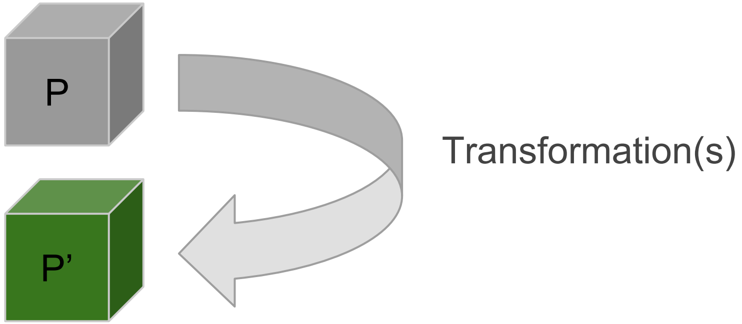
Symbolic Deobfuscation
From Virtualized Code Back to the Original

Jonathan Salwan,
Sébastien Bardin and Marie-Laure Potet
DIMVA 2018

Binary Protections

Binary Protections

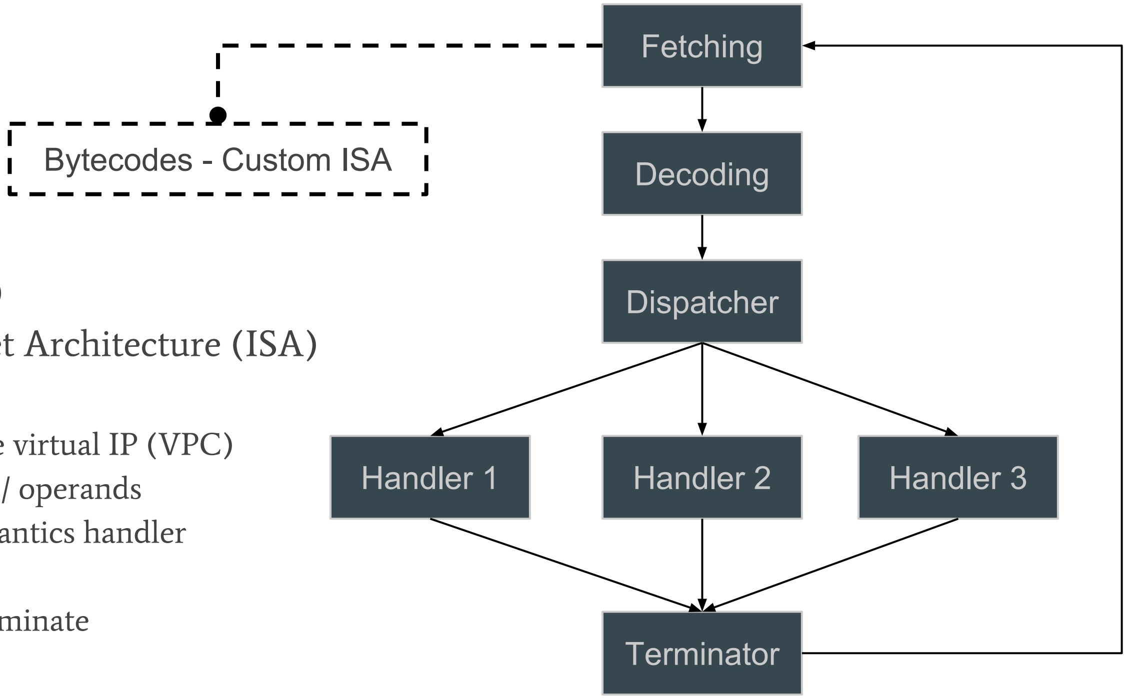
- Goal
    - Transform your program to make it hard to analyse
        - Protect your software against reverse engineering (Intellectual Property)



Binary Protections

- There are several kinds of protections
  - Anti-Tampering
    - Anti-debug
    - Anti-VM
    - Integrity checks
  - Data protection
    - Data encoding
    - Data encryption
    - Opaque constants
  - Code protection
    - Code flattening
    - Junk code injection
    - Operations encoding
    - **Virtualization-based software protection**

