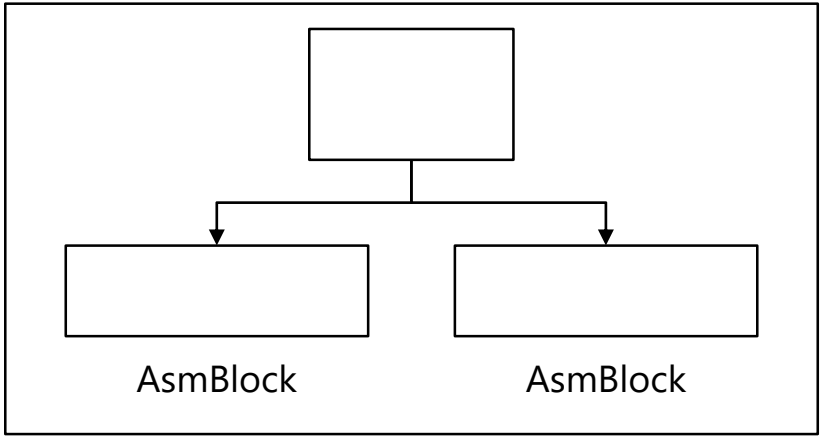
- Our aim is not about mastering Miasm itself
  - Other tools may be appropriate for some tasks
  - The point is to understand the principle of binary analysis explained so far
- Interface differences are not the essence
  - Binary analysis tools are basically designed to create instances of a target binary and analysis methods, and then perform analysis via the instance methods
    - Triton: Initialize TritonContext, allocate Instruction to it, and communicate with the solver from it
    - Angr: Initialize Project, generate CFG from it, and manage the symbolic state with SimulationManager
    - Miasm: Generate AsmCFG, convert it to IRCFG, and symbolically execute it via SymbolicExecutionEngine

Miasm

```
from miasm.analysis.binary import Container
from miasm.analysis.machine import Machine
from miasm.ir.symbexec import SymbolicExecutionEngine
...

# Initialize location database
loc_db = LocationDB()

# Get architecture
with open('target.bin', 'rb') as fstream:
    cont = Container.from_stream(fstream, loc_db)
machine = Machine(cont.arch)

# Get a "factory" for the detected architecture
mdis = machine.dis_engine(cont.bin_stream, loc_db=cont.loc_db)

# Get AsmCFG at the entry point
asmcfg = mdis.dis_multiblock(cont.entry_point)

# Get IRCFG from the AsmCFG
ir = machine.ir(cont.loc_db)
ircfg = ir.new_ircfg_from_asmcfg(asmcfg)

# Add name to the offset
cont.loc_db.add_location(offset=cont.entry_point, name='entrypoint')

# Initialize symbolic execution engine
sb = SymbolicExecutionEngine(ir ...)
```

Miasm Disassembly

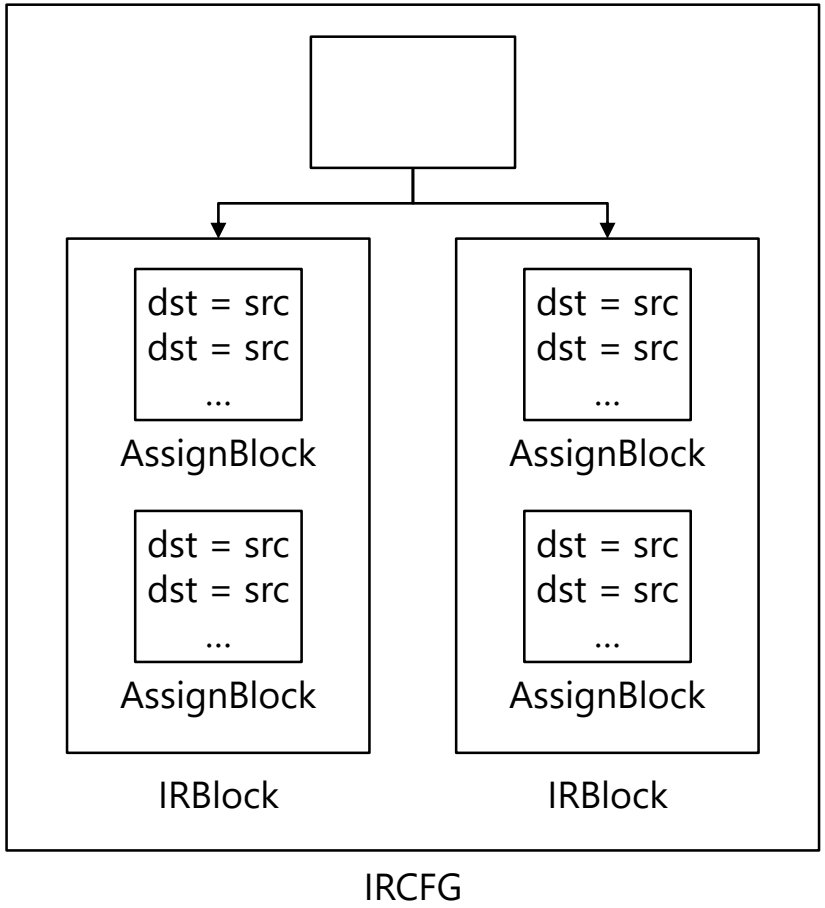
- AsmCFG
  - Control flow graph contains AsmBlocks
- AsmBlock
  - Basic block



AsmCFG

Miasm IR

- IRCFG
  - Control flow graph contains IRBlocks
- IRBlock
  - Contains AssignBlocks
- AssignBlock
  - Consists of assignments: dst = src
  - SSA from



Miasm IR

Element	Human Form
ExperInt	0x18
ExperId	EAX
ExprLoc	loc_17
ExprCond	A? B : C
ExprMem	@16[ESI]
ExprOp	A + B
ExprSlice	AH = EAX[8 : 16]
ExprCompose	AX = AH.AL
ExprAff	A = B

mov eax, ebx

```
ExprAff(ExprId("EAX", 32), ExprId("EBX", 32))
```

push eax

```
esp = esp - 0x4
@32[esp - 0x4] = eax
```

cmp eax, ebx

```
zf = (EAX - EBX)?0:1
cf = (((EAX ^ EBX) ^ (EAX - EBX)) ^ ...
of = ...
```

Desclaux and Mougey. Miasm: Reverse Engineering Framework. Black Hat USA'18.

(De-)Obfuscation Techniques

- Different techniques, common ideas:
  - Do useless things
    - Garbage/Dead Code Insertion
  - Change syntax
    - Instruction Substitution
    - Encode Literals
    - Encode Arithmetic
  - Change not only syntax but also semantics
    - Opaque Predicate
    - Virtualization Obfuscation
    - Control Flow Flattening

Dataflow Analysis
(Liveness Analysis)

Dataflow Analysis
(Reachable Definition Analysis)

Symbolic Execution
Equivalence Checking

VMHunt
Program Synthesis

Graph Pattern Matching
Dynamic Symbolic Execution

Dataflow Analysis

- Reachable Definition Analysis
  - Analyze where the value of each variable x was defined when a certain point p in the program was reached
  - *Forward* dataflow analysis
  - Application:
    - Constant propagation/folding
    - Transform expressions
- Liveness Analysis
  - Analyze whether the value x in the program point p may be used when following the edge starting from p in the flow graph with respect to x
  - *Backward* dataflow analysis
  - Application:
    - Dead code elimination

Dataflow Analysis

- Both reachable definition analysis and liveness analysis are IR optimization techniques used by a compiler backend, and are also useful for binary analysis
- Behind Insights
  - Obfuscation
    - Opposite of Optimization
  - Compiler
    - Generate, analyze, and optimize IR
  - Binary Analysis Tool
    - Generate, analyze, and optimize IR

Reachable Definition Analysis

Original code
(Obfuscated by InstructionSubstitution)

```
...
01. mov eax, [ebp+arg_0]
02. xor ecx, ecx
03. mov [ebp+var_8], eax
04. mov [ebp+var_C], 0Ch
05. mov [ebp+var_10], 38h
06. mov [ebp+var_14], 7Fh
07. mov eax, [ebp+var_C]
08. mov edx, [ebp+var_10]
09. sub eax, 2598A32Bh
10. add eax, edx
11. add eax, 2598A32Bh
12. mov edx, [ebp+var_14]
```

```
...
reach={01}
reach={01, 02}
reach={01, 02, 03}
reach={01, 02, 03, 04}
reach={01, 02, 03, 04, 05}
reach={01, 02, 03, 04, 05 ,06}
reach={02, 03, 04, 05 ,06, 07}
reach={02, 03, 04, 05 ,06, 07, 08}
reach={02, 03, 04, 05 ,06, 08, 09}
reach={02, 03, 04, 05 ,06, 08, 10}
reach={02, 03, 04, 05 ,06, 08, 11}
reach={02, 03, 04, 05 ,06, 11, 12}
```

Code after constant propagation
(Partially constant folding applied)

```
...
01. mov eax, [ebp+arg_0]
02. xor ecx, ecx
03. mov [ebp+var_8], eax
04. mov [ebp+var_C], 0Ch
05. mov [ebp+var_10], 38h
06. mov [ebp+var_14], 7Fh
07. mov eax, 0Ch
08. mov edx, 38h
09. eax = 0Ch - 2598A32Bh
10. eax = 0Ch - 2598A32Bh + 38h
11. eax = 0Ch - 2598A32Bh + 38h + 2598A32Bh
12. mov edx, 7Fh
```

