How Triton can help to reverse virtual machine based software protections

How to don't kill yourself when you reverse obfuscated codes.

Jonathan Salwan and Romain Thomas

CSAW SOS in NYC, November 10, 2016

About us

Romain Thomas

- Security Research Engineer at Quarkslab
- Working on obfuscation and software protection

Jonathan Salwan

- Security Research Engineer at Quarkslab
- Working on program analysis and software verification

Roadmap of this talk

Part 1 Short introduction to the Triton framework

Part 2 Short introduction to virtual machine based software protections

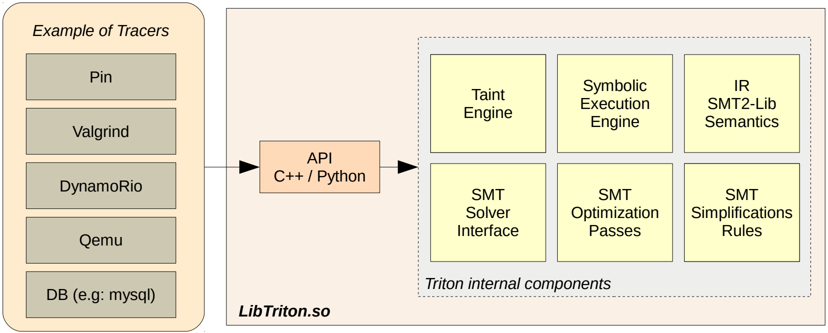
Part 3 Demo - Triton vs VMs

The Triton framework [6]

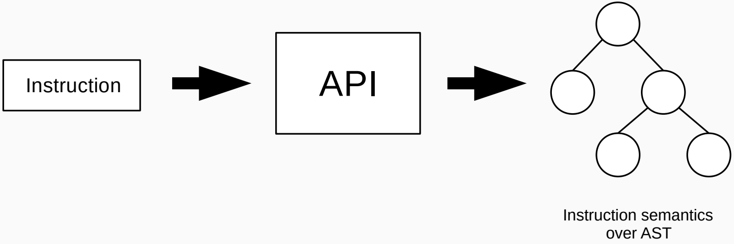
Triton in a nutshell

- A Dynamic Binary Analysis Framework
- Deals with the Intel `x86` and `x86_64` Instruction Set Architecture (*ISA*)
- Contains:
    - Dynamic Symbolic Execution (*DSE*) engine [4, 7]
    - Taint analysis engine
    - Emulation engine
    - Representation of the *ISA* behaviour into an Abstract Syntax Tree (*AST*)
    - *AST* simplification engine
    - Two syntax representations of the *AST*
        - Python
        - SMT2