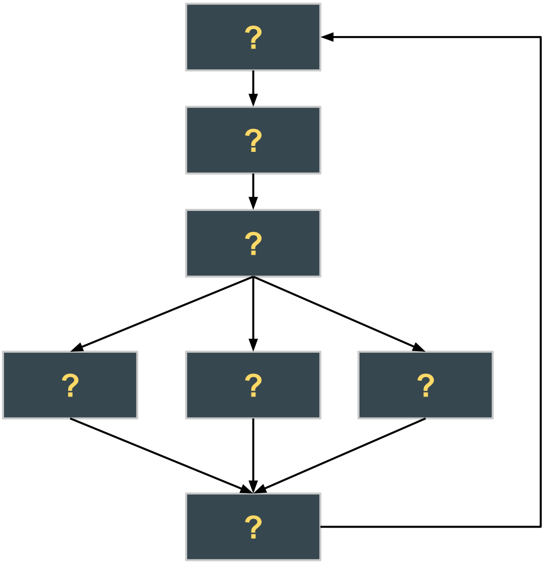
Binary Protection - Virtualization Design (a simple one)



- Also called Virtual Machine (VM)
- Virtualize a custom Instruction Set Architecture (ISA)
- Close to a CPU design
    - Fetch the opcode pointed via the virtual IP (VPC)
    - Decode the opcode - mnemonic / operands
    - Dispatch to the appropriate semantics handler
    - Execute the semantics
    - Go to the next instruction or terminate

Virtual Machine - Challenges for a Reverser

1. Identify that the obfuscated program is virtualized, and identify its inputs
2. Identify each component of the virtual machine
3. Understand how these components work together
4. Understand how VPC is computed
5. Create a disassembler for the custom ISA
6. Start to analyse the original behavior



State of the Art

Position of our Approach

	Manual	Kinder	Coogan	Yadegari	Our Approach
Identify input **Understand VPC** **Understand dispatcher** **Understand bytecode**	Required Required Required Required	Required Required No No	Required No No No	Required No No No	Required No No No
output	Simplified CFG	CFG + invariants	Simplified Trace	Simplified CFG	Simplified Code
Key techno.	-	Static analysis (abstract interp.)	Value-based slicing	Taint, symbolic and instruction simpl.	Taint, symbolic, formula simpl. and code simpl.
xp: type of code **xp: #samples** **xp: evaluation metrics**	- - -	Toy example 1 Known invariants	Toys + malware 12 %Simplification	Toys+malware 44 Similarity	Hash functions 945 Size, Correctness

Our Approach
Automatic Deobfuscation

Our Approach - Automatic Deobfuscation

Key intuition:

obfuscated trace = **original instructions** + **virtual instructions**

Our Approach - Automatic Deobfuscation

Key intuition:

obfuscated trace = **original instructions** + **virtual instructions**

0. Identify inputs

Our Approach - Automatic Deobfuscation

Key intuition:

obfuscated trace = **original instructions** + **virtual instructions**

0. Identify inputs
1. On a trace, isolate pertinent instructions using a dynamic taint analysis

Our Approach - Automatic Deobfuscation

Key intuition:

obfuscated trace = **original instructions** + **virtual instructions**

0. Identify inputs
1. On a trace, isolate pertinent instructions using a dynamic taint analysis
2. Build a symbolic representation of these tainted instructions

Our Approach - Automatic Deobfuscation

Key intuition:

obfuscated trace = **original instructions** + **virtual instructions**

0. Identify inputs
1. On a trace, isolate pertinent instructions using a dynamic taint analysis
2. Build a symbolic representation of these tainted instructions
3. Perform a path coverage analysis to reach new tainted paths

Our Approach - Automatic Deobfuscation

Key intuition:

obfuscated trace = **original instructions** + **virtual instructions**

0. Identify inputs
1. On a trace, isolate pertinent instructions using a dynamic taint analysis
2. Build a symbolic representation of these tainted instructions
3. Perform a path coverage analysis to reach new tainted paths
4. Reconstruct a devirtualized binary from the path-tree of pertinent instructions

