Dynamic Binary Analysis and Obfuscated Codes

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

Jonathan Salwan and Romain Thomas
St’Hack in Bordeaux, April 8, 2016
Quarkslab
About us

Romain Thomas

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

Jonathan Salwan

- Security Research Engineer at Quarkslab
- Ph.D student on Software Verification supervised by:
  - Sébastien Bardin from CEA
  - Marie-Laure Potet from VERIMAG
Roadmap of this talk

1. Obfuscation introduction
2. Dynamic Binary Analysis introduction
3. The Triton framework
4. Conclusion
5. Future works
Obfuscation Introduction
What is an obfuscation?

Wikipedia: "Obfuscation is the obscuring of intended meaning in communication, making the message confusing, willfully ambiguous, or harder to understand." ¹

¹https://en.wikipedia.org/wiki/Obfuscation
Why softwares may contain obfuscated codes?

- Intellectual property
- DRM
- Hiding secrets
What kind of obfuscations may we find in modern softwares?

- Opaque predicates
- Control-flow flattening
- Virtualization
- MBA and bitwise operations
- Use of uncommon instructions.
Example: Opaque predicates

- Objective: Create unreachable basic blocks
- The constraint \( \neg \pi_1 \) is always UNSAT
- The basic block \( \phi_3 \) is never executed

**Figure 1:** Control Flow Graph
Example: CFG flattened

- Objective: Remove structured control flows
- The basic block $\varphi_2$ is now used as dispatcher
- The dispatcher manages the control flow
  - Static analysis: hard to predict which basic block will be called next

**Figure 2:** Original CFG

**Figure 3:** Flattened CFG
Objective: Emulate the original code via a custom ISA (Instruction Set Architecture)

Example:

```assembly
xor R1, R2
push R1
push R2
mov eax, [esp]
mov ebx, [esp - 0x4]
xor eax, ebx
push eax
```
Figure 4: An example of a VM’s CFG
Example: MBA and bitwise operations

- Objective: Transform the normal form of an expression to a more complex one
- The transformation output may also be transformed again and so one

\[
\begin{align*}
a + b &= (a \lor b) + (a \land b) \\
a \ast b &= (a \land b) \ast (a \lor b) + (a \land \neg b) \ast (\neg a \land b) \\
a \oplus b &= (a \land \neg b) \lor (\neg a \land b) \\
a \oplus b &= ((a \land \neg a) \land (\neg b \lor \neg a)) \land ((a \lor b) \lor (\neg b \lor b)) \\
0 &= (a \lor b) - (a + b) + (a \land b)
\end{align*}
\]
Example: Use of uncommon instructions

- **Objective:**
  - Break your tools
  - Break your mind!
- May transform classic operations using AVX and SSE

```assembly
public foo
proc near
  vmovd xmm0, esi
  vpxor xmm1, xmm1, xmm1
  vpshufb xmm0, xmm6, xmm1
  vpunpcklbw xmm0, xmm0, xmm1
  vpmovzxwd xmm0, xmm0
  vpaddd xmm0, xmm0, xmm0
  vmovd xmm2, edi
  vpshufb xmm2, xmm2, xmm1
  vpunpcklbw xmm1, xmm2, xmm1
  vpmovzxwd xmm1, xmm1
  vpaddd xmm1, xmm1, xmm1
  vpandn xmm1, xmm0, xmm1
  vpblendw xmm0, xmm1, xmm0, 0ccch
  vmovshdup xmm1, xmm0
  vpermilpd xmm2, xmm0, 1
  vpermilps xmm0, xmm3, 0ef7h
  vcmpps xmm0, xmm3, xmm0
  vcmpps xmm0, xmm6, xmm1
  vcmpps xmm0, xmm0, xmm2
  vmovd eax, xmm0
  vmovd xmm0, eax
  vpshufd xmm0, xmm0, 0
  vmovd eax, xmm0
  vpextrd ecx, xmm0, 1
  vpextrd edx, xmm0, 2
  sub eax, edx
  add eax, edx
  movzx eax, al
  ret
```

**Figure 5:** Uncommon instructions
Dynamic Binary Analysis Introduction
What is a DBA?

- **Dynamic Binary Analysis**
  - Any way to analyze a binary dynamically
  - Most popular analysis
    - Dynamic information extraction
    - Dynamic taint analysis [4]
    - Dynamic symbolic execution [3, 2, 6, 1]
Why use a DBA?

- To get runtime values at each program point
- To get the control flow for a given input
- To follow the spread of a specific data
What is a dynamic taint analysis?

- Taint analysis is used to follow a specific information through a data flow
  - Cell memory
  - Register
- The taint is spread at runtime
- At each program point you are able to know what cells and registers interact with your initial value
What is a dynamic symbolic execution?

