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
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

**Example of Tracers**
- Pin
- Valgrind
- DynamoRio
- Qemu
- DB (e.g: mysql)

**Triton internal components**
- API
  - C++ / Python
- Taint Engine
- Symbolic Execution Engine
- IR SMT2-Lib Semantics
- SMT Solver Interface
- SMT Optimization Passes
- SMT Simplifications Rules

**LibTriton.so**
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

Instruction semantics over AST

Taint analysis
The API’s input - Symbolic Execution

Instruction semantics over AST → API → Symbolic Execution

π
The API's input - Simplification / Transformation

Instruction semantics over AST

API

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

Instruction 1
Instruction 2
Instruction 3
Instruction 4

API

Symbolic Emulation
Example - How to define an opcode and context

```python
>>> 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 ^ [...] & 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

```python
>>> 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

```python
>>> 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) & 0xFFFFFFFFFFFFFFF)

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

```assembly
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

Fetch Instruction

Decode Instruction

Dispatch

Handler 2

Handler 1

Handler 3

Terminator
**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
\[
\text{mov REG, IMM}
\]
could be:
\[
\text{xor REG, REG}
\]
or
\[
\text{or REG, IMM}
\]

**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
Using Triton to reverse a VM
Demo: Tigress VM
<table>
<thead>
<tr>
<th>Challenge</th>
<th>Description</th>
<th>Number of binaries</th>
<th>Difficulty (1-10)</th>
<th>Script Prize</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>One level of virtualization, random dispatch.</td>
<td>5</td>
<td>1</td>
<td>script Certificate issued by DAPA</td>
<td>Solved</td>
</tr>
<tr>
<td>0001</td>
<td>One level of virtualization, superoperators, split instruction handlers.</td>
<td>5</td>
<td>2</td>
<td>script Signed copy of Suspeptitious Software</td>
<td>Open</td>
</tr>
<tr>
<td>0002</td>
<td>One level of virtualization, bogus functions, implicit flow.</td>
<td>5</td>
<td>3</td>
<td>script Signed copy of Suspeptitious Software</td>
<td>Open</td>
</tr>
<tr>
<td>0003</td>
<td>One level of virtualization, instruction handlers obfuscated with arithmetic encoding, virtualized function is split and the split parts merged.</td>
<td>5</td>
<td>2</td>
<td>script Signed copy of Suspeptitious Software</td>
<td>Open</td>
</tr>
<tr>
<td>0004</td>
<td>Two levels of virtualization, implicit flow.</td>
<td>5</td>
<td>4</td>
<td>script USD 100.00</td>
<td>Open</td>
</tr>
<tr>
<td>0005</td>
<td>One level of virtualization, one level of jitting, implicit flow.</td>
<td>5</td>
<td>4</td>
<td>script USD 100.00</td>
<td>Open</td>
</tr>
<tr>
<td>0006</td>
<td>Two levels of jitting, implicit flow.</td>
<td>5</td>
<td>4</td>
<td>script USD 100.00</td>
<td>Open</td>
</tr>
</tbody>
</table>
Tigress challenges

$ ./tigress-challenge 1234
3920664950602727424

$ ./tigress-challenge 326423564
16724117216240346858
**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

Arybo AST

Arybo AST on steroids

https://pythonhosted.org/arybo/concepts.html
Step 6: From Arybo to LLVM-IR

Arybo AST  \[\pi\]  LLVM-IR
Step 7: Recompile with -O2 optimization and win!