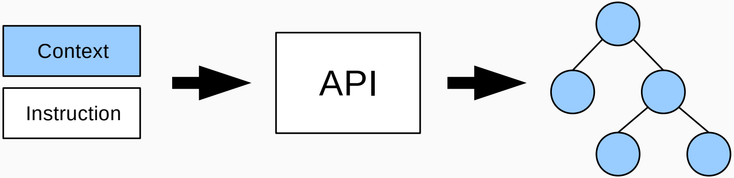
Triton's design



The API's input - Opcode to semantics

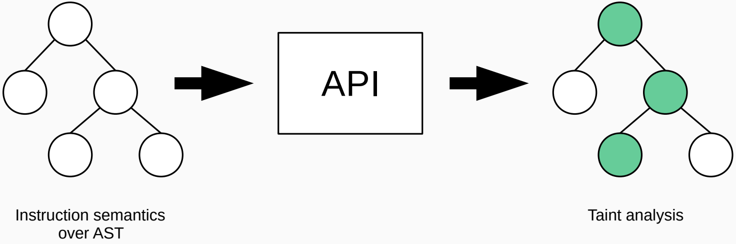


The API's input - Semantics with a context

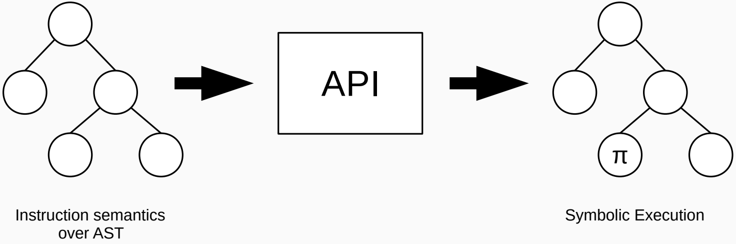


Instruction semantics
over AST

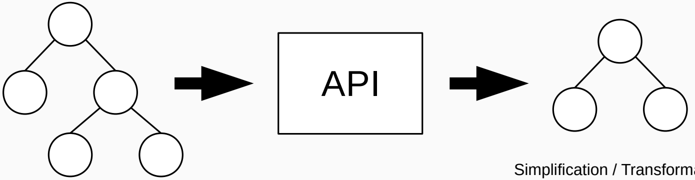
The API's input - Taint Analysis



The API's input - Symbolic Execution



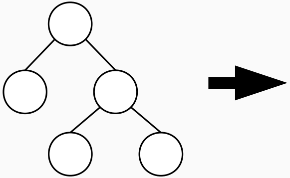
The API's input - Simplification / Transformation



Instruction semantics
over AST

Simplification / Transformation

The API's input - AST representations

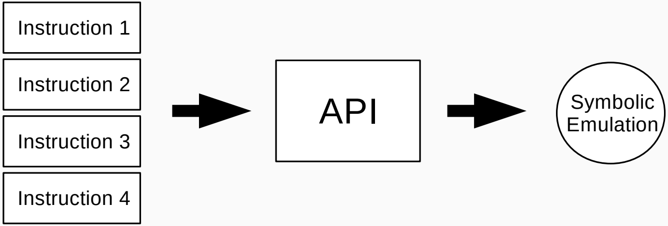


Instruction semantics
over AST

(bvadd (_ bv1 8) (_ bv2 8))

((0x1 + 0x2) & 0xFF)

The API's input - Symbolic Emulation



Example - How to define an opcode and context

```
>>> inst = Instruction("\x48\x31\xD0") # xor rax, rdx

>>> inst.setAddress(0x400000)
>>> inst.updateContext(Register(REG.RAX, 0x1234))
>>> inst.updateContext(Register(REG.RDX, 0x5678))

>>> processing(inst)
```

Example - How to get semantics expressions

```
>>> processing(inst)

>>> print inst
400000: xor rax, rdx

>>> for expr in inst.getSymbolicExpressions():
...     print expr
...
ref_0 = (0x1234 ^ 05678) # XOR operation
ref_1 = 0x0 # Clears carry flag
ref_2 = 0x0 # Clears overflow flag
ref_3 = ((0x1 ^ [... skipped ...] & 0x1)) # Parity flag
ref_4 = ((ref_0 >> 63) & 0x1) # Sign flag
ref_5 = (0x1 if (ref_0 == 0x0) else 0x0) # Zero flag
ref_6 = 0x400003 # Program Counter
```

Example - How to get implicit and explicit read registers

```
>>> for r in inst.getReadRegisters():
...     print r
...
(rax:64 bv[63..0], 0x1234)
(rdx:64 bv[63..0], 0x5678)
```

Example - How to get implicit and explicit written registers

```
>>> for w in inst.getWrittenRegisters():
...     print w
...
(rax:64 bv[63..0], (0x1234 ^ 0x5678))
(rip:64 bv[63..0], 0x400003)
(cf:1 bv[0..0], 0x0)
(of:1 bv[0..0], 0x0)
(pf:1 bv[0..0], ... skipped ...)
(sf:1 bv[0..0], ((ref_0 >> 63) & 0x1))
(zf:1 bv[0..0], (0x1 if (ref_0 == 0x0) else 0x0))
```

To resume: What kind of information can I get from an instruction?

- All implicit and explicit semantics of an instruction
  - `GET`, `PUT`, `LOAD`, `STORE`

- Semantics (side effects included) representation via an abstract syntax tree based on the Static Single Assignment (SSA) form

What about emulation?

```
>>> inst1 = Instruction("\x48\xc7\xc0\x05\x00\x00\x00") # mov rax, 5
>>> inst2 = Instruction("\x48\x83\xC0\x02")             # add rax, 2

>>> processing(inst1)
>>> processing(inst2)

>>> getFullAstFromId(getSymbolicRegisterId(REG.RAX))
((0x5 + 0x2) & 0xFFFFFFFFFFFFFFFF)

>>> getAstFromId(getSymbolicRegisterId(REG.RAX)).evaluate()
7L
```

Ok, but what can I do with all of this?

