Automating Binary Deobfuscation Processes: Dynamic Taint Analysis and Symbolic Execution

Usama Saqib

- Bilkent Üniversitesi - Bilgisayar Mühendisliği 4. Sınıf Öğrencisi
- Binary Analysis, Reverse Engineering, Malware Analysis.

Berk Cem Göksel

- Tobb ETÜ - Siber Güvenlik MSc.
- Bilkent Üniversitesi - BFA, Theatre
- STM A.Ş. Siber Tehdit İstihbarat Analisti
- 5+ Sızma Testi / Red Team

Overview - Content

Obfuscation techniques we will focus on:

-Control Flow Obfuscation using Opaque Predicates

-Code Virtualization

De-obfuscation techniques we will use:

-Dynamic Taint Analysis

-Symbolic Execution

Overview - Why?

Malware Analysis project:

- Phylogenetic analysis of malwares to construct families.
- Perform Code-reuse detection.
- Attack attribution.

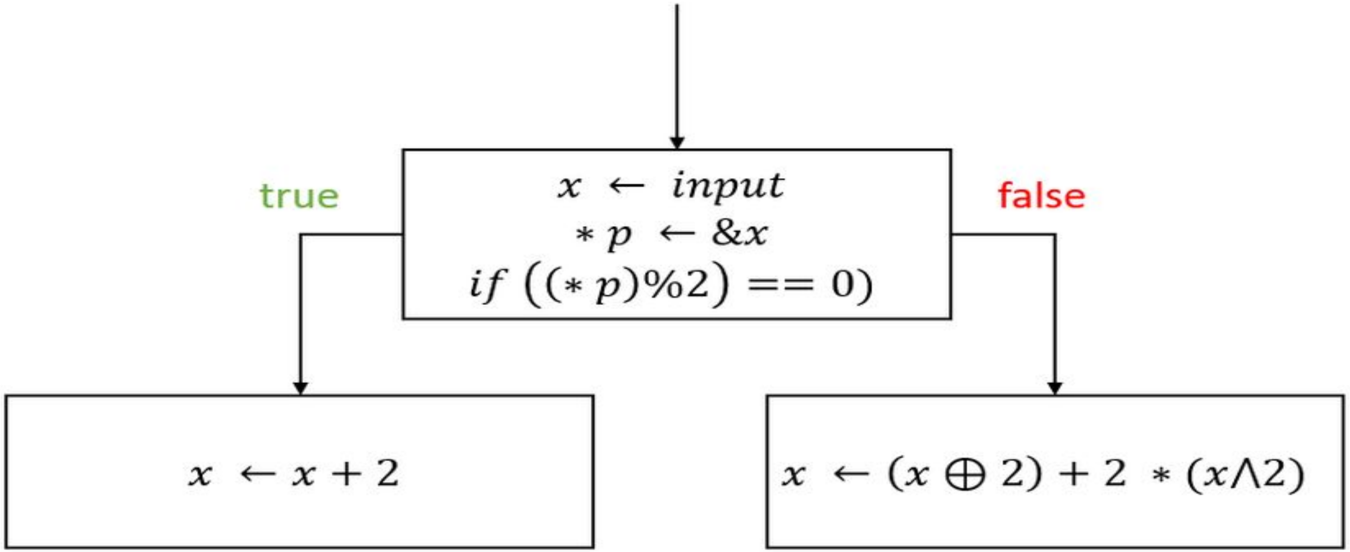
To do this we need:

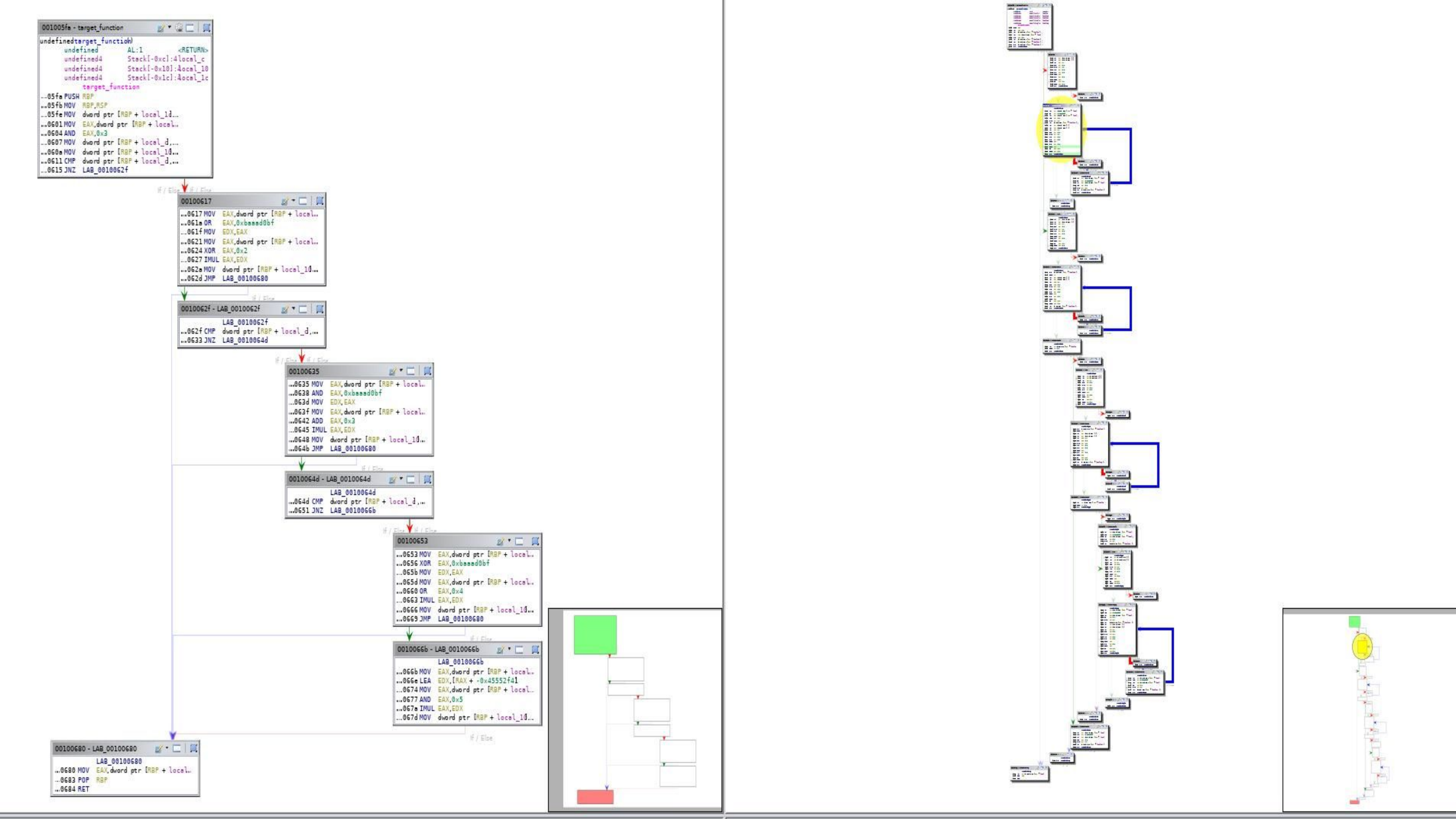
- Accurate execution traces
- Bypassing encryption, packing, and code obfuscations, (semi)automatically.