38h+0Ch

Variables are replaced by existing constant definitions

*Analysis is performed after IR lifting

Reachable Definition Analysis

NTT

Original code
(Obfuscated by InstructionSubstitution)

```
13. mov esi, ecx
14. sub esi, eax
15. mov eax, ecx
16. sub eax, edx
17. add esi, eax
18. mov eax, ecx
19. sub eax, esi
20. mov edx, [ebp+var_8]
21. sub ecx, edx
22. sub eax, ecx
23. mov [ebp+var_18], eax
24. mov eax, [ebp+var_18]
...
```

```
reach={02, 03, 04, 05 ,06, 11, 12, 13}
reach={02, 03, 04, 05 ,06, 11, 12, 14}
reach={02, 03, 04, 05 ,06, 12, 14, 15}
reach={02, 03, 04, 05 ,06, 12, 14, 16}
reach={02, 03, 04, 05 ,06, 12, 16, 17}
reach={02, 03, 04, 05 ,06, 12, 17, 18}
reach={02, 03, 04, 05 ,06, 12, 17, 19}
reach={02, 03, 04, 05 ,06, 17, 19, 20}
reach={03, 04, 05 ,06, 17, 19, 20, 21}
reach={03, 04, 05 ,06, 17, 20, 21, 22}
```

Code after constant propagation
(Partially constant folding applied)

```
13. mov esi, 0h
14. esi = 0h - (38h + 0Ch)
15. mov eax, 0h
16. eax = 0h - 7Fh
17. esi = 0h – (38h+0Ch) + 0h – 7F
18. mov eax, 0h
19. eax = 0h – (-(38h+0Ch+7Fh))
20. mov edx, arg_0
21. ecx = 0h – arg_0
22. eax = 38h+0Ch+7Fh-(-arg_0)
23. mov [ebp+var_18], eax
24. mov eax, [ebp+var_18]
...
```

-(38h+0Ch+7Fh)

38h+0Ch+7Fh

38h+0Ch+7Fh+arg0

*Analysis is performed after IR lifting

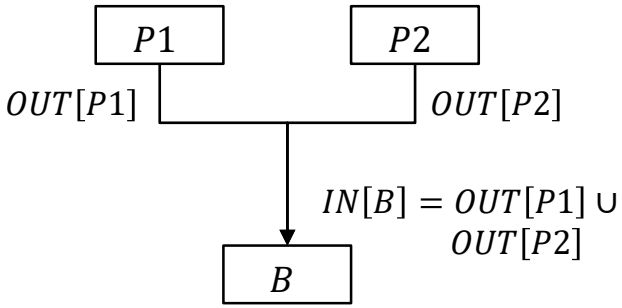
Reachable Definition Analysis

Notation	Description
B	Basic Block
$gen[B]$	Set of definitions generated in B and arriving at the end of B
$kill[B]$	Set of definitions killed by B
$IN[B]$	Set of definitions arriving at the start of B
$OUT[B]$	Set of definitions arriving at the end of B

Dataflow Equations:

$$IN[B] = \bigcup_{P \in Predecessors(B)} OUT[P]$$

$$OUT[B] = gen[B] \cup (IN[B] - kill[B])$$



Liveness Analysis

Variable assignments that are defined but not used can be removed

```
...
mov edx, 0xdeadc00d
mov eax, [ebp+arg_4]
mov ecx, [ebp+arg_0]
mov edx, 5
mov [ebp+var_4], ecx
mov [ebp+var_8], eax
mov eax, [ebp+var_4]
add eax, [ebp+var_8]
mov ecx, [ebp+var_8]
mov ecx, 0
mov [ebp+var_10], edx
mov edx, ecx
mov ecx, [ebp+var_10]
div ecx
mov [ebp+var_C], eax
...
```

```
...
def={edx}
def={eax}, use={[ebp+arg_4]}
def={ecx}, use={[ebp+arg_0]}
def={edx}
def={[ebp+var_4]}, use={ecx}
def={[ebp+var_8]}, use={eax}
def={eax}, use={[ebp+var_4]}
def={eax}, use={eax, [ebp+var_8]}
def={ecx} use={[ebp+var_8]}
def={ecx}
def={[ebp+var_10]}, use ={edx}
def={edx} use={ecx}
def={ecx}, use={[ebp+var_10]}
def={edx, eax}, use={edx, eax, ecx}
def={[ebp+var_C]} use={eax}
...
```

```
...
live={[ebp+arg_0], [ebp+arg_4]}
live={[ebp+arg_0], [ebp+arg_4]}
live={eax, [ebp+arg_0]}
live={eax, ecx}
live={edx, eax, ecx}
live={edx, [ebp+var_4], eax}
live={edx, [ebp+var_8], [ebp+var_4]}
live={edx, [ebp+var_8], eax}
live={eax, edx, [ebp+var_8]}
live={eax, edx}
live={eax, ecx, edx}
live={eax, [ebp+var_10], ecx}
live={edx, eax, [ebp+var_10]}
live={edx, eax, ecx}
live={eax}
```

live = A variable used before being redefined

$live_n = use_n + live_{n+1} - def_n$

*Analysis is performed after IR lifting

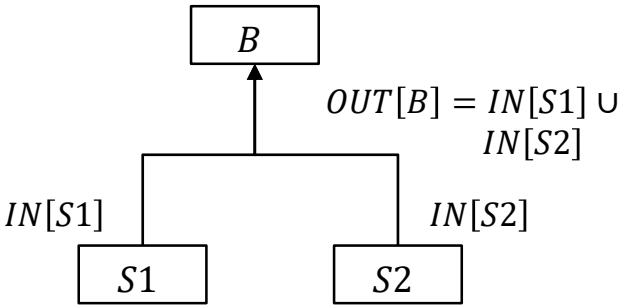
Liveness Analysis

Notation	Description
B	Basic Block
$def[B]$	Set of variables defined in B
$use[B]$	Set of variables used in B
$IN[B]$	Set of variables arriving at the start of B
$OUT[B]$	Set of variables arriving at the end of B

Dataflow Equations:

$$OUT[B] = \bigcup_{S \in Successors(B)} IN[S]$$

$$IN[B] = use[B] \cup (OUT[B] - def[B])$$



Hands-On 2: Deobfuscation via Optimization

- Duration: 30min
- Objective: Deobfuscate code via Miasm's dataflow analysis
- Step 1: Launch Jupyter Notebook
  - jupyter notebook
- Step 2: hands-on2/deadcode_removal.ipynb
  - Which line is the dead code within the code shown on the right?
  - Execute cells and confirm its effect
    - What's the Python method performs the removal?
- Step3: hands-on2/optimizer.ipynb
  - Optimize a binary obfuscated by O-LLVM –sub option
    - Using test-*-sub.bin generated at Hands-On 1
    - What's the Python method performs the constant propagation/folding?

```
0 main:
1 PUSH EBP
2 MOV EBP, ESP
3 MOV ECX, 0x23
4 MOV ECX, 0x4
5 MOV EAX, ECX
6 POP EBP
7 RET
```

Hands-On 2: Deobfuscation via Optimization

- Step 4: hands-on2/deadcode_unremoval.ipynb
  - Tweak the code shown on the right to prevent dead code removal
  - Constraint:
    - Do not change the output
  - Hint:
    - Branch instruction
  - Discussion
    - Discuss with your neighbors how to do it

```
0 main:
1 PUSH EBP
2 MOV EBP, ESP
3 MOV ECX, 0x23
4 MOV ECX, 0x4
5 MOV EAX, ECX
6 POP EBP
7 RET
```

*Dead code removal is done on the IR, while JIT execution is done on the original code;
The dead code would not disappear when viewed in JIT, yet this is normal behavior

Hands-On 2: Deobfuscation via Optimization

```
def do_dead_removal(self, ircfg):
    """
    Remove useless assignments.
    This function is used to analyse relation of a * complete function *
    This means the blocks under study represent a solid full function graph.
    Source : Kennedy, K. (1979). A survey of data flow analysis techniques.
    IBM Thomas J. Watson Research Division, page 43
    @ircfg: IntermediateRepresentation instance
    """

    modified = False
    reaching_defs = ReachingDefinitions(ircfg)
    defuse = DiGraphDefUse(reaching_defs, deref_mem=True)
    useful = self.get_useful_assignments(ircfg, defuse, reaching_defs)
    useful = set(useful)
    for block in list(viewvalues(ircfg.blocks)):
        irs = []
        for idx, assignblk in enumerate(block):
            new_assignblk = dict(assignblk)
            for lval in assignblk:
                if AssignblkNode(block.loc_key, idx, lval) not in useful:
                    del new_assignblk[lval]
                    modified = True
            irs.append(AssignBlock(new_assignblk, assignblk.instr))
        ircfg.blocks[block.loc_key] = IRBlock(block.loc_db, block.loc_key, irs)
    return modified
```

Dataflow Analysis

- Limitations
  - Reachable definition analysis/liveness analysis are conservative methods
    - Preserve program semantics
    - Assume passing the entire path of a program
  - Flow-sensitive, path-insensitive
    - Take care of the order of instructions
    - Do not take care of the conditional branches
  - How to interfere dataflow analysis?
    - Insert an opaque predicate and then create a dependency from it
    - Dataflow analysis does not take care of whether the path is actually executed
      - Junk code insertion would also work

Dataflow Analysis

- Notes
  - We used the methods of Miasm this time
  - Another possible approach is to convert IR to LLVM IR and apply LLVM's optimization

Garba and Favaro. SATURN - Software Deobfuscation Framework Based On LLVM. SPRO'19.