- Use taint analysis to help during reverse engineering
- Use symbolic execution to cover code
- Use symbolic execution to know what value(s) can hold a register or memory cell
- Simplify expressions for deobfuscation
- Transform expressions for obfuscation
- Match behaviour models for vulnerabilities research
- Be imaginative :)

Mmmmh, and where instruction sequences can come from?

- From dynamic tracers like Pin, Valgrind, Qemu, ...
- From a memory dump
- From static tools like IDA or whatever...

Cool, but how many instruction semantics are supported by Triton?

- **Development:**
  - 256 Intel x86_64 instructions [1]
  - Included 116 SSE/MMX/AVX instructions
- **Testing:**
  - The tests suite [2] of the Qemu TCG [3]
  - Traces differential [4]

---

[1] http://triton.quarkslab.com/documentation/doxygen/SMT_Semantics_Supported_page.html
[2] http://github.com/qemu/qemu/tree/master/tests/tcg
[3] http://wiki.qemu.org/Documentation/TCG
[4] http://triton.quarkslab.com/blog/What-kind-of-semantics-information-Triton-can-provide/#4

Virtual Machine Based Software Protections

VM Based Software Protections

Definition:

It's a kind of obfuscation which transforms an original instruction set (e.g. `x86`) into another custom instruction set (VM implementation).

Example: Virtualization

```
mov rax, 0x123456
```

```
push 0x1 # rax_id
push 0x123456
call VM_MOVE
```

```
and rax, rbx
```

```
push rbx
push rax
mov rcx, [rsp]
mov rdx, [rsp - 0x4]
and rcx, rdx
mov rax, rcx
```

```
call func
```

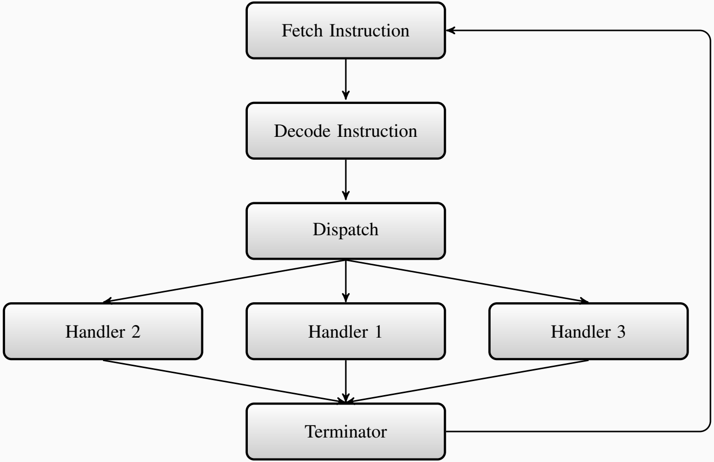
```
mov rbx, 0x1
call trampoline
```

Where are VMs

- Languages: Python, Java...
- Obfuscator: VM Protect [5], Tigress [6] [1, 3], Denuvo [7]
- Malwares: Zeus [8]
- CTF...

[5] http://vmpsoft.com/
[6] http://tigress.cs.arizona.edu/
[7] http://www.denuvo.com/
[8] http://www.miasm.re/blog/2016/09/03/zeusvm_analysis.html

VM abstract architecture



VM abstract architecture

Fetch Instruction:

Fetch the instruction which will be executed by the VM.

Decode Instruction:

Decode the instruction according to the VM instruction set.

Example:

`decode(01 11 12):`

- Opcode: `0x01`
- Operand 1: `0x11`
- Operand 2: `0x12`

VM abstract architecture

Dispatcher:

Jump to the right handler according to opcode and/or operands.

Handlers:

Handlers are the implementation of the VM instruction set.

For instance, the handler for the instruction

`mov REG, IMM`

could be:

```
xor REG, REG
or REG, IMM
```

Terminator:

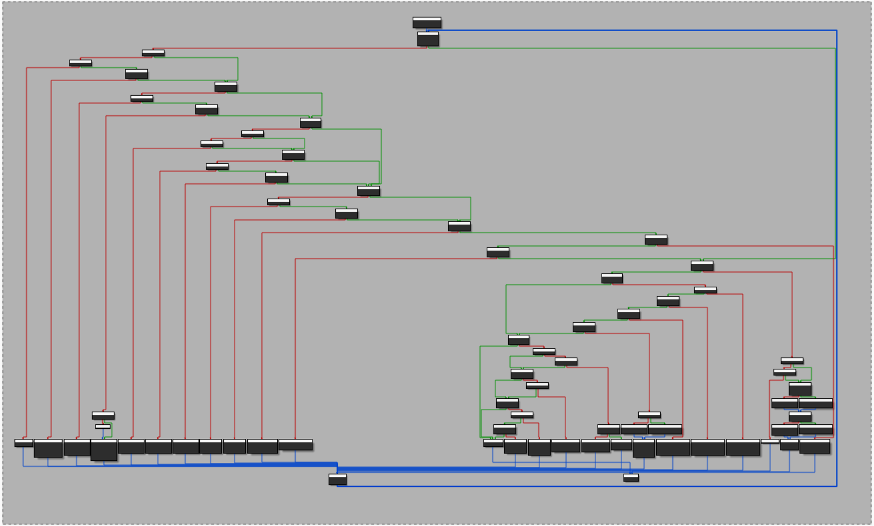
Finishes the VM execution or continues its execution.

Dispatcher

We can have two kinds of dispatcher:

- switch case like
- jump table

A switch case like dispatcher

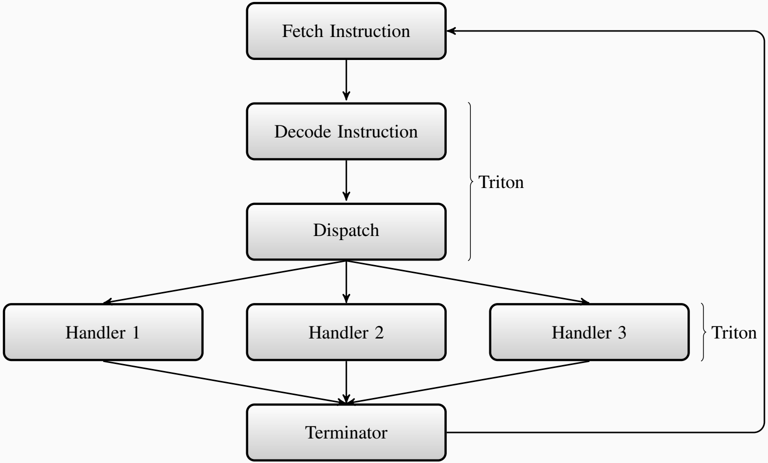


A jump table based dispatcher

```
00000000004147D3
00000000004147D3 loc_4147D3:
00000000004147D3 mov     eax, [ebp+var_50]
00000000004147D6 movzx   eax, byte ptr [eax]
00000000004147D9 lea     ecx, [ebp+var_50]
00000000004147DC call    off_427018[eax*4]
00000000004147E3 test    al, al
00000000004147E5 jnz     short loc_4147D3
```

```
dd offset sub_40EDE0     ; DATA XR
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
dd offset sub_40F1E2
dd offset sub_40F208
dd offset sub_40F230
dd offset sub_40F257
dd offset sub_40F27F
dd offset sub_40F2AF
dd offset sub_40F348
dd offset sub_40F379
```

Using Triton to reverse a VM



Demo: Tigress VM

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.	Open
0002	One level of virtualization, bogus functions, implicit flow.	5	3	script	Signed copy of Surreptitious Software.	Open
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.	Open
0004	Two levels of virtualization, implicit flow.	5	4	script	USD 100.00	Open
0005	One level of virtualization, one level of jitting, implicit flow.	5	4	script	USD 100.00	Open
0006	Two levels of jitting, implicit flow.	5	4	script	USD 100.00	Open

Tigress challenges

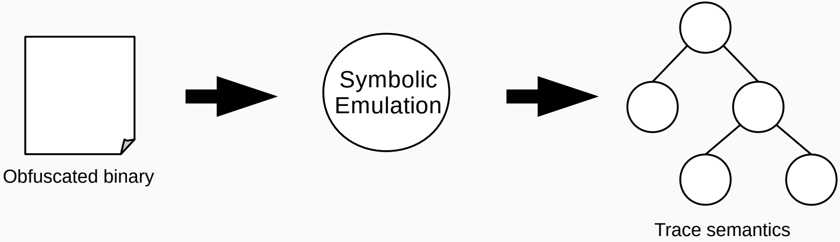
```
$ ./tigress-challenge 1234
3920664950602727424

$ ./tigress-challenge 326423564
16724117216240346858
```

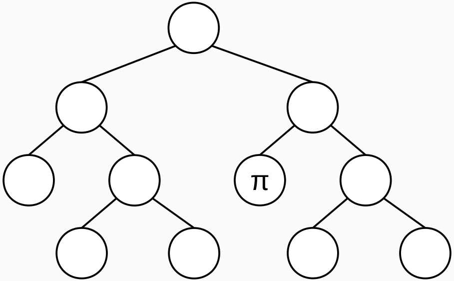
Tigress challenges

Problem: Given a *very secret* algorithm obfuscated with a VM. How can we recover the algorithm without fully reversing the VM?