Obfuscation Routines

Control Flow Obfuscation: Opaque Predicates

Opaque Predicate: An expression that evaluates to a boolean at runtime. Hard to determine which, statically [5].




001005fa - target_function
undefined target_function()
undefined AL:1 <RETURN>
undefined4 Stack[-0xc]:4 local_c
undefined4 Stack[-0x10]:4 local_10
undefined4 Stack[-0x1c]:4 local_1c
target_function
...05fa PUSH RBP
...05fb MOV RBP,RSP
...05fe MOV dword ptr [RBP + local_1c]...
...0601 MOV EAX,dword ptr [RBP + local...
...0604 AND EAX,0x3
...0607 MOV dword ptr [RBP + local_c],...
...060a MOV dword ptr [RBP + local_10]...
...0611 CMP dword ptr [RBP + local_c],...
...0615 JNZ LAB_0010062f
00100617
...0617 MOV EAX,dword ptr [RBP + local...
...061a OR EAX,0xbaaad0bf
...061f MOV EDX,EAX
...0621 MOV EAX,dword ptr [RBP + local...
...0624 XOR EAX,0x2
...0627 IMUL EAX,EDX
...062a MOV dword ptr [RBP + local_10]...
...062d JMP LAB_00100680
0010062f - LAB_0010062f
LAB_0010062f
...062f CMP dword ptr [RBP + local_c],...
...0633 JNZ LAB_0010064d
00100635
...0635 MOV EAX,dword ptr [RBP + local...
...0638 AND EAX,0xbaaad0bf
...063d MOV EDX,EAX
...063f MOV EAX,dword ptr [RBP + local...
...0642 ADD EAX,0x3
...0645 IMUL EAX,EDX
...0648 MOV dword ptr [RBP + local_10]...
...064b JMP LAB_00100680
0010064d - LAB_0010064d
LAB_0010064d
...064d CMP dword ptr [RBP + local_c],...
...0651 JNZ LAB_0010066b
00100653
...0653 MOV EAX,dword ptr [RBP + local...
...0656 XOR EAX,0xbaaad0bf
...065b MOV EDX,EAX
...065d MOV EAX,dword ptr [RBP + local...
...0660 OR EAX,0x4
...0663 IMUL EAX,EDX
...0666 MOV dword ptr [RBP + local_10]...
...0669 JMP LAB_00100680
0010066b - LAB_0010066b
LAB_0010066b
...066b MOV EAX,dword ptr [RBP + local...
...066e LEA EDX,[RAX + -0x45552f41]
...0674 MOV EAX,dword ptr [RBP + local...
...0677 AND EAX,0x5
...067a IMUL EAX,EDX
...067d MOV dword ptr [RBP + local_10]...
00100680 - LAB_00100680
LAB_00100680
...0680 MOV EAX,dword ptr [RBP + local...
...0683 POP RBP
...0684 RET

```
ulong target_function(uint param_1)

{
  uint uVar1;
  uint local_10;

  uVar1 = param_1 & 3;
  if (uVar1 == 0) {
    local_10 = (param_1 ^ 2) * (param_1 | 0xbaaad0bf); /Let's focus here.
  }
  else {
    if (uVar1 == 1) {
      local_10 = (param_1 + 3) * (param_1 & 0xbaaad0bf);
    }
    else {
      if (uVar1 == 2) {
        local_10 = (param_1 | 4) * (param_1 ^ 0xbaaad0bf);
      }
      else {
        local_10 = (param_1 & 5) * (param_1 + 0xbaaad0bf);
      }
    }
  }
  return (ulong)local_10;
}
```

```
  if ((x * (x + -1) & 1U) == 0 || y < 10) goto LAB_004004d5;
  do {
    puVar1 = local_30 + -0xc;
    local_30[-4] = param_1;
    local_30[-8] = local_30[-4] & 3;
    *puVar1 = 0;
LAB_004004d5:
    local_20 = puVar1 + -4;
    local_28 = puVar1 + -8;
    local_30 = puVar1 + -0xc;
    *local_20 = param_1;
    *local_28 = *local_20 & 3;
    *local_30 = 0;
    local_31 = *local_28 == 0;
  } while ((x * (x + -1) & 1U) != 0 && 9 < y);
  if (*local_28 == 0) {
    *local_30 = (*local_20 | 0xbaaad0bf) * (*local_20 ^ 2);
  }
```