(De-)Obfuscation Techniques

- Different techniques, common ideas:
  - Do useless things
    - Garbage/Dead Code Insertion
  - Change syntax
    - Instruction Substitution
    - Encode Literals
    - Encode Arithmetic
  - Change not only syntax but also semantics
    - Opaque Predicate
    - Virtualization Obfuscation
    - Control Flow Flattening

Dataflow Analysis
(Liveness Analysis)

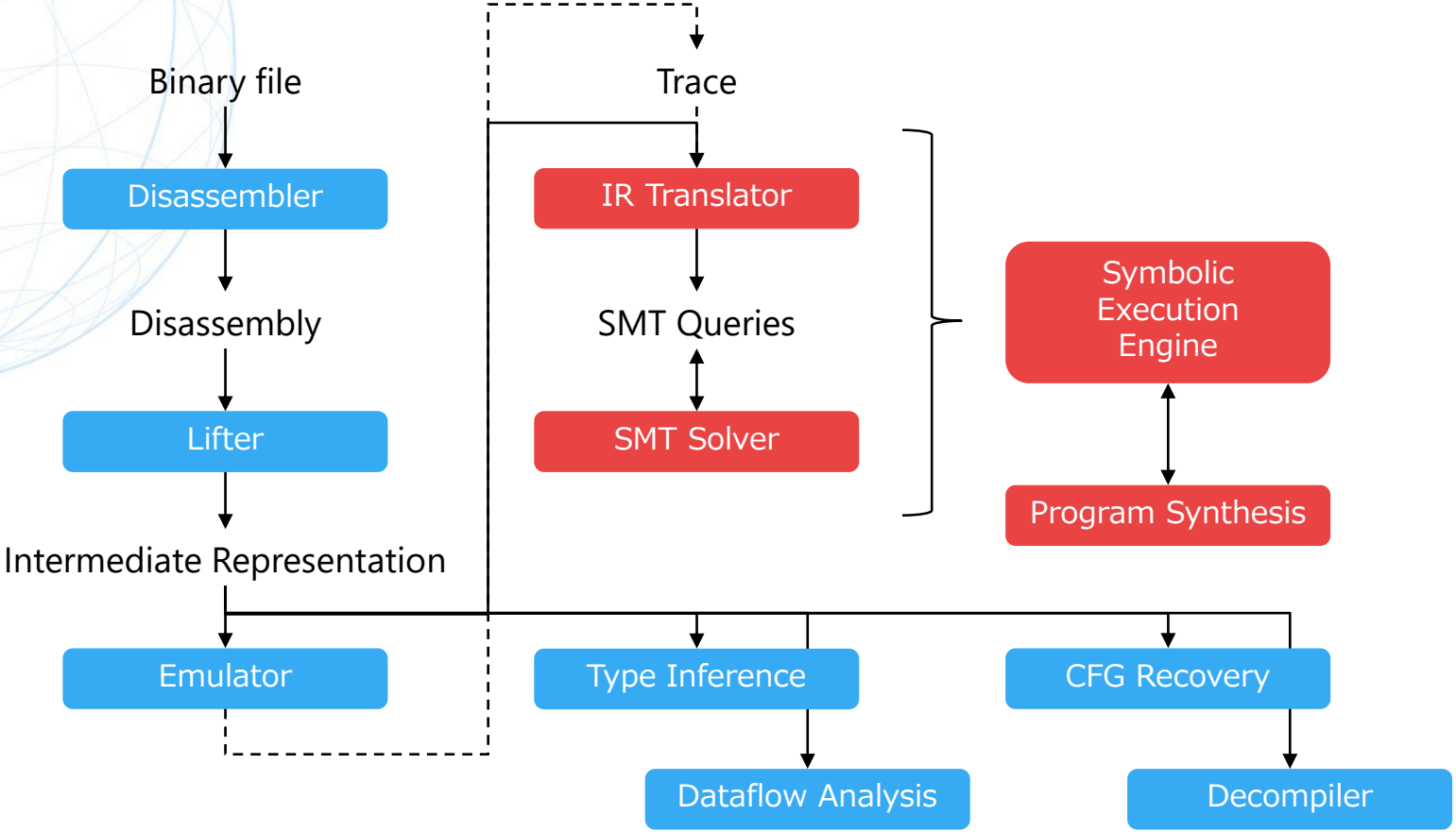
Dataflow Analysis
(Reachable Definition Analysis)

Symbolic Execution
Equivalence Checking

VMHunt
Program Synthesis

Graph Pattern Matching
Dynamic Symbolic Execution

Binary Analysis Backend



SAT Solver

- SAT: Satisfiability Problem
  - Propositional logic

$$(malicous \vee benign) \wedge (\neg malicous \vee benign) \wedge (\neg malicous \vee \neg benign)$$

$\longrightarrow$ 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())
```

Search for variable assignments that satisfy a given constraint

Clarke et al. Handbook of Model Checking. 2018.

SMT Solver = SAT Solver + Theories

- SMT: Satisfiability Modulo Theories
  - First-order predicate logic

$$(malicous \vee benign) \wedge (\neg malicous \vee benign) \wedge (\neg malicous \vee \neg benign) \wedge x * x - x = 2$$

⟶ SATisfiable

Theories
- EUF
- Arithmetic
- Array
- BitVector, etc.

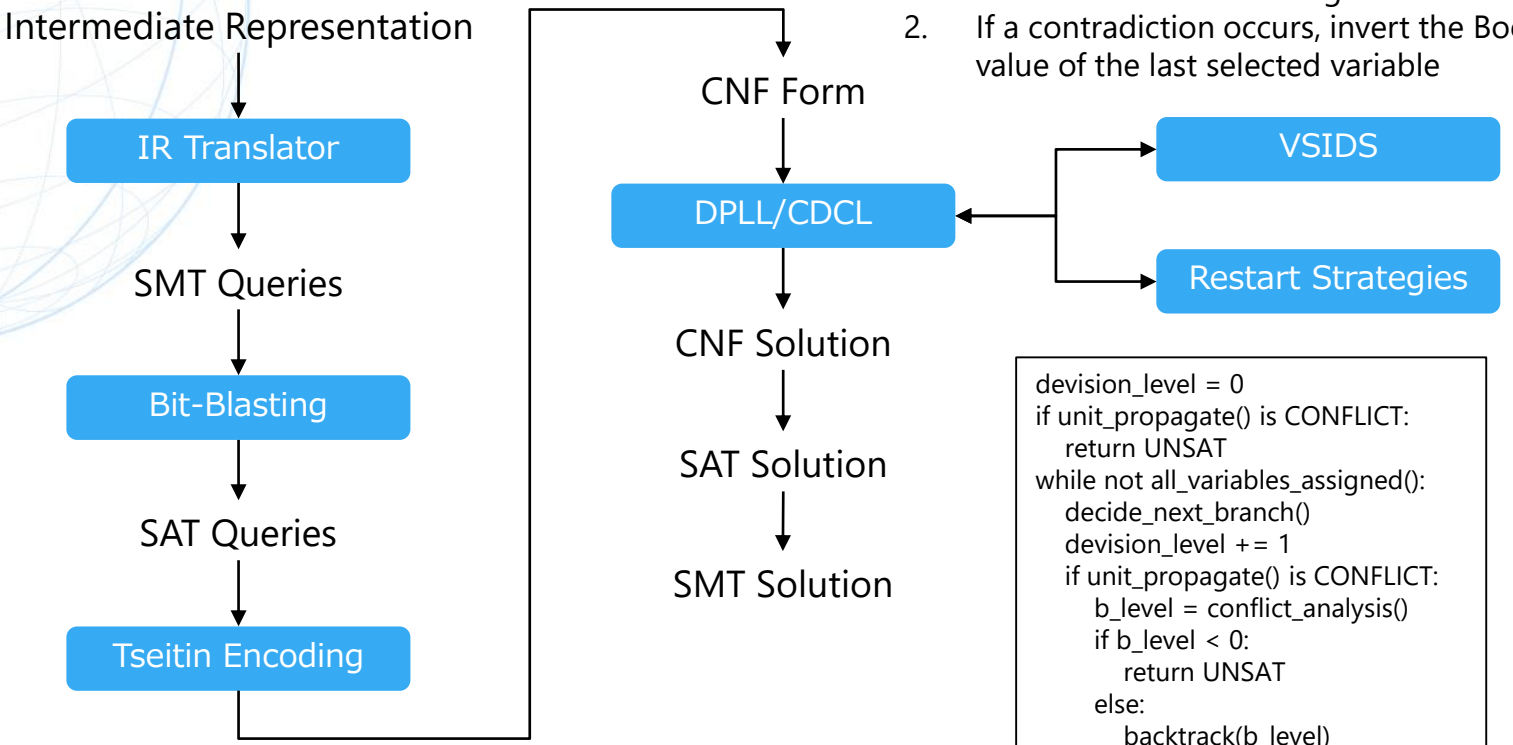
```
from z3 import *
malicious, benign = Bools('malicious
                           benign')
x, y = Int('x ')
s = Solver()
s.add(Or(malicious, benign),
   Or(Not(malicious), benign),
   Or(Not(malicious), Not(benign)),
   And(x * x – x == 2))

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

Clarke et al. Handbook of Model Checking. 2018.