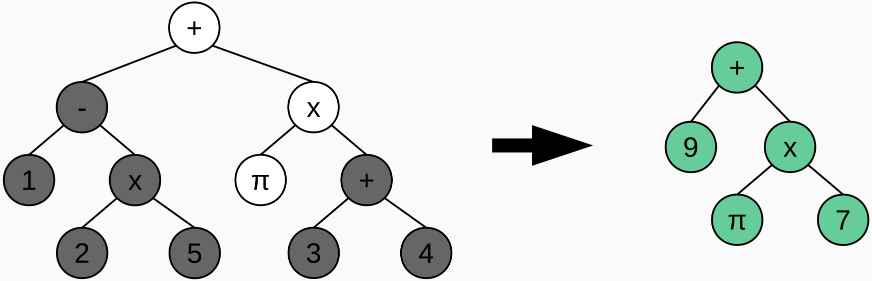
Step 1: Symbolically emulate the binary



Step 2: Define the user input as symbolic variable

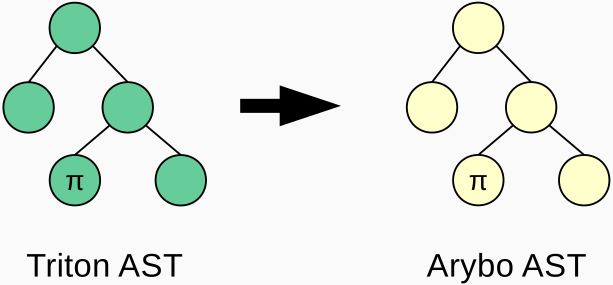


Step 3: Concretize everything which is not related to user input

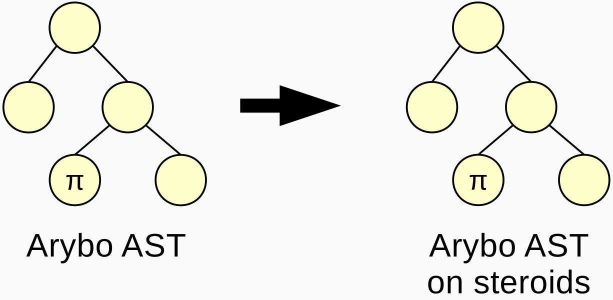


Step 4: Use a better canonical representation of expressions

- Arybo [2] uses the Algebraic Normal Form (ANF) representation

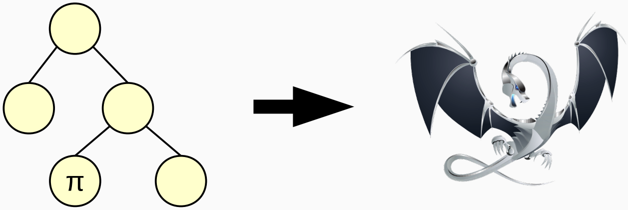


Step 5: Possible use of symbolic simplifications



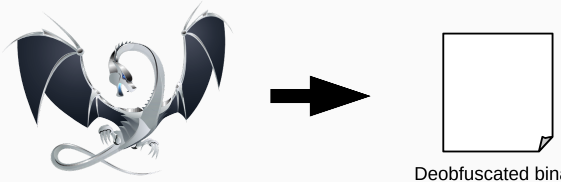
[8] https://pythonhosted.org/arybo/concepts.html

Step 6: From Arybo to LLVM-IR



Arybo AST LLVM-IR

Step 7: Recompile with -O2 optimization and win!