Code Virtualization [4]

The obfuscated region is written using a virtualized instruction set.

An interpreter stub is inserted into the program, which decodes and executes the virtualized instructions during runtime.

Code Virtualization: Why?

- Execution traces will only show VM machinery
- AV/EDR Evasion
- Static analysis will involve reversing the VM and its semantics.

READ: reads an integer on stdin and push the value on the stack, or exit if input is invalid

WRITE: pops the top value of the stack, and prints it on stdout

DUP: duplicate the value on top of the stack

ADD: pops the two top value of the stack, add them and push the result on top of the stack
push the result on the stack

GT: LT: EQ: pops the two top values from the stack, compare them for TOP > SECOND, TOP < SECOND or TOP == SECOND and push the result as 0 or 1 on the stack

JMPZ: pops the two top value of the stack. Jump to the <n>th instruction, where <n> was the first value on the stack, if the top value is null. Otherwise just drop these two values

PUSH: push the integer value <n> on the stack

POP: pop n value from the stack

ROT: perform a circular rotation on the first n value of the stack toward the top for instance the stack : BOTTOM [1,2,4,8] TOP becomes BOTTOM [1, 8, 2, 4] TOP after ROT 3

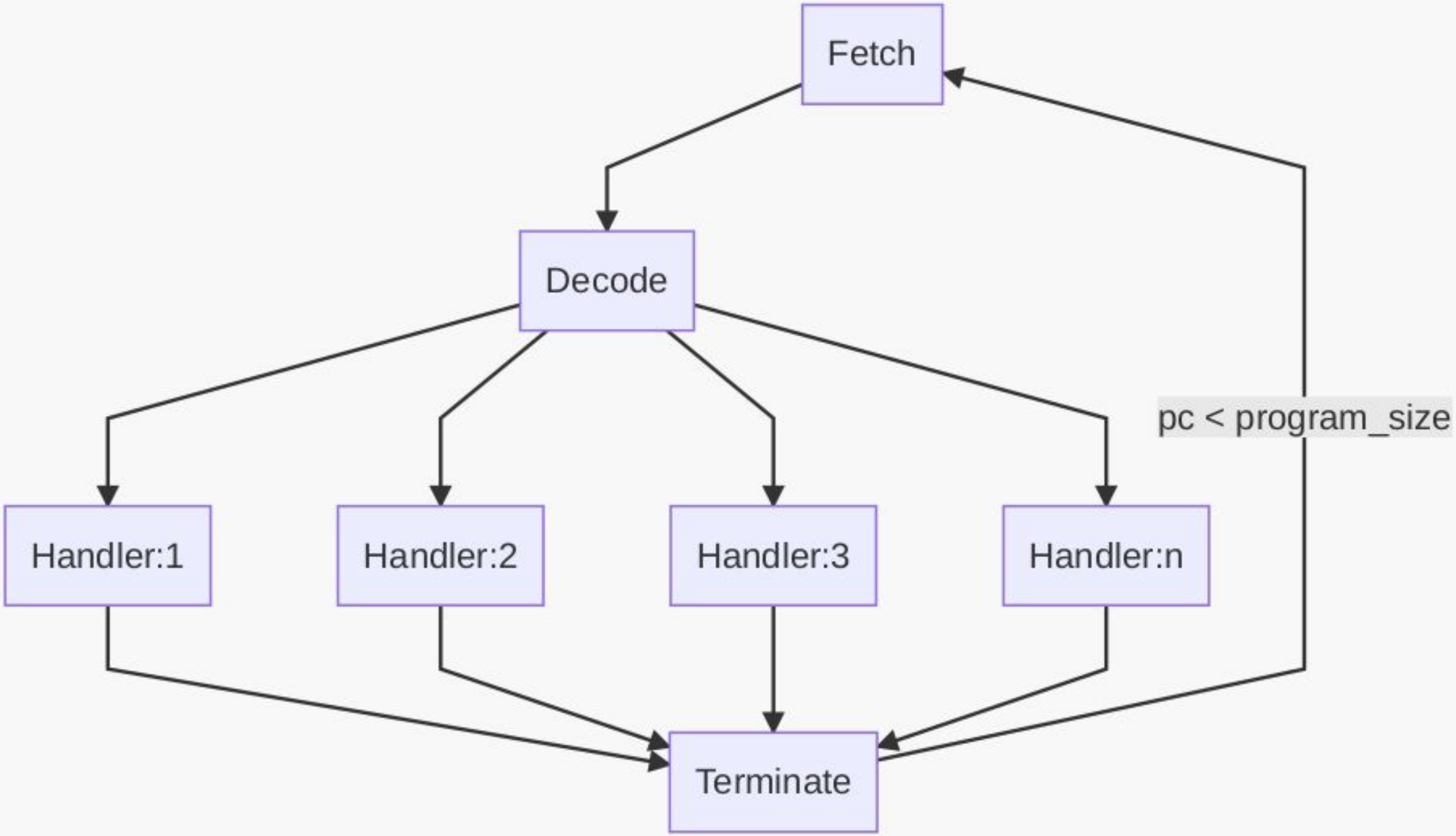
Interpreter Structure

1. **Fetch**: The instructions are fetched from the machine's program memory, as indicated by a program counter.