SMT Solver

- Can handle various types and operators in addition to propositional logic (SAT)
- Treat variables as BitVectors in binary analysis

SMT Solver in Binary Analysis



1. Select a variable and assign true or false
2. If a contradiction occurs, invert the Boolean value of the last selected variable

```
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
```

SMT Solver in Binary Analysis

- Applications
  - Symbolic Execution
  - Equivalence Checking
  - Program Synthesis

Symbolic Execution

- A method for test case generation, proposed by J. C. King in 1976
  - What input values are needed to satisfy the conditions s at a program point p?
- How it works
  - Execute a program sequentially while treating input values as symbols that represent all possible values
  - Add constraints on symbols
    - Path Constraint: Constraints to execute a path
    - Symbolic Store: Updated symbol information
  - When the point is reached, solve the constraints with SMT solver and get a concrete input value
    - Need to convert IR to SMT solver-acceptable expressions

Symbolic Execution

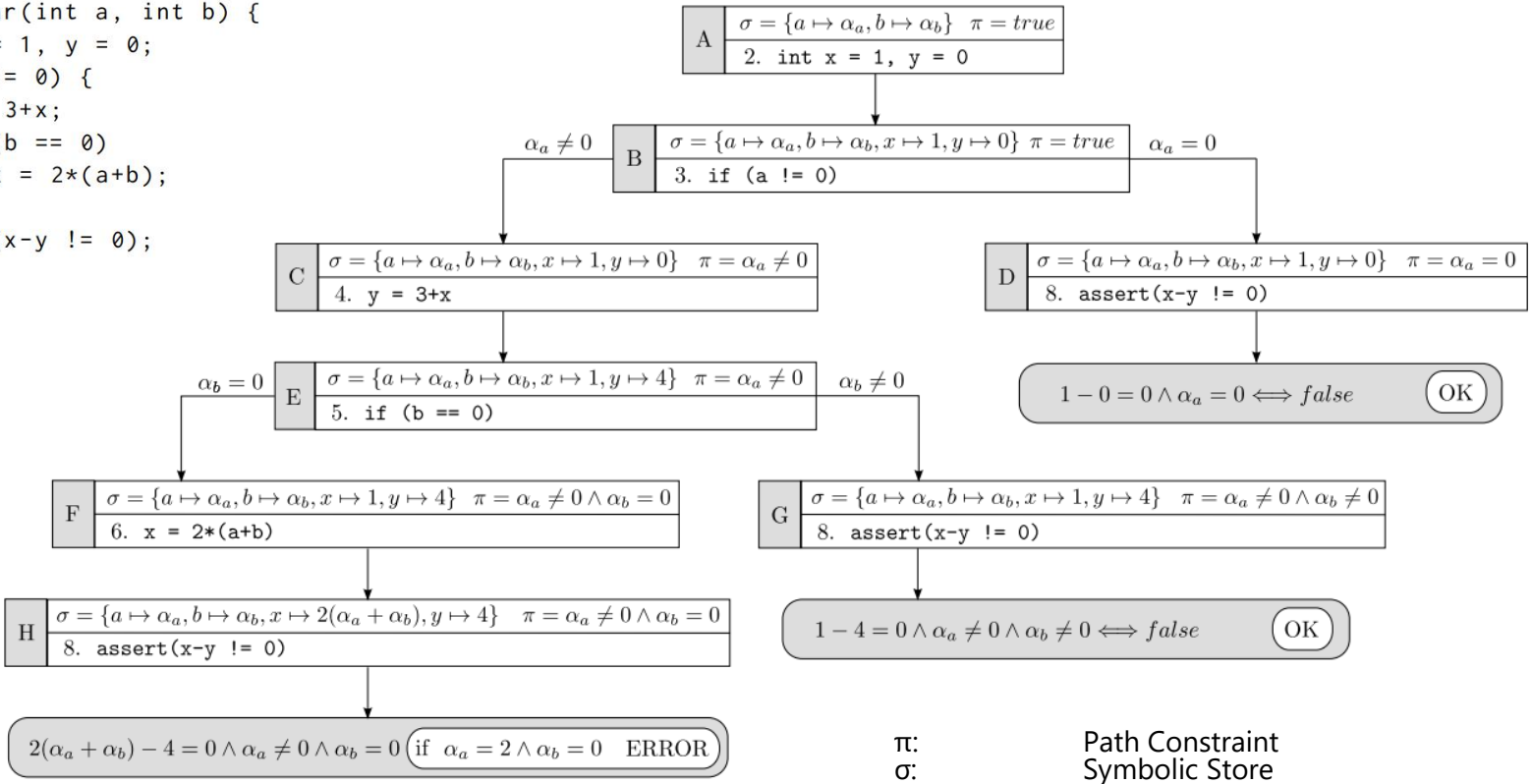
```
1.  void foobar(int a, int b) {
2.      int x = 1, y = 0;
3.      if (a != 0) {
4.          y = 3+x;
5.          if (b == 0)
6.              x = 2*(a+b);
7.      }
8.      assert(x-y != 0);
9.  }
```

Fig. 1. Warm-up example: which values of a and b make the assert fail?

Baldori et al. A Survey of Symbolic Execution Techniques. CSUR'18.

Symbolic Execution

```
1.  void foobar(int a, int b) {
2.     int x = 1, y = 0;
3.     if (a != 0) {
4.        y = 3+x;
5.        if (b == 0)
6.           x = 2*(a+b);
7.     }
8.     assert(x-y != 0);
9.  }
```



π:	Path Constraint
σ:	Symbolic Store
a→α:	Substitute α for variable a

Baldori et al. A Survey of Symbolic Execution Techniques. CSUR'18.

Symbolic Execution

- Use case
  - Extract conditions when a malware would be activated
  - Extract conditions when a vulnerability would be triggered (Automatic Exploit Generation)
  - …

Symbolic Execution with Miasm

- Path explorer
  - hands-on3/simple_explore.ipynb
  - Traverses multiple paths with symbolic execution and returns the final state

```
# Generate IRCFG from the AsmCFG
ircfg = ir_arch.new_ircfg_from_asmcfg(asmcfg)

# Initialize symbolic variables
symbols_init = {
    ExprMem(ExprId('ESP_init', 32), 32) : ExprInt(0xdeadbeef, 32)
}
for i, r in enumerate(all_regs_ids):
    symbols_init[r] = all_regs_ids_init[i]

final_states = []

# Explore symbolic states
explore(ir_arch,
            0,
            symbols_init,
            ircfg,
            lbl_stop=0xdeadbeef,
            final_states=final_states)
```

```
class FinalState:
    def __init__(self, result, sym, path_conds, path_history):
        self.result = result
        self.sb = sym
        self.path_conds = path_conds
        self.path_history = path_history


def explore(ir, start_addr, start_symbols,
            ircfg, cond_limit=30, uncond_limit=100,
            lbl_stop=None, final_states=[]):

    def codepath_walk(addr, symbols, conds, depth, final_states, path):
        sb = SymbolicExecutionEngine(ir, symbols)
        ....

     return codepath_walk(start_addr, start_symbols, [], 0, final_states, [])
```

Symbolic Execution with Miasm

- Path explorer

```
def codepath_walk(addr, symbols, conds, depth, final_states, path):
    sb = SymbolicExecutionEngine(ir, symbols)

    for _ in range(uncond_limit):
        if isinstance(addr, ExprInt):
            if addr == lbl_stop:
                final_states.append(FinalState(True, sb, conds, path))
                return

        path.append(addr)

        pc = sb.run_block_at(ircfg, addr)

        if isinstance(pc, ExprCond): # If conditional branch
            # Calculate the condition to take true or false paths
            cond_true  = {pc.cond: ExprInt(1, 32)}
            cond_false = {pc.cond: ExprInt(0, 32)}

            # The destination addr of the true or false paths
            addr_true  = expr_simp(
                    sb.eval_expr(pc.replace_expr(cond_true), {}))
            addr_false = expr_simp(
                    sb.eval_expr(pc.replace_expr(cond_false), {}))
            ...
            # Add the path conditions to reach this point
            conds_true = list(conds) + list(cond_true.items())
            conds_false = list(conds) + list(cond_false.items())

            # Recursive call for the true or false path
            # i.e., Duplicate the states
            codepath_walk(
                    addr_true, sb.symbols.copy(),
                    conds_true, depth + 1, final_states, list(path))

            codepath_walk(
                    addr_false, sb.symbols.copy(),
                    conds_false, depth + 1, final_states, list(path))

            return
        else:
            addr = expr_simp(sb.eval_expr(pc))

    final_states.append(FinalState(True, sb, conds, path))
    return
```

Symbolic Execution with Miasm

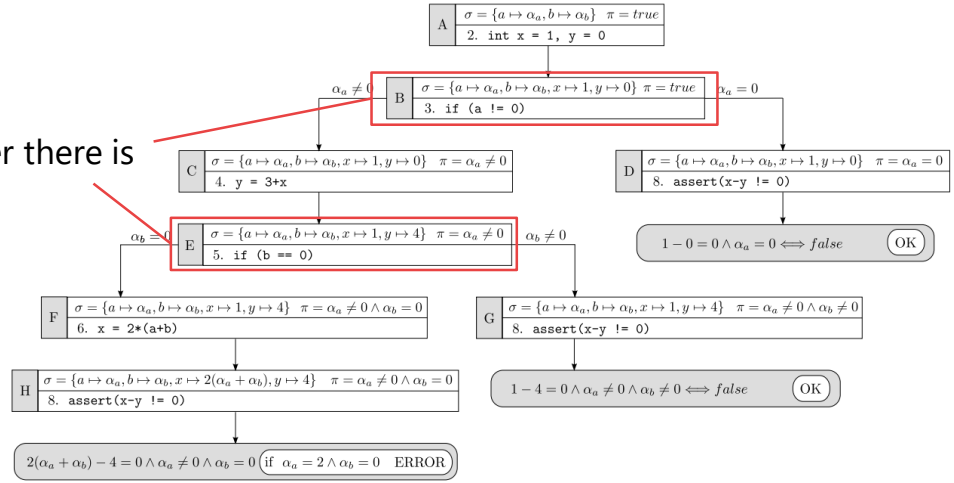
- Path explorer
  - How the function codepath_walk() works
    - Execute a block
    - Calculate constraints for True/False paths
    - Duplicate the symbolic state for each path
    - Invoke codepath_walk()
      - This means the script forks the state each time a branch is taken
  - Make sense? Let's move on to the opaque predicate detection with this

Opaque Predicate Detection

- How to detect opaque predicate with path exploration?
- Opaque predicate returns deterministic value regardless of an input value
  - Thus what we need do is to find the branch that determines True or False regardless of the input value

Implementation Plan:
Invoke the SMT solver at every branch and verify whether there is an input value that can take True Path or False Path;
if exists, the path is a normal;
if does not exist, the path should be opaque predicate

Q: Is it appropriate to do a feasibility check in the final state?