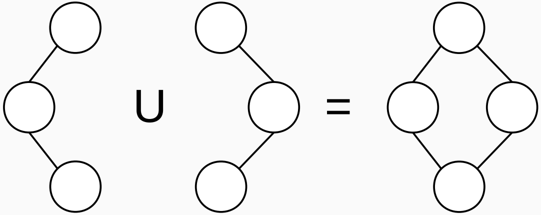
Deobfuscated binary

LLVM-IR

Results with only one trace

	Challenge-0	Challenge-1	Challenge-2	Challenge-3	Challenge-4
VM 0	100.00%	100.00%	34.70%	100.00%	89.60%
VM 1	100.00%	62.55%	100.00%	100.00%	100.00%
VM 2	53.83%	70.25%	100.00%	76.55%	100.00%
VM 3	100.00%	26.35%	92.12%	100.00%	100.00%
VM 4	97.90%	100.00%	79.62%	100.00%	100.00%
VM 5	not analyzed	not analyzed	not analyzed	not analyzed	not analyzed
VM 6	not analyzed	not analyzed	not analyzed	not analyzed	not analyzed
F	Full expressions of the hash algorithm extracted **with** 100.00% of success				
P	Partial expressions of the hash algorithm extracted **without** 100.00% of success				

Cover paths to reconstruct the CFG



Results with the union of two traces

	Challenge-0	Challenge-1	Challenge-2	Challenge-3	Challenge-4
VM 0	100.00%	100.00%	loop on input	100.00%	100.00%
VM 1	100.00%	100.00%	100.00%	100.00%	100.00%
VM 2	loop on input	100.00%	100.00%	100.00%	100.00%
VM 3	100.00%	100.00%	100.00%	100.00%	100.00%
VM 4	100.00%	100.00%	100.00%	100.00%	100.00%
VM 5	not analyzed	not analyzed	not analyzed	not analyzed	not analyzed
VM 6	not analyzed	not analyzed	not analyzed	not analyzed	not analyzed
F	Full expressions of the hash algorithm extracted **with** 100.00% of success				
P	Partial expressions of the hash algorithm extracted **without** 100.00% of success. Loops on input are not trivial to reconstruct – we need more time to work on it.				

Time of extraction per trace

	Challenge-0	Challenge-1	Challenge-2	Challenge-3	Challenge-4
VM 0	3.85 seconds	9.20 seconds	3.27 seconds	4.26 seconds	1.58 seconds
VM 1	1.26 seconds	1.42 seconds	3.27 seconds	2.49 seconds	1.74 seconds
VM 2	6.58 seconds	2.02 seconds	2.63 seconds	4.85 seconds	3.82 seconds
VM 3	45.59 seconds	11.30 seconds	8.84 seconds	4.84 seconds	21.64 seconds
VM 4	361 seconds	315 seconds	588 seconds	8040 seconds	1680 seconds
	Few seconds to extract the equation and less than 200 MB of RAM used				
	Few minutes to extract the equation and ~4 GB of RAM used				
	Few minutes to extract the equation and ~5 GB of RAM used				
	Few minutes to extract the equation and ~9 GB of RAM used				
	Few minutes to extract the equation and ~21 GB of RAM used				
	Few hours to extract the equation and ~170 GB of RAM used				

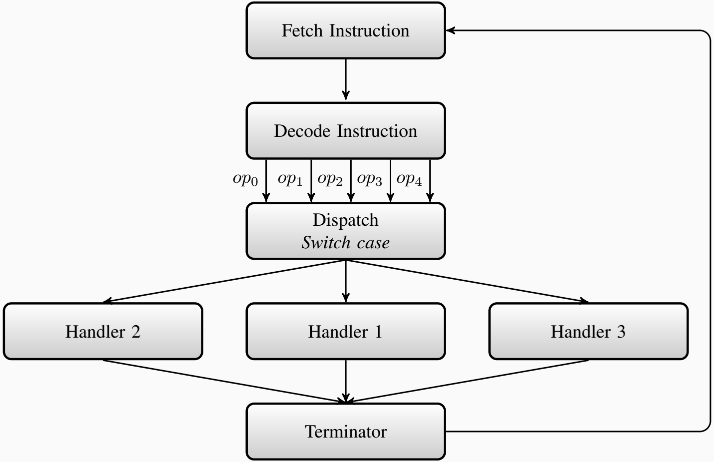
Let me try by myself

Release: Everything related to this analysis is available on github [9].