2. **Decode**: The instruction is decoded and and the control flow is passed to the relevant handler.

3. **Handle**: Each instruction has a handler which implements its logic. The handler interacts with the stack memory and updates it as required.

4. **Terminate**: The machine terminates once all the instructions have been executed, or an illegal instruction is met.


Fetch
Decode
Handler:1
Handler:2
Handler:3
Handler:n
Terminate
pc < program_size

Virtualization Providers

- The Tigress C Diversifier/Obfuscator
- VMProtect
- EXECryptor
- Themida
- Code Virtualizor

```cpp
typedef enum Mnemonic{
    READ, WRITE, DUP, MUL, ADD, SUB, GT, LT, EQ, JMPZ, PUSH, POP, ROT
    }Mnemonic;

typedef struct Ins {
    Mnemonic op;
    int32_t arg = 0;
} Ins;

class Interpreter
{
private:
    std::stack<int32_t> mem;
    Ins* program;
    int program_size;
    int pc;

public:
    Interpreter(Ins* program, int size);
    void run();
    void terminate(bool*);
    void print_stack(Ins ins);
};
```

De-obfuscation Techniques

Dynamic Taint Analysis

The purpose of dynamic taint analysis is to track information flow between sources and sinks [1].

- Define Taint Sources -> Ex: User Input (Sources)
- Check which portions of the code use that memory space. (Sinks)

The way information is tracked, is done via **taint policies**

Dynamic Taint Analysis - Policy

A policy consists of three properties:

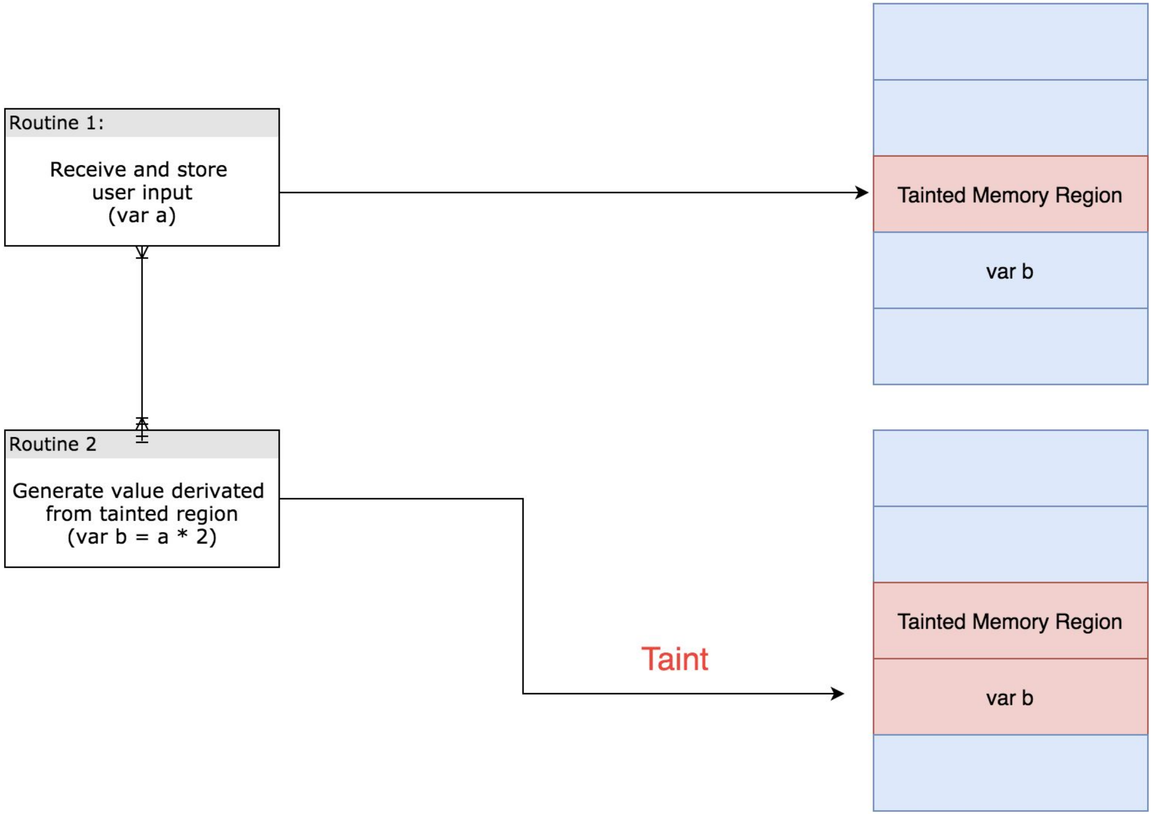
- Taint Introduction (syscall return values [eax/rax] and library returns)
- Taint Propogation (var a = var a + var b) ---> var a
- Taint Checking (Ex: Kill process/thread if execution redirects to user supplied input)