Ming et al. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. CCS'15.

Hands-On 3: Opaque Predicate Detection

- Duration: 30min
- Objective: Let's implement the detection method described above
- Step 1: hands-on3/simple_explore_smt.ipynb
  - Implement the method
  - Assert would fail by default
- Step 2: hands-on3/opfind.ipynb
  - Import the explorer from the simple_explore_smt.ipynb and complete the opfind.ipynb
  - Detect opaque predicate within O-LLVM-obfuscated code
    - Using test-*-bcf.bin generated at Hands-On 1
    - Open the binary with IDA and execute IDC file generated by opfind
      - It colors opaque predicates

Hands-On 3: Opaque Predicate Detection

- Step 3: APT malware analysis
  - Detect opaque predicate within X-Tunnel*1 malware via opfind.ipynb
    - Zip password: infected
    - Target function: 0x405710
  - Generate IDC and colorize IDA view
  - Detect opaque predicates in other functions
- Step 4 (Optional): More APT malware analysis
  - Detect opaque predicate within ANEL*2 malware via opfind.ipynb
    - Zip password: infected
  - …

*1 MD5: ac3e087e43be67bdc674747c665b46c2
X-Tunnel is a malicious implant used by APT28 a.k.a. Fancy Bear or Sofacy
Bardin et al. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. Oakland'17.
*2 MD5: a79f59b1b17e8bfa3299e50a8af9cdaf
ANEL is a RAT used by APT10 a.k.a. MenuPass, Stone Panda, or Red Apollo
Haruyama. Defeating APT10 Compiler-Level Obfuscations. VB'19.

Symbolic Execution

- Limitations
  - A search algorithm is up to its use case
    - DFS, random path selection, coverage-guided search, etc.
  - A room for optimization
    - State memorization, function summary
  - Accurate implementation of memory model and instruction semantics is difficult

Baldori et al. A Survey of Symbolic Execution Techniques. CSUR'18.
Xu et al. Concolic Execution on Small-Size Binary Codes: Challenges and Empirical Study. DSN'17.

Symbolic Execution

- Limitations
  - Attacks generate constraints difficult to solve for SMT solver
    - Hash/Crypto functions
    - Nonlinear functions
    - Collatz conjecture
  - Path explosion
    - Loop/recursion
    - When to or how much use the concrete value obtained by actual execution?
  - Attacks aimed at path explosion
    - *Range Divider*
    - Input-dependent loop/recursion

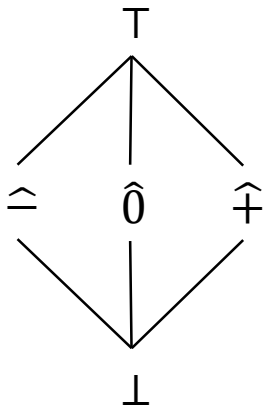
Banescu et al. Code Obfuscation Against Symbolic Execution Attacks. ACSAC'16.
Olivier et al. How to Kill Symbolic Deobfuscation for Free. ACSAC'19.
Wang et al. Linear Obfuscation to Combat Symbolic Execution. ESORICS'11.
Sharif et al. Impeding Malware Analysis Using Conditional Code Obfuscation. NDSS'08.
*Symbolic execution with a concrete execution/values together is called Dynamic Symbolic Execution (DSE) or Concolic Execution

Abstract Interpretation

- Another approach that allows us to detect opaque predicate
  - Originally a software verification technique
  - Analyze an abstracted program rather than a concrete program itself
  - Track only specified properties of variables used in the program
- How it works
  - Convert each instruction to an abstract semantics
  - Map variables to an abstract state according to the semantics
  - Simulate abstract semantics and update abstract state
  - Check abstract state at the desired time

Abstract Interpretation

- Example: Sign Analysis
  - Define the following abstract domain:
    - Neg: $\hat{-} := \{x \in \mathbb{Z} | x < 0\}$
    - Zero: $\hat{0} := \{0\} \subset \mathbb{Z}$
    - Pos: $\hat{+} := \{x \in \mathbb{Z} | x > 0\}$
    - Top (don't know): $\top := \mathbb{Z}$
    - Bottom (empty): $\bot := \emptyset$
  - Define the abstract semantics:
    - $\hat{+} + \hat{+} = \hat{+}$
    - $\hat{+} + \hat{0} = \hat{+}$
    - $\hat{+} + \hat{-} = \top$
    - …
  - Define functions:
    - Abstract function α: maps sets of concrete variables to the most precise value in the abstract domain
    - Concretization function γ: maps each abstract value to sets of concrete elements



A power set of the sets
(More precisely, a complete lattice)

Abstract Interpretation

- Example: Sign Analysis

Concrete		Abstract	
Semantics	**State**	**Semantics**	**State**
$x_1 = 1$	$\overset{x_1, x_2, x_3, x_4}{\langle 1, ?, ?, ? \rangle}$	$x_1 = \hat{+}$	$\overset{x_1, x_2, x_3, x_4}{\langle \hat{+}, \bot, \bot, \bot \rangle}$
$x_2 = -1$	$\langle 1, -1, ?, ? \rangle$	$x_2 = \hat{-}$	$\langle \hat{+}, \hat{-}, \bot, \bot \rangle$
$x_3 = x_1 * x_2$	$\langle 1, -1, -1, ? \rangle$	$x_3 = \hat{-} * \hat{+}$	$\langle \hat{+}, \hat{-}, \hat{-}, \bot \rangle$
$x_4 = x_1 + x_2$	$\langle 1, -1, -1, 0 \rangle$	$x_4 = \hat{+} + \hat{-}$	$\langle \hat{+}, \hat{-}, \hat{-}, \top \rangle$

Abstract Interpretation

- Galois Connection
  - Once abstracted by the function α, you cannot retrieve the concrete value;
  - Yet, you can analyze exact properties of variables on the abstract domain (Soundness)
- The abstract domain can be any form as long as the inclusion relation can be described as a power set
  - Sign
  - Type information
  - …

Abstract Interpretation

- The theory is profound; the paper seems to be obfuscated … 🤔

$$\text{For orders and posets, } \mathbb{O}[\![o]\!] \triangleq o,\ o \in \{\Rightarrow, \Leftrightarrow, \leq, \subseteq, =\},$$
$$\mathbb{O}[\![\dot{\leq}]\!] \triangleq \subseteq, \ldots, \mathbb{O}[\![\ddot{o}]\!] \triangleq ((\mathbb{O}[\![o]\!])), \text{ and } \mathscr{P}[\![\langle s,\ o\rangle]\!] \triangleq \mathcal{S}[\![s]\!] \circledast \mathbb{O}[\![o]\!].$$
$$\text{For } GCs,\ \mathcal{T}[\![\mathbb{1}[p]]\!] \triangleq \mathscr{P}[\![p]\!] \leftrightharpoons \mathscr{P}[\![p]\!],\ \mathcal{T}[\![\curvearrowright[s_{\mathbb{L}}, s_{\mathcal{M}}]]\!] \triangleq \mathbf{P}(\mathcal{S}[\![s_{\mathbb{L}}]\!] * \mathcal{S}[\![s_{\mathcal{M}}]\!]) \circledast \subseteq \leftrightharpoons \mathcal{S}[\![s_{\mathbb{L}}]\!] \nrightarrow \mathbf{P}\,\mathcal{S}[\![s_{\mathcal{M}}]\!] \circledast \dot{\subseteq},$$
$$\mathcal{T}[\![\cup[s]]\!] \triangleq \mathbf{P}\,(\mathbf{P}\,\mathcal{S}[\![s]\!]) \circledast \subseteq \leftrightharpoons \mathbf{P}\,\mathcal{S}[\![s]\!] \circledast \subseteq,\ \mathcal{T}[\![\neg[s]]\!] \triangleq \mathbf{P}\,\mathcal{S}[\![s]\!] \circledast \subseteq \leftrightharpoons \mathbf{P}\,\mathcal{S}[\![s]\!] \circledast \subseteq^{-1},$$
$$\mathcal{T}[\![\infty[s]]\!] \triangleq \mathbf{P}\,(\mathbf{seq}\,\mathcal{S}[\![s]\!]) \circledast \subseteq \leftrightharpoons \mathbf{P}\,\mathcal{S}[\![s]\!] \circledast \subseteq,\ \mathcal{T}[\![\rightsquigarrow[s_1, s_2]]\!] \triangleq$$
$$\mathbf{P}\,(\mathcal{S}[\![s_1]\!] * \mathcal{S}[\![s_2]\!]) \circledast \subseteq \leftrightharpoons \mathbf{P}\,\mathcal{S}[\![s_1]\!] \nrightarrow \mathbf{P}\,\mathcal{S}[\![s_2]\!] \circledast \dot{\subseteq},\ \mathcal{T}[\![\mapsto[s_1, s_2]]\!] \triangleq$$
$$\mathbf{P}\,(\mathcal{S}[\![s_1]\!] \nrightarrow \mathcal{S}[\![s_2]\!]) \circledast \subseteq \leftrightharpoons \mathbf{P}\,\mathcal{S}[\![s_1]\!] \nrightarrow \mathbf{P}\,\mathcal{S}[\![s_2]\!] \circledast \dot{\subseteq},\ \mathcal{T}[\![\times[s_1, s_2]]\!]$$
$$\triangleq \mathbf{P}\,(\mathcal{S}[\![s_1]\!] \nrightarrow \mathcal{S}[\![s_2]\!]) \circledast \subseteq \leftrightharpoons \mathcal{S}[\![s_1]\!] \nrightarrow \mathbf{P}\,\mathcal{S}[\![s_2]\!] \circledast \dot{\subseteq},\ \mathcal{T}[\![\mathsf{R}[g]]\!] \triangleq$$
$$\mathcal{T}[\![g]\!],\ \mathcal{T}[\![s \to g]\!] \triangleq \mathcal{S}[\![s]\!] \nrightarrow \mathcal{T}[\![g]\!], \ldots$$

Cousot and Cousot. A Galois Connection Calculus for Abstract Interpretation. POPL'14.