LLVM-IR

Deobfuscated binary
# Results with only one trace

<table>
<thead>
<tr>
<th></th>
<th>Challenge-0</th>
<th>Challenge-1</th>
<th>Challenge-2</th>
<th>Challenge-3</th>
<th>Challenge-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM 0</td>
<td>100.00%</td>
<td>100.00%</td>
<td>34.70%</td>
<td>100.00%</td>
<td>89.60%</td>
</tr>
<tr>
<td>VM 1</td>
<td>100.00%</td>
<td>62.55%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>VM 2</td>
<td>53.83%</td>
<td>70.25%</td>
<td>100.00%</td>
<td>76.55%</td>
<td>100.00%</td>
</tr>
<tr>
<td>VM 3</td>
<td>100.00%</td>
<td>26.35%</td>
<td>92.12%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>VM 4</td>
<td>97.90%</td>
<td>100.00%</td>
<td>79.62%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>VM 5</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
</tr>
<tr>
<td>VM 6</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th></th>
<th>Challenge-0</th>
<th>Challenge-1</th>
<th>Challenge-2</th>
<th>Challenge-3</th>
<th>Challenge-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM 0</td>
<td>100.00%</td>
<td>100.00%</td>
<td>loop on input</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>VM 1</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>VM 2</td>
<td>loop on input</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>VM 3</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>VM 4</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>VM 5</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
</tr>
<tr>
<td>VM 6</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
<td>not analyzed</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Challenge-0</th>
<th>Challenge-1</th>
<th>Challenge-2</th>
<th>Challenge-3</th>
<th>Challenge-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM 0</td>
<td>3.85 seconds</td>
<td>9.20 seconds</td>
<td>3.27 seconds</td>
<td>4.26 seconds</td>
<td>1.58 seconds</td>
</tr>
<tr>
<td>VM 1</td>
<td>1.26 seconds</td>
<td>1.42 seconds</td>
<td>3.27 seconds</td>
<td>2.49 seconds</td>
<td>1.74 seconds</td>
</tr>
<tr>
<td>VM 2</td>
<td>6.58 seconds</td>
<td>2.02 seconds</td>
<td>2.63 seconds</td>
<td>4.85 seconds</td>
<td>3.82 seconds</td>
</tr>
<tr>
<td>VM 3</td>
<td>45.59 seconds</td>
<td>11.30 seconds</td>
<td>8.84 seconds</td>
<td>4.84 seconds</td>
<td>21.64 seconds</td>
</tr>
<tr>
<td>VM 4</td>
<td>361 seconds</td>
<td>315 seconds</td>
<td>588 seconds</td>
<td>8040 seconds</td>
<td>1680 seconds</td>
</tr>
</tbody>
</table>

- 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.

⁹https://github.com/JonathanSalwan/Tigress_protection
Demo: Unknown VM
VM Architecture

Fetch Instruction
→
Decode Instruction

$op_0$  $op_1$  $op_2$  $op_3$  $op_4$

Dispatch
Switch case

Handler 1
Handler 2
Handler 3

Terminator
Goal

Fetch Instruction

<table>
<thead>
<tr>
<th>Decode Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$op_0$</td>
</tr>
</tbody>
</table>

Dispatch
*Switch case*

Handler 2

Handler 1

Handler 3

Terminator
Goal

Fetch Instruction

Decode Instruction

Dispatch

Switch case

Handler 1

Handler 2

Handler 3

Terminator

op0 op1 op2 op3 op4

Timeout

54
Goal

Fetch Instruction
Decode Instruction
Dispatch
Switch case
Handler 1
Handler 2
Handler 3
Terminator

Step 1: op0, op1, op2, op3, op4
Step 2:

op0, op1, op2, op3, op4

Step 1
Step 2

Handler 2
Handler 1
Handler 3
Terminator

op0, op1, op2, op3, op4

Step 1
Step 2
Goal

Figure 1: CFG switch case representation

- **Decode**
  - $c_1 \rightarrow 2$ and $c_2 \rightarrow 4$
  - $(BB_4$ and $c_4 \rightarrow 5)$ or $(BB_3$ and $c_3 \rightarrow 5)$
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
**Distributed application tamper detection via continuous software updates.**  

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

Y. Kanzaki, A. Monden, and C. Collberg.  
**Code artificiality: A metric for the code stealth based on an n-gram model.**  
J. C. King.

**Symbolic execution and program testing.**


C. Lattner and V. Adve.

**LLVM: A compilation framework for lifelong program analysis and transformation.**

pages 75–88, San Jose, CA, USA, Mar 2004.
F. Saudel and J. Salwan.

**Triton: A dynamic symbolic execution framework.**


K. Sen, D. Marinov, and G. Agha.

**Cute: a concolic unit testing engine for c.**