Step 1 - Dynamic Taint Analysis

- **Goal:** Separate original instructions from virtual machine instructions
- **Input:** A protected binary
- **Process:** Taint user inputs (discovered over Step-0) and execute
- **Output:** Two sub-traces of instructions
    - Tainted instructions = pertinent instructions

Step 2 - Symbolic Representation

- **Goal:** Abstract the pertinent instruction sub-trace in terms of symbolic expressions
    a. Prepare a dynamic symbolic exploration (Step-3)
    b. Provide a symbolic representation to ease translation (Step-4)
- **Input:** A sub-trace of pertinent instructions
- **Process:** Represent the trace execution as symbolic expressions and concretize everything which is not tainted (guided by Step-1).
- **Output:** A sub-trace of pertinent instructions as symbolic expressions

Step 3 - Path Coverage

- **Goal:** Reconstruct the whole program behavior
- **Input:** A sub-trace of pertinent instructions as symbolic expressions
- **Process:** Perform a **dynamic symbolic exploration** based on pertinent instructions
- **Output:** A path-tree of pertinent instructions as symbolic expressions

Step 4 - Generate a New Binary

- **Goal:** Provide a new binary, **devirtualized**
- **Input:** A path-tree of pertinent instructions as symbolic expressions
- **Process:** Translate the symbolic representation to LLVM-IL, then **compile and optimize**
- **Output:** A new binary

Experiments

Experiments

- Controlled Experiment Setup
  - 920 protected binaries
- Uncontrolled Experiment Setup (Tigress challenges)
  - 25 protected binaries

Controlled Experiment Setup

Hash	Loops	Binary Size (inst)	# executable paths
Adler-32	✓	78	1
CityHash	✓	175	1
Collberg-0001-0	✓	167	1
Collberg-0001-1	×	177	2
Collberg-0001-2	×	223	1
Collberg-0001-3	✓	195	1
Collberg-0001-4	✓	183	1
Collberg-0004-0	×	210	2
Collberg-0004-1	×	143	1
Collberg-0004-2	✓	219	2
Collberg-0004-3	✓	171	1
Collberg-0004-4	✓	274	1
Fowler-Noll-Vo Hash (FNV1a)	×	110	1
Jenkins	✓	79	1
JodyHash	✓	90	1
MD5	✓	314	1
SpiHash	✓	362	1
SpookyHash	✓	426	1
SuperFastHash	✓	144	1
Xxhash	✓	182	1

Controlled Experiment Setup

Protecticons	Options
Anti Branch Analysis	goto2push, goto2call, branchFuns
Max Merge Length	0, 10, 20, 30
Bogus Function	0, 1, 2, 3
Kind of Operands	stack, registers
Opaque to VPC	true, false
Bogus Loop Iterations	0, 1, 2, 3
Super Operator Ratio	0, 0.2, 0.4, 0.6, 0.8, 1.0
Random Opcodes	true, false
Duplicate Opcodes	0, 1, 2, 3
Dispatcher	binary, direct, call, interpolation, indirect, switch, ifnest, linear
Encode Byte Array	true, false
Obfuscate Decode Byte Array	true, false
Nested VMs	1, 2, 3

Experiments: Criteria

C1. Precision
C2. Efficiency
C3. Robustness w.r.t. the protection

Experiments Results: Precision (C1)

- Objectives:
  - **Correctness:** Is the deobfuscated code semantically equivalent to the original code?
  - **Conciseness:** Is the size of the deobfuscated code similar to the size of the original code?
- Metrics used:
  - **Correctness:** P(seed) == P'(seed)
  - **Conciseness:**
    - Ratio of the number of instructions **Original** → **Obfuscated**
    - Ratio of the number of instructions **Original** → **Deobfuscated**