Abstract Interpretation

- Applications
  - Opaque Predicate Detection
    - How it works
      - For each block, find the branch where the condition is true for any value;
      - Find when the variable which triggers the *conditional* jump is ⊤ – it's *unconditional!*
    - The plugin for a Ghidra disassembler/decompiler by Rolf Rolles is publicly available:
      - https://www.msreverseengineering.com/blog/2019/4/17/an-abstract-interpretation-based-deobfuscation-plugin-for-ghidra
  - Value-Set Analysis (VSA)
    - A binary analysis method which uses type information as an abstract domain
    - Uses pointer analysis together
    - Can be used to buffer overflow detection
    - The tool with the most advanced VSA interface is angr

Preda et al. Opaque Predicates Detection by Abstract Interpretation. AMAST'06.
Shoshitaishvili et al. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. Oakland'16.

Abstract Interpretation

- Abstract Interpretation vs Symbolic Execution
  - Both approximate states

Nature of Program Analysis for Testing and Verification

	Sound	Complete
Under-approximate Testing, Dynamic Symbolic Execution, Dynamic Software Model Checking	All programs reported unsafe are actually unsafe *Only real bugs*	Successfully reports all unsafe programs *Finds all bugs*
Over-approximate Abstract Interpretation, Software Model Checking with Predicate Abstraction	All programs proven safe are actually safe *Only real proofs*	Successfully proves safety of all safe programs *Finds all proofs*

https://twitter.com/johannes_kinder/status/1105138480303218688

Abstract Interpretation

- Limitations
  - Accurate implementation of memory model and instruction semantics is difficult – same as symbolic execution
  - *Range Divider* – same as symbolic execution-based opaque predicate detection

Range Divider

- A method to cause path explosion in symbolic execution by adding extra branches
  - # of branches: k → # of states: 2^k
- Path constraint would not be UNSAT like opaque predicate
  - Execute a branch depends on an input value
  - Whichever a branch executed, the result is the same

Listing 2: *Range divider* with 2 branches

```
1 if (x > 42)
2   z = x + y + w
3 else
4   z = (((x ^ y) + ((x & y) << 1)) | w) +
5       (((x ^ y) + ((x & y) << 1)) & w);
```

Both paths would be executed!

Table 3: Impact of obfuscations on DSE

Transformation (#TO/#Samples)	Dataset #1		Dataset #2		
	Goal 1 3h TO	*Goal 2 1h TO*	*Goal 1 24h TO*	*Goal 2 3h TO*	*Goal 2 8h TO*
Virt	0/46	0/15	0/7	0/7	0/7
Virt ×2	1/46	0/15	0/7	0/7	0/7
Virt ×3	5/46	2/15	1/7	0/7	0/7
SPLIT ($k = 10$)	1/46	0/15	0/7	0/7	0/7
SPLIT ($k = 13$)	4/46	0/15	1/7	1/7	0/7
SPLIT ($k = 17$)	18/46	2/15	3/7	2/7	1/7
FOR ($k = 1$)	2/46	0/15	0/7	0/7	0/7
FOR ($k = 3$)	30/46	8/15	3/7	2/7	1/7
FOR ($k = 5$)	46/46	15/15	7/7	7/7	7/7

Timeout
3h for Dataset #1, 24h for Dataset #2

Banescu et al. Code Obfuscation Against Symbolic Execution Attacks. ACSAC'16.
Olivier et al. How to Kill Symbolic Deobfuscation for Free. ACSAC'19.

Range Divider

- Robust to symbolic execution/abstract interpretation
- a.k.a. 2-way Opaque Predicates, Code Clone
- Tigress can apply Range Divider via OpaqueKinds=question option

Banescu et al. Code Obfuscation Against Symbolic Execution Attacks. ACSAC'16.
Olivier et al. How to Kill Symbolic Deobfuscation for Free. ACSAC'19.

SMT Solver in Binary Analysis

- Applications
  - Symbolic Execution
  - Equivalence Checking
  - Program Synthesis

Equivalence Checking

- A method to determine if two given codes have the same behavior
  - Same code: Syntactically-equivalent
  - Different code, but same behavior: Semantically-equivalent
- How it works
  - Perform symbolic execution per basic block
    - All inputs (register, memory) are symbolized
  - Compare outputs
    - With SMT solver by generating a *counterexample*
  - If no counterexample, the blocks are equivalent

Implementation Plan:
Verify if there is a solution that negates "the two basic blocks are not equivalent".
If exists, the assumption is correct, i.e. not equivalent;
Otherwise, the assumption is incorrect, i.e. equivalent.

```
# Convert miasm IR to Z3 expression
cond_1 = Translator.to_language('z3')
                                .from_expr(v1)
cond_2 = Translator.to_language('z3')
                                .from_expr(v2)
solver = z3.Solver()

solver.add(???) # Hint: z3.Not() is negation

if solver.check() == sat:
        # not equivalent
        …
else: # UNSAT
        # equivalent!
        …
```

Gao et al. BinHunt: Automatically Finding Semantic Differences in Binary Programs. ICICS'08.
Shirazi et al. DoSE: Deobfuscation based on Semantic Equivalence. SSPREW'18.

Equivalence Checking

- a.k.a. (Semantic) Binary Diffing, Code Clone Detection
- Applications
  - Opaque Predicate Detection
  - N-days Vulnerability Detection

Hands-On 4: Range Divider Detection

- Duration: 30min
- Objective: Let's implement the equivalence checking method
- Step 1: hands-on4/eqcheck.ipynb
  - Implement the method
  - Of syntax_compare and semantic_compare, only the former is implemented

*1 MD5: 2aea3b217e6a3d08ef684594192cafc8
*2 MD5: 0d655ecb0b27564685114e1d2e598627
Vipasana is a ransomware; Asprox is a trojan

Hands-On 4: Range Divider Detection

- Step 2: Malware analysis
  - Detect range-divided functions within Vipasana*1 and Asprox*2 malware
    - Zip password: infected
    - Vipassana target function: 0x434DF0
    - Asprox target function: 0x100091AC
  - Detect other equivalent functions (including Range Divider and syntactically-equivalent functions)

*1 MD5: 2aea3b217e6a3d08ef684594192cafc8
*2 MD5: 0d655ecb0b27564685114e1d2e598627
Vipasana is a ransomware; Asprox is a trojan

Hands-On 4: Range Divider Detection

- Step 3: Tigress Range Divider
  - Test the script against Tigress-obfuscated code
    - Use OpaqueKinds=question option
  - The range divider detection method would not work, why?
  - Discussion
    - How can we improve the method?

Equivalence Checking

- Limitations
  - Is Basic Blocks-oriented comparison appropriate?
    - Chunk with in a block, 1 block, 2 blocks, … function, functions, …?
    - Path explosion occurs when the # of unit becomes large
    - BinSim: Defines system call, its arguments, and parts that depend on those arguments as a unit
    - The essence is how to determine a unit: Point of interests and range of its dependency
  - Definition of equivalence
    - We should regard units as equivalent even when they have different outputs not used later
      - We need to concretize this idea to combat Tigress's Range Divider

Ming et al. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking

(De-)Obfuscation Techniques

NTT

- Different techniques, common ideas:
  - Do useless things
    - Garbage/Dead Code Insertion
  - Change syntax
    - Instruction Substitution
    - Encode Literals
    - Encode Arithmetic
  - Change not only syntax but also semantics
    - Opaque Predicate
    - Virtualization Obfuscation
    - Control Flow Flattening

Dataflow Analysis
(Liveness Analysis)

Dataflow Analysis
(Reachable Definition Analysis)

Symbolic Execution
Equivalence Checking

VMHunt
Program Synthesis

Graph Pattern Matching
Dynamic Symbolic Execution

VM Deobfuscation

- VM analysis task consists of the three parts:
  - Locate:
    - Where is the VMEntry, handlers, and VMExit?
      - And how is the VM EIP updated?
  - Extract:
    - Dump the VM handlers
  - Simplify:
    - Recover the original semantics of each handler:
      - Arithmetic operations (add, sub, mul, div, ...) typically retrieve 2 values from a virtual register and write the result back to the register. Once you notice it's an arithmetic operator, all you need to do is analyze the differences
      - A conditional jump typically takes two values from a virtual register and writes the result back to the virtual instruction pointer

Xu et al. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. ACM CCS'18.
Blazytko et al. Syntia: Synthesizing the Semantics of Obfuscated Code. USENIX Security'17.
Salwan et al. Symbolic Deobfuscation: From Virtualized Code to Back to The Original. DIMVA'18.