Taint Introduction

Example:

- Syscall return values [eax/rax]
- Library returns
- User input

Taint Propagation



Taint Checking in Exploitation Prevention

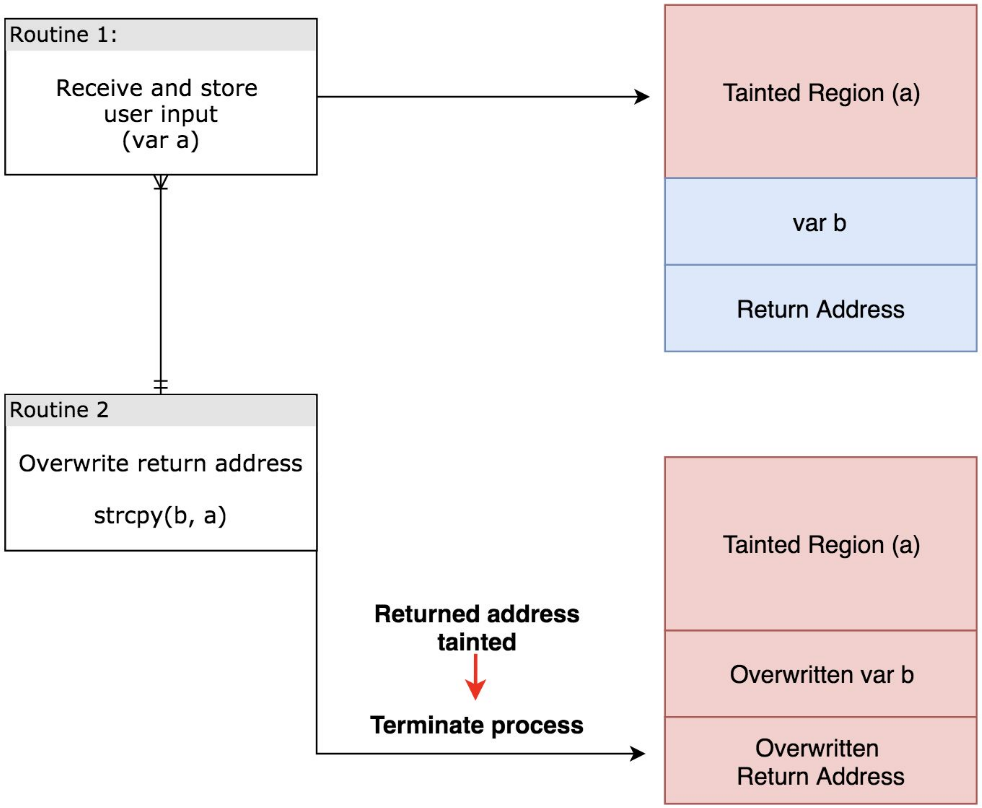
Shellcode Overwrite Exploits:

- Terminate if execution passes to tainted area (user supplied)

ROP Exploits:

- Terminate if a return address or a function pointer is overwritten with a tainted value.

Taint Checking - Exploitation Prevention



Dynamic Taint Analysis on Production Environment?

- DTA is built on top of Dynamic Binary Instrumentation Frameworks
- Expensive runtime overhead

Taint Checking

Example - Malware Analysis:

- Look for specific syscalls such as execve(), socket() and etc.
- Check how program utilizes the data coming from the environment.
- Identifying evasion, anti-debugging, C&C communication.

Dynamic Taint Analysis - Challenges [1]

Defines how that taint propagates

- Data dependency
- Control Flow dependency
- Implicit flows

Runtime overhead

Overtainting, Undertainting

- Taint bit
- Taint byte
- Taint word

Dynamic Taint Analysis - Challenges

Anti-Taint Analysis. Example: Implicit Flows

Consider the following code snippet:

```
var a //Tainted value

for 0 to a*2:

    var b = var b + 1
```

Above code equivalent to:

```
var b = var a * 2
```

Note that var b remains untainted.

Symbolic Code Execution

Software analysis technique that expresses program state in terms of logical formulae.

Forward symbolic execution allows us to reason about the behavior of a program by building a logical formula that represents a program execution [1].

- Is a particular program point reachable?
- Is a particular state possible, E.g. array access a[i] out of bounds?

Symbolic Code Execution

Detecting infeasible paths (Dead Code)

Generating test inputs (Code Coverage)

Finding bugs and vulnerabilities

Which instruction contributed to a value at a certain point (Backward Slicing)

Symbolic vs. Concrete Execution

Symbolic	Concrete
Executes with symbolic values Computes logical formulas over these symbols Emulates all possible control flow	Executes with actual values Computes exact values as determined by the execution Executes along a single control flow