- A DSE is used to represent the control and the data flow of an execution into arithmetical expressions
- These expressions may contain symbolic variables instead of concrete values
- Using a SMT solver\(^2\) on these expressions, we are able to determine an input for a desired state

\(^2\)https://en.wikipedia.org/wiki/Satisfiability_modulo_theories#SMT_solvers
\(^3\)http://smtlib.cs.uiowa.edu
SBA vs DBA

- **Static Binary Analysis**
  - Full CFG
  - No concrete value
  - Often based on abstract analysis
    - Scalable
    - False positive
  - Too complicated for analyze obfuscated code

- **Dynamic Binary Analysis**
  - Partial CFG (only one path at time)
  - Concrete values
  - Often based on concrete analysis
    - Not scalable
    - Less false positive
  - Lots of static protections may be broken
Online vs offline analysis

- **Online analysis**
  - Extract runtime information
  - Inject runtime values
  - Interact and modify the control flow
  - **Good for fuzzing**

- **Offline analysis**
  - Store the context of each program point into a database
  - Apply post analysis
  - Display the context information using both static and dynamic paradigms
  - **Good for reverse**
Offline analysis good for reverse

**Figure 6:** Example of an offline analysis infrastructure
 Offline analysis and symbolic emulation

- Explore more than one path using symbolic emulation from a concrete path
  - From one path emulate them all

Figure 7: Concrete execution
• Keep both concrete and symbolic values of each symbolic variable
• Use the concrete value for the emulation part and the symbolic value for expressions and models
• Get the model of the new branch and restore the concrete value of the symbolic variable

Figure 8: Symbolic emulation from a concrete path
• Concrete and emulated paths are merged with disjunctions to get a coverage expression

\[ \varphi_1 \land (\varphi_3 \lor \varphi_4) \land \varphi_5 \land \varphi_6 \]

**Figure 9:** Disjunction of paths
The Triton [5] framework
Triton in a nutshell

- **Dynamic Binary Analysis Framework**
  - x86 and x86_64 binaries analysis
  - Dynamic Taint Analysis
  - Dynamic Symbolic Execution
  - Partial Symbolic Emulation
  - Python or SMT semantics representation
  - Simplification passes
  - Python and C++ API

- **Tracer independent**
  - A Pintool \(^4\) is shipped with the project

- **Free and opensource** \(^5\)

\(^4\)https://software.intel.com/en-us/articles/pintool/
\(^5\)http://triton.quarkslab.com
The Triton’s design

Figure 10: The Triton’s design
The Triton’s design

- **libpintool.so**
  - Used as tracer to give the execution context to the Triton library
  - Python bindings on some Pin’s features

- **libtriton.so**
  - Takes as input opcodes and a potential context
  - Contains all engines and analysis
  - Python and C++ API
In what scenarios should I use Triton?

- If I want to use basic Pin’s features with Python bindings
- If I’m working on a trace and want to perform a taint or symbolic analysis
- If I want to simplify expressions using my own rules or those of z3⁶

⁶https://github.com/Z3Prover/z3
The classic count_inst example

count = 0
def mycb(inst):
    global count
    count += 1
def fini():
    print count

if __name__ == '__main__':
    setArchitecture(ARCH.X86_64)
    startAnalysisFromEntry()
    addCallback(mycb, CALLBACK.BEFORE)
    addCallback(fini, CALLBACK.FINI)
    runProgram()
Can I use the libTriton into IDA?

Figure 11: Triton and RPyC ⁷

⁷https://rpyc.readthedocs.org
Can I emulate code via the libTriton into IDA?

```python
import classic
import triton

inst = triton.Instruction()
opcode = idc.GetManyBytes(pc, idc.ItemSize(pc))
inst.setOpcodes(opcode)
inst.setAddress(pc)
triton.processing(inst)
print(inst)

pc = triton.getSymbolicRegisterValue(triton.REG_RIP)
```

**Figure 12:** Symbolic Emulation into IDA
Simplify expressions
Simplify expressions

- Simplification passes may be applied at different levels:
  - Runtime node assignment (registers, memory cells, volatile)
  - Specific isolated expressions

- Triton allows you to:
  - Apply your own transformation rules based on smart patterns
  - Use z3\(^8\) to apply transformations

\(^8\)(simplify <expr>)
Simplification passes at different levels

Figure 13: Runtime simplification
Simplify expressions with your own rules

Rule example: $A \oplus A \rightarrow A = 0$

```python
def xor(node):
    if node.getKind() == AST_NODE.BVXOR:
        if node.getChilds()[0] == node.getChilds()[1]:
            return bv(0, node.getBitvectorSize())
    return node
```

```python
if __name__ == '__main__':
    # [..]
    recordSimplificationCallback(xor)
    # [..]
```
Smart patterns matching

- Commutativity and patterns matching
  - A smart equality (==) operator

```
>>> a | >>> a | >>> a
((0x1 * 0x2) & 0xFF) | (((0x1 * 0x2) & 0xFF) ^ 0x3) | (0x1 / 0x2)
>>> b | >>> b | >>> b
((0x2 * 0x1) & 0xFF) | (0x3 ^ ((0x2 * 0x1) & 0xFF)) | (0x2 / 0x1)
>>> a == b | >>> a == b | >>> a == b
True | True | False
```
Triton to Z3 and vice versa