	Original	Obfuscated	Deobfuscated
Binary Size	min: 78 max: 426 avg: 196	min: 468 max: 5,424 avg: 1,205	min: 48 max: 557 avg: 119
Trace Size	min: 92 max: 9,743 avg: 726	min: 1,349 max: 47,927,795 avg: 229,168	min: 48 max: 557 avg: 143

(a) Sizes

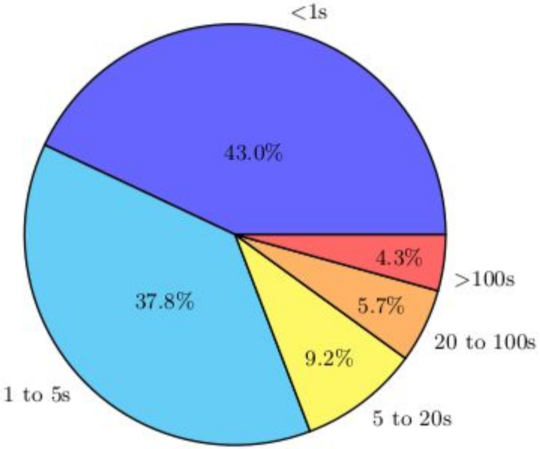
	Original → Obfuscated	Original → Deobfuscated
Correctness		100%
Binary Size	min: x3.3 max: x14.0 avg: x6	min: x0.1 max: x2.8 avg: x0.71
Trace Size	min: x17 max: x1252 avg: x424	min: x0.05 max: x0.9 avg: x0.39

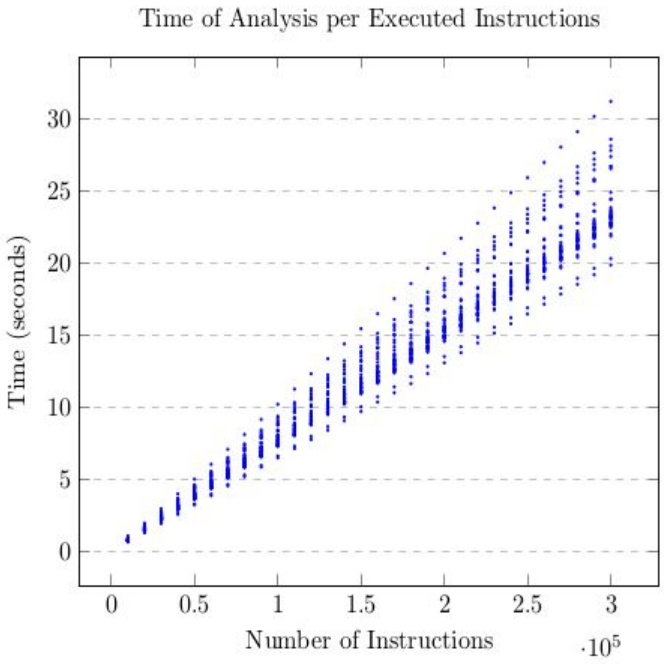
(b) Size ratios

Experiments Results: Efficiency (C2)

- Objective:
    - **Efficiency** (scalability):
        - How much time?
        - How much resources?
- Metrics used:
    - We measure the time at every 10,000 instructions handled
    - We measure the RAM consumed from the Step-1 to Step-4

Experiments Results: Efficiency (C2)