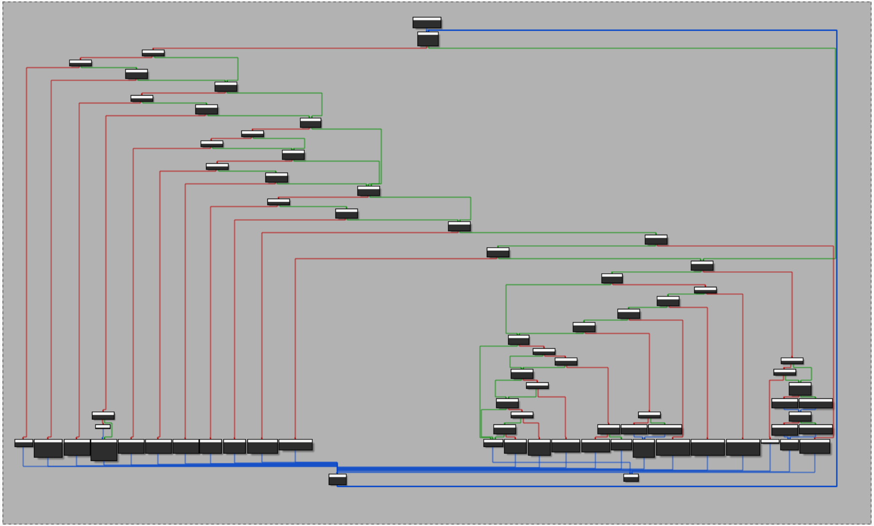
[9] https://github.com/JonathanSalwan/Tigress_protection