Symbolic State

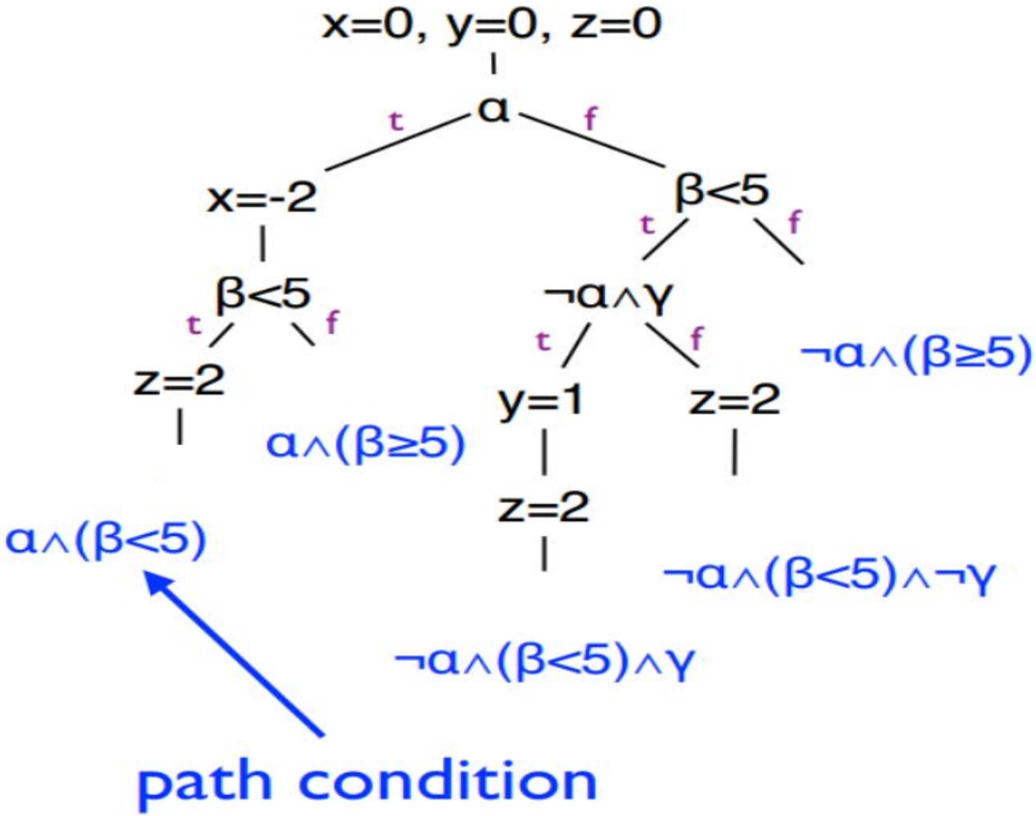
Symbolic state comprises of two logical formulae.

1. Symbolic expressions
    - Corresponds to either a symbolic value, var x <- $\boldsymbol{\alpha}$ var y <- $\boldsymbol{\beta}$, or
    - a mathematical combination of symbolic expressions, $\boldsymbol{\Phi = \alpha + \beta}$

2. Path constraints
    - Encode the limitations on symbolic expressions by branches.
    - if (x < 5) { … if (y > 0) { … } … } becomes $\boldsymbol{\alpha < 5 \land \beta > 0}$

Symbolic Code Execution Example

```
int a = α, b = β, c = γ;
                // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



PoC

Tools Used

Triton [6]:

- Symbolic Execution
- Backward Slicing

Libdft [3]:

- Dynamic Taint Analysis

Deobfuscation Procedure

Approach 1: Semi-automatic

- Analysis is performed on manually selected portions.

Approach 2: Automatic

- Full binary emulation.
- Scalability issues.

Deobfuscation Procedure: Approach 1

Variation of approach described in the paper, “Symbolic deobfuscation: from virtualized code back to the original” (Jonathan Salwan Sebastien Bardin and Marie-Laure Potet) [2]

Deobfuscation Procedure: Approach 1

Step 0:

- Manual Reverse Engineering
- Automatically identifying beginning of obfuscated region.
    - Projects such as VMHunt automatically find virtualized regions by identifying context switches.

Deobfuscation Procedure: Approach 1

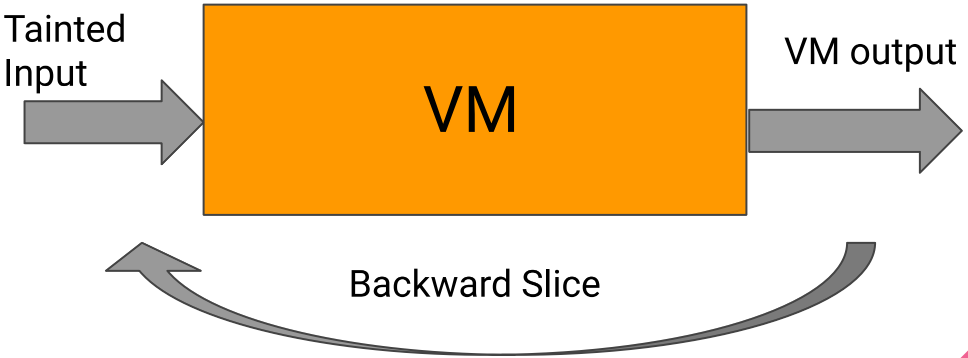
Step1: Dynamic Taint Analysis

- Identify source of input for the virtualized region.
- Taint the source.
- Perform Dynamic Taint Analysis to isolate VM machinery

Deobfuscation Procedure: Approach 1

Step 2: Symbolic Execution

- Perform symbolic execution.
- Compute backward slice from VM output to Tainted input.



Deobfuscation Procedure: Recovering Algorithm

- Backward slice computed from the instruction outputting the result of the virtualization.
- This gives us all the previous instructions that contributed to the final value.
- **We have effectively removed all instructions related to the VM machinery.**
- **And are left with the instructions that execute the virtualized program.**