VM Deobfuscation

- Several techniques come to the rescue:
  - Locate:
    - Virtualized Snippet Boundary Detection (VMHunt)
  - Extract:
    - Virtualized Kernel Extraction (VMHunt)
  - Simplify:
    - Writing an IDA Processor Module on your own
    - Symbolic Execution
      - With expression simplification
      - Including DSE and Multiple Granularity Symbolic Execution (VMHunt)
    - Program Synthesis
      - CEGIS, Syntia
    - Compiler Optimization

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

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

Xu et al. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. ACM CCS’18.
Blazytko et al. Syntia: Synthesizing the Semantics of Obfuscated Code. USENIX Security’17.
Salwan et al. Symbolic Deobfuscation: From Virtualized Code to Back to The Original. DIMVA’18.

VMHunt

- A method tailored for VM deobfuscation
- Based on dynamic analysis, some heuristics, and a variant of symbolic execution
- How it works
  - Locate context switch between the host and the VM
  - Extract the kernel from two operations:
    - The write to the VM context area
    - The write to the native stack area
  - Simplify the kernel code via symbolic execution

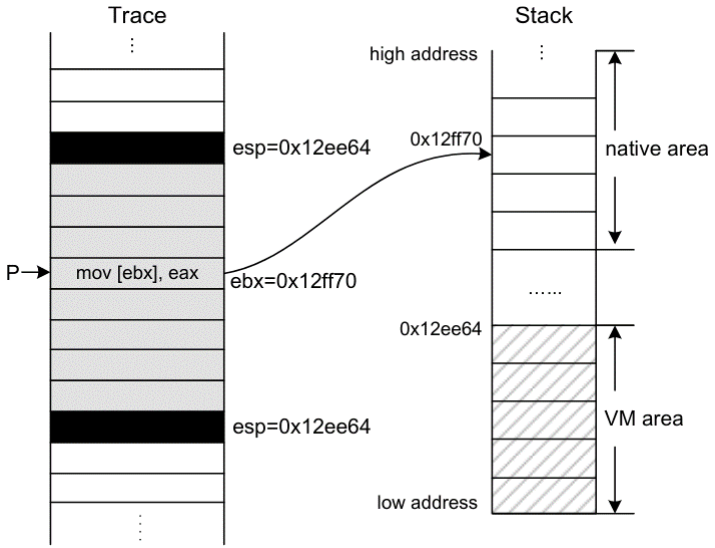
Xu et al. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. ACM CCS'18.

VMHunt

- Intuition: Context Switch

```
1  ...             // native program execution
2  push  edi       // context saving
3  push  esi
4  push  ebx
5  push  edx
6  push  ebp
7  push  ecx
8  push  eax
9  pushfd
10 jmp   0x1234
11 ...             // virtualized snippet begin
12 mov   ...
13 add   ...
14 xor   ...
15 ...             // virtualized snippet end
16 popfd
17 pop   eax       // context restoring
18 pop   ecx
19 pop   ebp
20 pop   edx
21 pop   ebx
22 pop   esi
23 pop   edi
24 jmp   0x8048123
25 ...             // continue native program
26 ...             // execution
```

Analyze only VM code with high accuracy



... while avoiding extra bytecode analysis

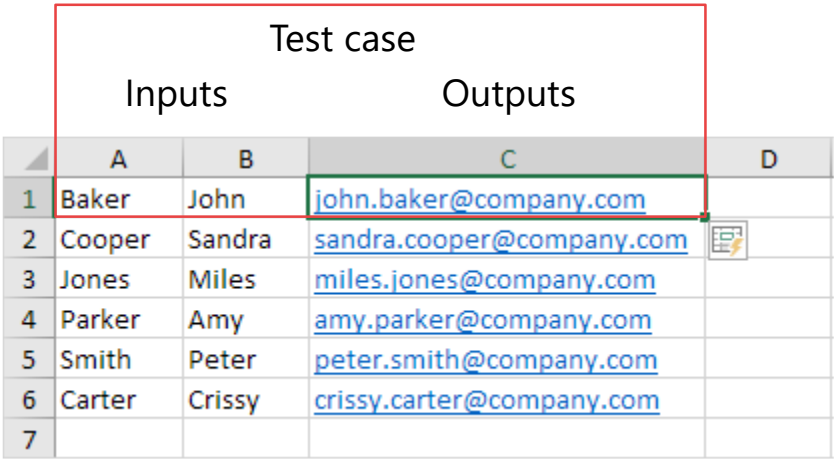
Xu et al. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. ACM CCS'18.

VMHunt

- Can be combined with other methods:
  - Optimization
  - Program Synthesis

Program Synthesis

- A method for generating program snippet from a test case
  - Infer the transformation (program) between input and output
  - Transform new input with an inferred program
- Example: Excel Flash Fill



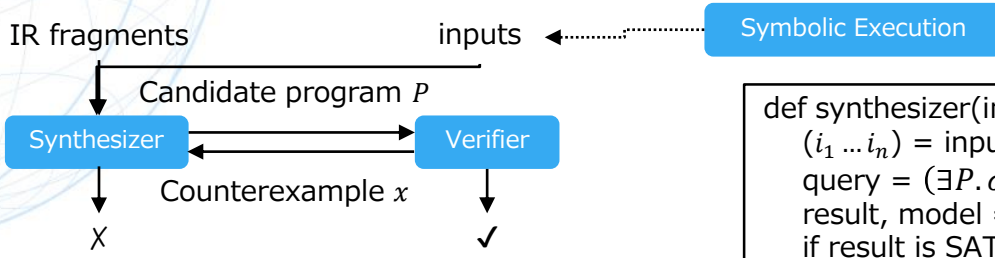
	A	B	C	D
1	Baker	John	john.baker@company.com	
2	Cooper	Sandra	sandra.cooper@company.com	
3	Jones	Miles	miles.jones@company.com	
4	Parker	Amy	amy.parker@company.com	
5	Smith	Peter	peter.smith@company.com	
6	Carter	Crissy	crissy.carter@company.com	
7				

Program Synthesis

- Formulated as a search problem
  - Input
    - Test case: Inputs, outputs
    - IR Fragments
  - Output
    - A combination of fragments which satisfies the test case
  - Algorithm
    - Enumerative Search (w/ Pruning)
    - SMT Solving; Counterexample-Guided Inductive Synthesis (CEGIS)
    - Metropolis-Hastings
    - Monte Carlo Tree Search (MCTS) – Syntia
    - Bayesian Net
    - …

Program Synthesis

- CEGIS
  - Reduce search space by SMT solving




```
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)
```

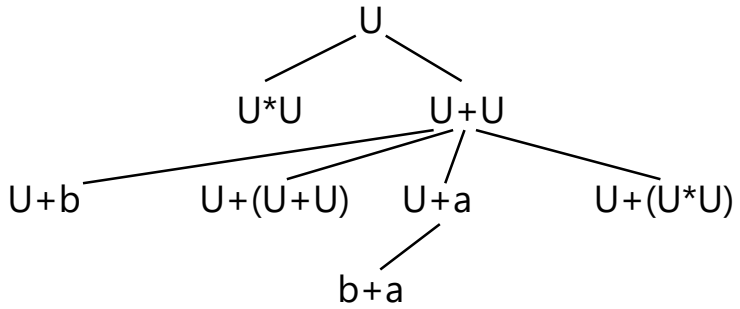
```
def synthesizer(inputs):
    (i_1 ... i_n) = inputs
    query = (∃P. σ(i_1, P) ∧ ... ∧ σ(i_N, P))
    result, model = decide(query)
    if result is SAT:
        return model
    else:
        return UNSAT
```

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


Gulwani et al. Program Synthesis. 2018.
Sharma et al. Finding Substitutable Binary Code for Reverse Engineering by Synthesizing Adapters. ICST'18.

Program Synthesis

- Syntia
  - Formulate the problem as a game tree search
  - Solve MCTS problem by UCT (Upper Confidence bounds applied to Trees) with simulated annealing
    - Calculate score for each node;
      - How to calculate it? – Similar to equivalence checking
    - Backtrack
  - Open source: https://github.com/RUB-SysSec/syntia



U: Non-terminal symbol
a, b, c, …: Input variables

Blazytko et al. Syntia: Synthesizing the Semantics of Obfuscated Code. USENIX Security'17.