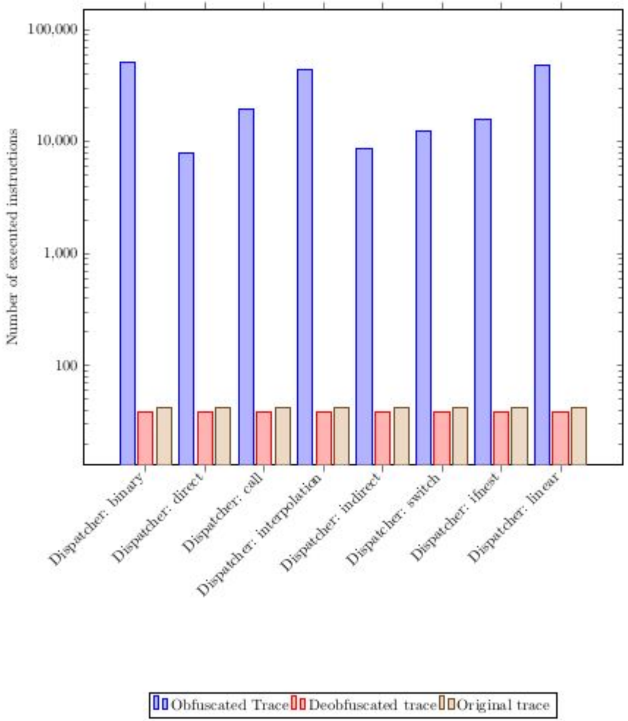


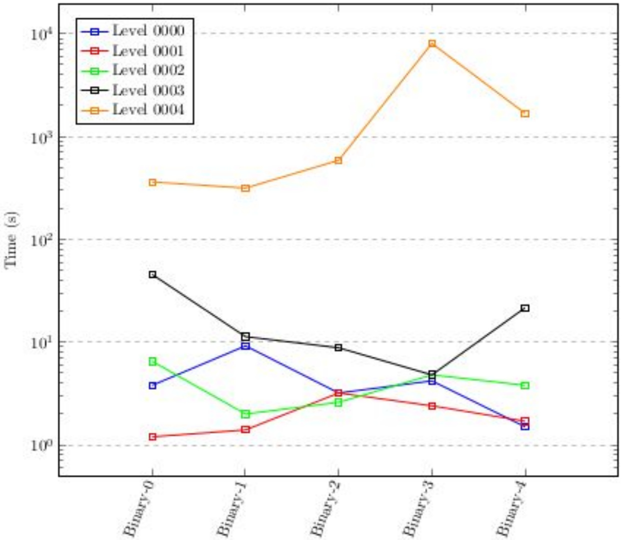


Experiments Results: Influence of Protections (C3)

- Objective:
  - **Robustness:** Do specific protections impact our analysis more than others?
- Metrics used:
  - We consider the **conciseness** metrics

Experiments Results: Influence of Protections (C3)





Experiments: Tigress Challenges

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

Experiments Results: Tigress Challenges

	Tigress Challenges				
	VM-0	**VM-1**	**VM-2**	**VM-3**	**VM-4**
0000	3.85s	9.20s	3.27s	4.26s	1.58s
0001	1.26s	1.42s	3.27s	2.49s	1.74s
0002	6.58s	2.02s	2.63s	4.85s	3.82s
0003	45.6s	11.3s	8.84s	4.84s	21.6s
0004	361s	315s	588s	8049s	1680s

Solving time (seconds)

	Tigress Challenges				
	VM-0	**VM-1**	**VM-2**	**VM-3**	**VM-4**
0000	x0.85	x1.09	x0.73	x0.89	x1.4
0001	x0.41	x0.60	x0.26	x0.22	x0.53
0002	x0.29	x0.28	x0.51	x1.4	x0.42
0003	x1.10	x1.17	x1.57	x0.46	x0.44
0004	x0.81	x0.38	x0.70	x0.37	x0.53

Ratio (size) original → deobfuscated

Limits and Mitigations

Limits

- Our approach is geared at programs with a small number of **tainted** paths
- Our current DSE model does not support user-dependent memory access
- Out of scope of our symbolic reasoning:
    - Multithreading
    - Intensive floating-point arithmetic
    - System calls
- Loops and recursive calls are handled as inlined or unrolled code
    - Increase considerably the size of the devirtualized code

Mitigations (potential defenses)

- Attacking our steps
    - The more the taint is interlaced with VM components, the less our approach will be precise
    - Hash functions over jump conditions to break paths exploration
        - E.g: if (hash(**tainted_x**) == 0x1234)
- Protecting the bytecode instead of the VM
    - If the virtual machine is broken, the attacker gets as a result an obfuscated pseudo code.

Thanks - Questions?

https://triton.quarkslab.com
https://github.com/JonathanSalwan/Tigress_protection