```
#1
READ
DUP
PUSH 0
LT
PUSH 28
JMPZ
DUP
PUSH 1
# 2
ROT 2
ROT 3
DUP
ROT 3
ROT 2
DUP
ROT 5
GT
PUSH 27
JMPZ
DUP
ROT 3
MUL
ROT 2
PUSH 1
ADD
PUSH 0
PUSH 8
JMPZ
# 3
ROT 2
WRITE
```

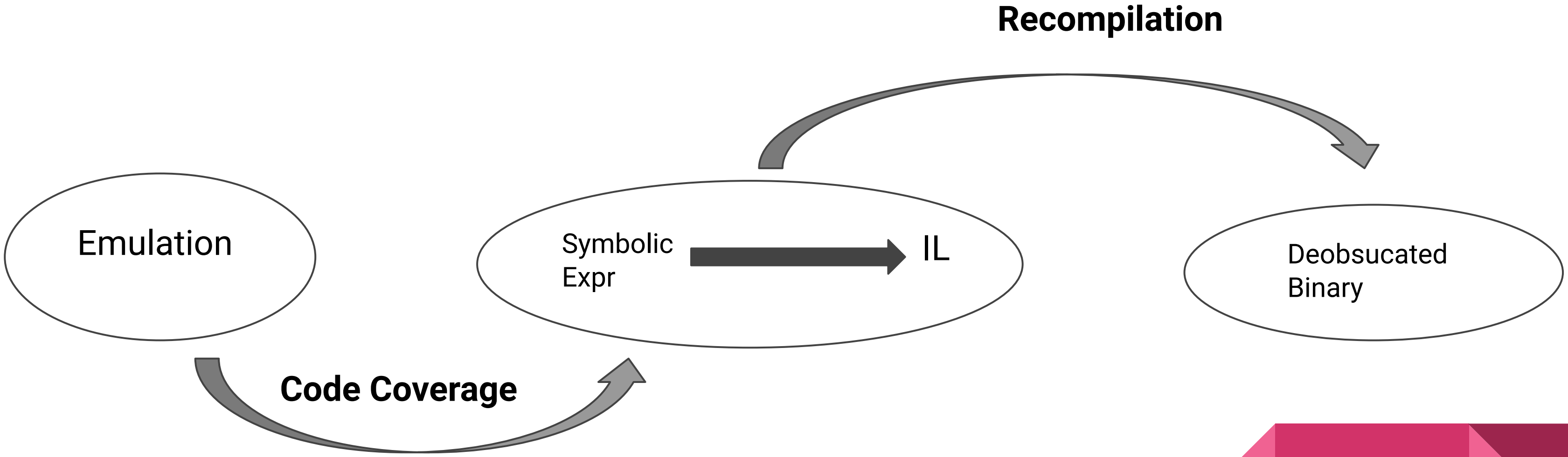
```
0x8048b12: imul eax, dword ptr [ebp - 0x74]
0x8048b82: add eax, edx
0x8048b12: imul eax, dword ptr [ebp - 0x74]
0x8048b82: add eax, edx
0x8048b12: imul eax, dword ptr [ebp - 0x74]
0x8048b82: add eax, edx
0x8048b12: imul eax, dword ptr [ebp - 0x74]
```

Deobfuscation Procedure: Approach 2

- Full Binary emulation
- Use Symbolic Execution to perform complete code coverage
- Translate symbolic expressions to LLVM IR
- Compile LLVM IR to binary
- Compiler optimization remove obfuscations.

Deobfuscation Procedure: Approach 2

Workflow



References

[1] “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask),” ACM Digital Library. [Online]. Available: https://dl.acm.org/citation.cfm?id=1849981. [Accessed: 21-Dec-2019].

[2] J. Salwan and M.-L. P. Sébastien Bardin, “Symbolic Deobfuscation: From Virtualized Code Back to the Original,” SpringerLink, 28-Jun-2018. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-93411-2_17. [Accessed: 21-Dec-2019].

[3] “libdft,” libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. [Online]. Available: http://www.cs.columbia.edu/~vpk/research/libdft/. [Accessed: 21-Dec-2019].

[4] “Improved Virtual Machine-Based Software Protection,” NISLVMP. [Online]. Available: https://dl.acm.org/citation.cfm?id=2586077. [Accessed: 21-Dec-2019].