Demo: Unknown VM

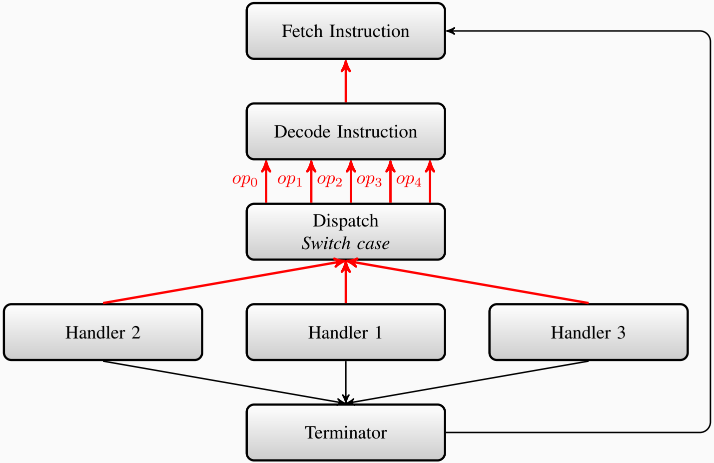
VM Architecture



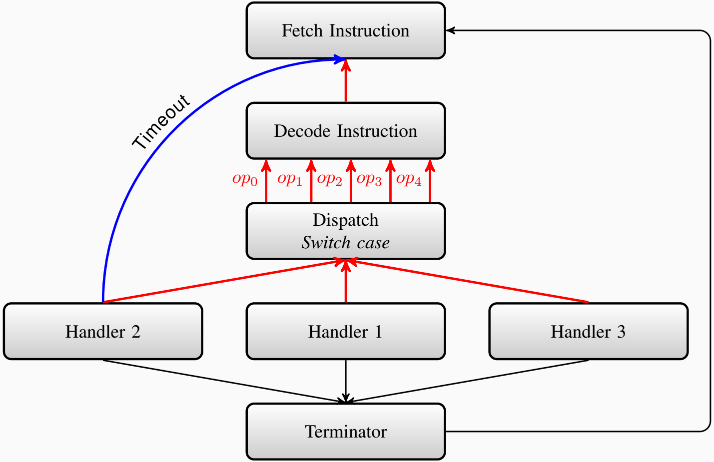
VM Architecture



Goal

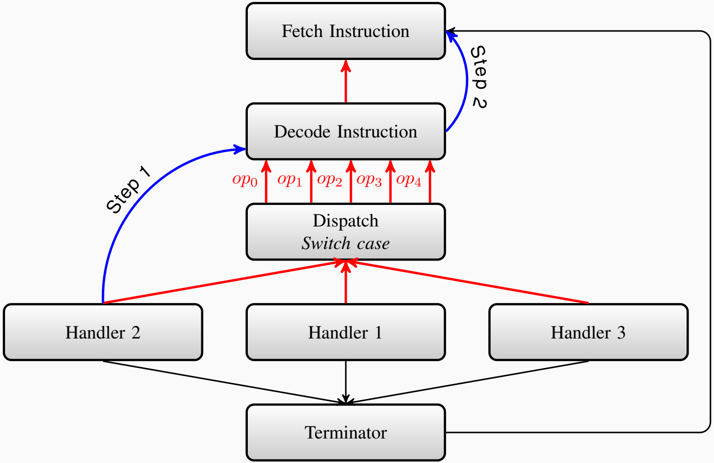

Fetch Instruction
Decode Instruction
op_0
op_1
op_2
op_3
op_4
Dispatch
Switch case
Handler 2
Handler 1
Handler 3
Terminator

Goal

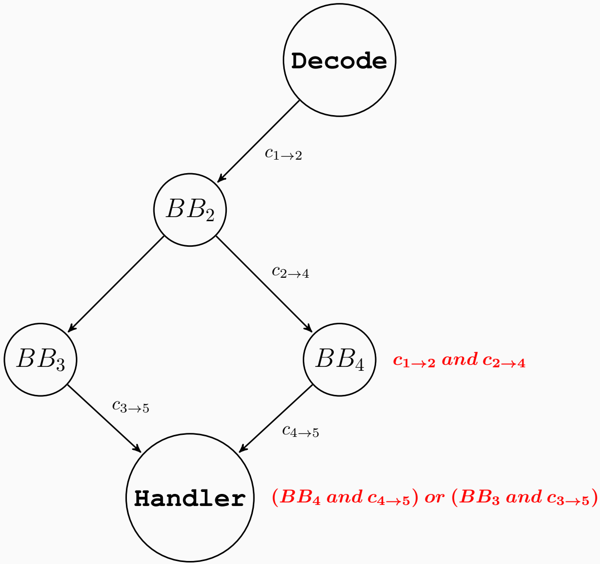


Fetch Instruction
Timeout
Decode Instruction
op_0
op_1
op_2
op_3
op_4
Dispatch
Switch case
Handler 2
Handler 1
Handler 3
Terminator

Goal


Fetch Instruction
Step 2
Decode Instruction
Step 1
op_0 op_1 op_2 op_3 op_4
Dispatch
Switch case
Handler 2
Handler 1
Handler 3
Terminator

Goal


Decode
$c_{1\to2}$
BB_2
$c_{2\to4}$
BB_3
BB_4
$c_{1\to2}$ and $c_{2\to4}$
$c_{3\to5}$
$c_{4\to5}$
Handler
$(BB_4$ and $c_{4\to5})$ or $(BB_3$ and $c_{3\to5})$

Conclusion

Conclusion

- Symbolic execution is powerful against obfuscations
- Use mathematical complexity expressions against such attacks
  - The goal is to imply a timeout on SMT solvers side

Thanks

Any Questions?

Acknowledgements

- Thanks to Brendan Dolan-Gavitt for his invitation to the S.O.S workshop!
- Kudos to Adrien Guinet for his Arybo [10] framework!

[10] https://github.com/quarkslab/arybo

Contact us

- **Romain Thomas**
  - `rthomas at quarkslab com`
  - `@rh0main`

- **Jonathan Salwan**
  - `jsalwan at quarkslab com`
  - `@JonathanSalwan`

- **Triton team**
  - `triton at quarkslab com`
  - `@qb_triton`
  - `irc: #qb_triton@freenode.org`

References I

C. Collberg, S. Martin, J. Myers, and J. Nagra.
Distributed application tamper detection via continuous software updates.
In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 319–328, New York, NY, USA, 2012. ACM.

N. Eyrolles, A. Guinet, and M. Videau.
Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions.
GreHack, France, Grenoble, 2016.

Y. Kanzaki, A. Monden, and C. Collberg.
Code artificiality: A metric for the code stealth based on an n-gram model.
In *SPRO 2015 International Workshop on Software Protection*, 2015.

References II

J. C. King.
Symbolic execution and program testing.
Communications of the ACM, 19(7):385–394, 1976.

C. Lattner and V. Adve.
LLVM: A compilation framework for lifelong program analysis and transformation.
pages 75–88, San Jose, CA, USA, Mar 2004.

References III

F. Saudel and J. Salwan.
Triton: A dynamic symbolic execution framework.
In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.

K. Sen, D. Marinov, and G. Agha.
Cute: a concolic unit testing engine for c.
In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.