Program Synthesis

- Limitations
  - Program synthesis in general
    - Non-determinism
    - Point functions
  - CEGIS
    - Sometimes tries to synthesize a constant

https://twitter.com/johnregehr/status/1216960249501900800

Hands-On 5: VM Deobfuscation

- Duration: 30min
- Objective: Deobfuscate ZeusVM with a simple symbolic execution
- Step 1: Malware analysis
  - Analyze zeus.bin with IDA and locate VM handlers

```
db    0
db    0
dd offset sub_40EDE0    ; DATA XREF: sub_414795+47↑r
dd offset sub_40EDFB
dd offset sub_40EE19
dd offset sub_40EE3A
dd offset sub_40EE64
dd offset sub_40EE92
dd offset sub_40EEBE
dd offset sub_40EEE8
dd offset sub_40EF16
dd offset sub_40EF42
dd offset sub_40EF6C
dd offset sub_40EF9A
dd offset sub_40EFC6
dd offset sub_40EFFD
dd offset sub_40F038
dd offset sub_40F070
dd offset sub_40F0A7
dd offset sub_40F0E2
dd offset sub_40F11A
dd offset sub_40F13E
dd offset sub_40F164
dd offset sub_40F189
dd offset sub_40F2E2
```

MD5: eabe05d521875308e724560be02f4482
ZeusVM is a notorious Trojan

Hands-On 5: VM Deobfuscation

- Step 2: hands-on5/vm_explore.ipynb
  - Run vm_explore.ipynb
    - The method expr_simp allows you to name and reduce symbolic expressions with lambda expressions

```
# Show results
print('final states:', len(final_states))

for final_state in final_states:
    if final_state.result:
        print('Feasible path:',
              '->'.join([str(x) for x in final_state.path_history]))
        print('¥t',final_state.path_conds)
    else:
        print('Infeasible path:',
              '->'.join([str(x) for x in final_state.path_history]))
        print('¥t',final_state.path_conds)

    final_state.sb.dump(ids=False)
    print('')
```

hands-on3/simple_explore.ipynb

```
for i in range(69): # Iterate over handlers
    # Get each handler address from the array
    addr = int(hex(upck32(bs.getbytes(mnemonic_array_addr + 4*i, 4))), 16)
    print('*'*40, 'Mnemonic', i, ' addr', hex(addr), '*'*40)
    ...
    # Show results
    for final_state in final_states:
        if final_state.result:
            ret_mn = expr_simp(final_state.sb.eval_expr(regs.EAX[:8]))
            ....
            # Show state after expr_simp
            dump_state(final_state.sb)

        #final_state.sb.dump(ids=False) # Show state before expr_simp
        print('')
```

hands-on5/vm_explore.ipynb

Hands-On 5: VM Deobfuscation

- Step 2: hands-on5/vm_explore.ipynb
  - What was needed in advance?
    - Locate handler addresses
    - Analyze some semantics (VM_PC_init and RET_ADDR in our hands-on)
  - How can we speed up the analysis?
    - Compare it with solved/zeus_get_ir.ipynb and implement your idea

solved/zeus_get_ir.ipynb was implemented with reference to https://miasm.re/blog/2016/09/03/zeusvm_analysis.html

VM Deobfuscation

- Limitations
  - Locate, Extract
    - In most cases, dynamic analysis is also required
      - Rarely as simple as ZeusVM
    - What if the assumption about VM context switch is broken?
  - Simplify
    - Our hands-on required manual semantic reasoning
    - Techniques described before are basically based on symbolic execution;
    - Limited by the limitations of symbolic execution
    - Which simplification method is better? Empirical analysis is required

(De-)Obfuscation Techniques

- Different techniques, common ideas:
  - Do useless things
    - Garbage/Dead Code Insertion
  - Change syntax
    - Instruction Substitution
    - Encode Literals
    - Encode Arithmetic
  - Change not only syntax but also semantics
    - Opaque Predicate
    - Virtualization Obfuscation
    - Control Flow Flattening

Dataflow Analysis
(Liveness Analysis)

Dataflow Analysis
(Reachable Definition Analysis)

Symbolic Execution
Equivalence Checking

VMHunt
Program Synthesis

Graph Pattern Matching
Dynamic Symbolic Execution

Control Flow Unflattening

- Control Flow Flattening can be addressed by the following:
  - Graph pattern matching
  - Symbolic execution
    - Example: https://github.com/eset/stadeo (IDAPython is required)
  - Dynamic symbolic execution
    - Symbolic execution + dynamic execution = dynamic symbolic execution
    - (w/ taint analysis)
- Sometimes manual or dynamic analysis is required

Graph Pattern Matching

- How it works
  - Locate a flattened jump table
  - Identify variables involved in state management
  - Keep track of the order in which blocks are executed
  - Generate a chain of the blocks
  - Rewrite the branch instruction to merge consecutive blocks
- The plugin for IDA is publicly available: https://github.com/carbonblack/HexRaysDeob/
  - Performs analysis at IDA Microcode (IR) level
  - Even works for ANEL malware sample
  - Unfortunately, IDA Freeware cannot process this plugin

Graph Pattern Matching

- Miasm provides interfaces for subgraph matching and block merging
- In Miasm, such a heuristics implementation would be:
  - Set a callback for disassemble;
  - Find blocks which match a specified pattern;
  - Apply a block merging pass of DiGraphSimplifier to the blocks
- Question: Is it robust?

Dynamic Symbolic Execution

- If the target is doing something with the input, skipping static analysis would be efficient
- In Miasm, DSE implementation would be: prepare a Sandbox instance and attach DSEPathConstraint to it
  - Still, this requires a manual analysis of the input size/type and termination conditions
- Example: hands-on6/cff_dse.ipynb

```
# Initialize a sandbox environment
sb = Sandbox_Linux_x86_64(loc_db, options.filename, options,
      globals())
machine = Machine('x86_64')

ret_addr = 0x000000001337beef
sb.jitter.add_breakpoint(ret_addr, code_sentinelle)
sb.jitter.push_uint64_t(ret_addr)
sb.jitter.vm.add_memory_page(...
```

```
# Initialize a DSE instance with a given strategy
dse = DSEPathConstraint(machine, produce_solution=strategy)
dse.attach(sb.jitter) # Attach to the sandbox
dse.update_state_from_concrete()

# Symbolize the argument
regs = sb.jitter.ir_arch.arch.regs

...
```

Hands-On 6: Control Flow Unflattening

- Duration: 30min
- Objective: Unflatten the flattened binary files
- Step 1: hands-on6/cff_dse.pynb
  - Run cff_dse for the flattening-volatile.bin
    - Returns 0 only when given a specific string (flag)
- Step 2: hands-on6/cff_explore.ipynb
  - Customize cff_explore for simplifying the CFG of the test-mod2-fla.bin
  - Let's make a unflattened CFG and display it
    - No answer provided, try your idea

```
import pydotplus
from IPython.display import Image, display_png
# Visualize the CFG
with open('cfg.dot', 'w') as f:
    f.write(ircfg.dot())
graph = pydotplus.graphviz.graph_from_dot_file('cfg.dot')
graph.write_png('cfg.png')
display_png(Image(graph.create_png()))
```

Control Flow Unflattening

- Limitations
  - Hardening CFF is easy
    - *Inter-procedural data flow*
      - Initialize global variables at the call site for the state management
    - Adding opaque predicate
    - Encoding/hiding next block number

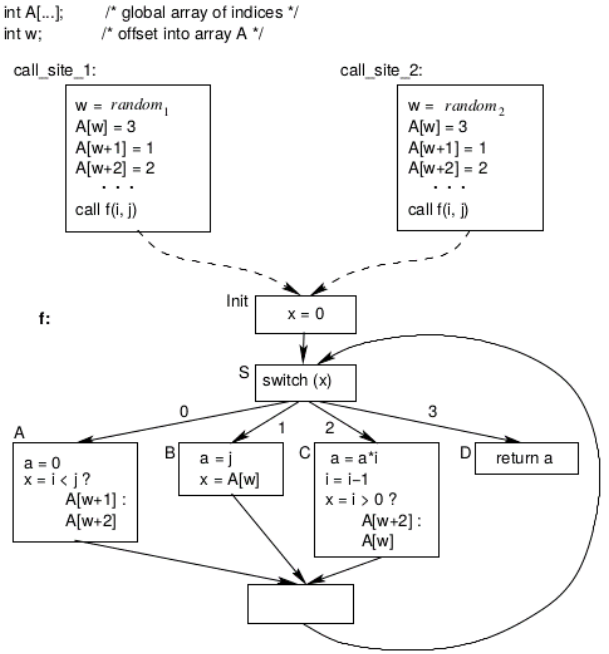


Figure 3: Enhancing flattening with Interprocedural Data Flow

Udupa et al. Deobfuscation: Reverse Engineering Obfuscated Code. WCRE'05.
Cappaert and Preneel. A General Model for Hiding Control Flow. DRM'10.

Conclusion

Summary

- Obfuscation Techniques
  - Preliminaries
  - Garbage Code Insertion, Instruction Substitution, …
  - Hands-On
- Deobfuscation Techniques
  - Preliminaries
  - Dataflow Analysis, Symbolic Execution, Equivalence Checking, …
  - Hands-On
- Now you have a skill of implementing these techniques in practice
  - But all have their limitations

Takeaways

- Difficulty level
  - Easy:
    - Being used by tools/methods: "It doesn't work."
  - Medium:
    - Mastering tools/methods: “It doesn’t work, because …”
  - Hard:
    - Mastering tools/methods; push the envelope: “I got this to work!”

Takeaways

- Lets' look back:
  - What are the obfuscation techniques?
  - What are the deobfuscation techniques?
  - What is the scope and limitations of deobfuscation methods?
- Imagine:
  - How to overcome the limitations of existing methods?
  - What if you face unknown obfuscation techniques?
- Let's see through the essence

Other Topics

- Other dataflow analysis methods
- Taint analysis

Acknowledgement

- Yuhei Kawakoya
  - Course material and sample code
- Makoto Iwamura
  - Sample code
- Ryo Ichikawa (icchy)
  - Environment setup

Recommended Readings

- Aho et al. Compilers: Principles, Techniques, and Tools. 1986.
- Collberg and Nagra. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. 2009.
- Gazet et al. Practical Reverse Engineering: X86, X64, ARM, Windows Kernel, Reversing Tools, and Obfuscation. 2014.
- Andriesse. Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly. 2018.
- Banescu. Breaking Obfuscated Programs with Symbolic Execution. https://www.slideshare.net/SebastianBanescu/breaking-obfuscated-programs-with-symbolic-execution. 2017.
- Rolles. Möbius Strip Reverse Engineering. http://www.msreverseengineering.com/.
- Yurichev. SAT/SMT by Example. https://yurichev.com/writings/SAT_SMT_by_example.pdf.
- And research papers referenced in this material