[5] D. Xu, J. Ming, and D. Wu, “Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method: Semantic Scholar,” undefined, 01-Jan-1970. [Online]. Available: https://www.semanticscholar.org/paper/Generalized-Dynamic-Opaque-Predicates:-A-New-Flow-Xu-Ming/0133dc4995301d41e5862a161220b7ec0e43a45f. [Accessed: 21-Dec-2019].

[6] Quarkslab, “Internal Views,” Triton. [Online]. Available: https://triton.quarkslab.com/. [Accessed: 21-Dec-2019].

BACKUP SLIDES

Dynamic Taint Analysis

The purpose of dynamic taint analysis is to track information flow between sources and sinks.

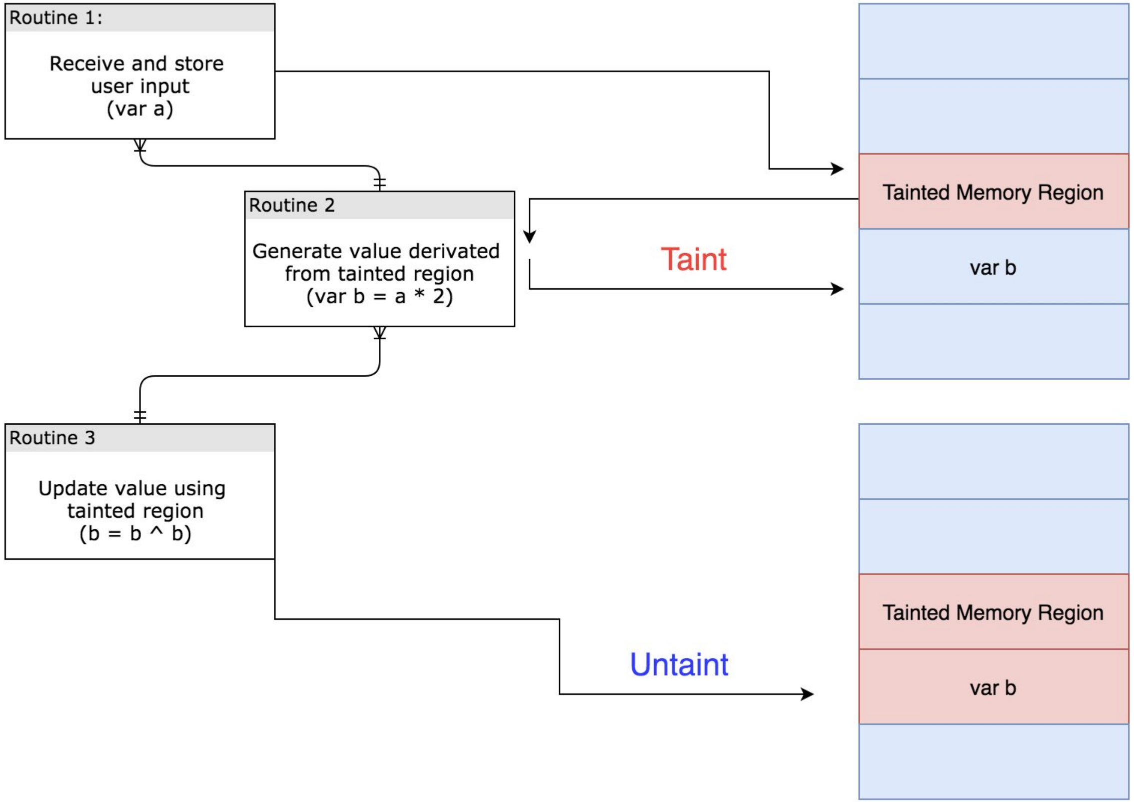
Any program value whose computation depends on data derived from a taint source is considered tainted

We can track which portions of execution are affected by tainted data.

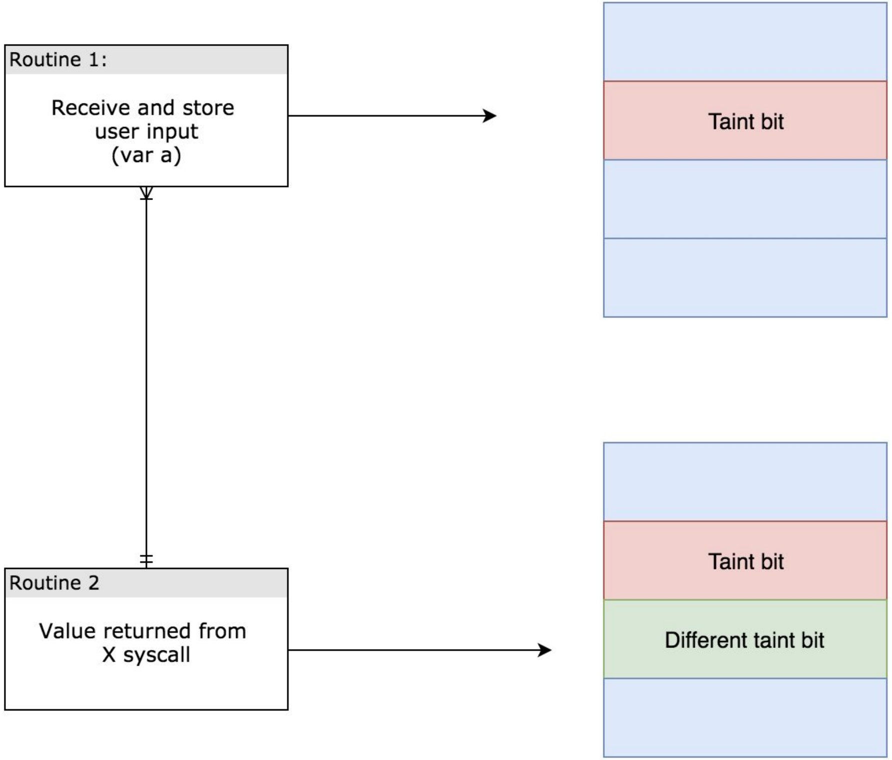
Rules are defined to propagate taint to other memory regions.

{Ref; All You Ever Wanted to Know AboutDynamic Taint Analysis and Forward Symbolic Execution(but might have been afraid to ask}

Taint Policy



Taint Policy



Deobfuscation Procedure: Approach 1

Variation of approach described in the paper, “Symbolic deobfuscation: from virtualized code back to the original” (Jonathan Salwan Sebastien Bardin and Marie-Laure Potet) [2]

3 Step Algorithm:

- Step 0: Identify input to the obfuscated region.
- Step 1: Perform Taint Analysis to identify boundaries of the region to be analyzed.
- Step 2: Reconstruct virtualized algorithm.