Figure 14: $\text{AST}_{\text{triton}} \leftrightarrow \text{AST}_{\text{z3}}$
Simplify expressions via z3

```python
>>> enableSymbolicZ3Simplification(True)
>>> a = ast.variable(newSymbolicVariable(8))
>>> b = ast.bv(0x38, 8)
>>> c = ast.bv(0xde, 8)
>>> d = ast.bv(0x4f, 8)
>>> e = a * ((b & c) | d)
>>> print(e)
(bvmul SymVar_0 (bvor (bvand (_ bv56 8) (_ bv222 8)) (_ bv79 8)))
>>> f = simplify(e)
>>> print(f)
(bvmul (_ bv95 8) SymVar_0)
```
Note that solvers’ simplification does not converge to a more human readable expression.
Analyse opaque predicates
Analyse opaque predicates
Analyse opaque predicates

Figure 15: Bogus Flow Control
Analyse opaque predicates

\[ \forall x, y \ (y < 10 \ \lor \ x(x + 1) \mod 2 == 0) \ \text{is True} \]
Analyse opaque predicates

Convert x and y as symbolic variable;

for basic block in graph do
  for instruction in basic block do
    triton.emulate(instruction);
    if instruction.type is conditionnal jump and zf expression is symbolized then
      Check if zf has solutions ;
    end
  end
end
x_addr = 0x601BE0
y_addr = 0x601BDC

x_symVar = convertMemyToSymVar(Memory(x_addr, CPUSIZE.DWORD))
y_symVar = convertMemyToSymVar(Memory(y_addr, CPUSIZE.DWORD))
graph = idaapi.FlowChart(idaapi.get_func(FUNCTION_ADDRESS))
for block in graph:
    if block.startEA != 0x401637:
        analyse_basic_block(block)
def analyse_basic_block(BB):
    pc = BB.startEA
    while pc <= BB.endEA:
        instruction = triton.emulate(pc)
        pc = triton.getSymbolicRegisterValue(triton.REG.RIP)
        if instruction.isControlFlow():
            break
...

zf_expr = triton.getFullAst(zf_expr.getAst())

eq_false = ast.assert_(ast.equal(zf_expr, ast.bvfalse()))
eq_true = ast.assert_(ast.equal(zf_expr, ast.bvtrue()))
Analyse opaque predicates (3)

```python
models_true = triton.getModels(eq_true, 4)
models_false = triton.getModels(eq_false, 4)

addr_next = instruction.getNextAddress()
addr_jmp = instruction.getFirstOperand().getValue()

if len(models_true) != 0:  # addr_jmp is not taken
    bb = get_basic_block(addr_jmp)
    dead_blocks.append(bb)
if len(models_false) != 0:  # addr_next is not taken
    bb = get_basic_block(addr_next)
    dead_blocks.append(bb)
```
Figure 16: Bogus Flow Control simplified with Triton
First demo!
Reconstruct a CFG from trace differential
Problem: Given two sequences what is the minimal edition distance?

\[ T_1 : A \ T \ C \ T \ G \ A \ T \]
\[ T_2 : A \ A \ T \ C \ T \ G \ A \ T \]
Reconstruct a CFG from trace differential

Levenshtein algorithm (dynamic programming)

\[ T_1 : \text{A} - \text{T C T G A T} \]
\[ T_2 : \text{A A T C T G A T} \]
We can see a trace as a DNA sequence on a bigger alphabet. Many algorithms have been developed to analyze/compare a DNA sequence and they can be used on traces.

- Levenshtein algorithm: optimal alignment, if, else detection
- Suffix Tree: Longest repeated factor, loops detection
int f(int x) {
    int result = 0;
    result = x;
    result = result >> 3;
    if (result % 4 == 2) {
        result += 5;
        result = result + x;
    }
    result = result * 7;
    return result;
}
0x400536 push rbp
0x400537 mov rbp, rsp
0x40053a mov dword ptr [rbp - 0x14], edi
0x400544 mov eax, dword ptr [rbp - 0x14]
0x40054a sar dword ptr [rbp - 0x14], 3
0x40054e mov eax, dword ptr [rbp - 0x14]
0x400551 cdq
0x400552 shr edx, 0x1e
0x400555 add eax, edx
0x400557 and eax, 3
0x40055a sub eax, edx
0x40055c cmp eax, 2
0x40055f jne 0x40056b
0x400561 add dword ptr [rbp - 4], 5
0x400565 mov eax, dword ptr [rbp - 0x14]
0x40056b mov edx, dword ptr [rbp - 4]
0x40056e mov eax, edx
0x400570 shl eax, 3
0x400573 sub eax, edx
0x400575 mov dword ptr [rbp - 4], eax
0x400577 mov eax, dword ptr [rbp - 4]
0x40057b pop rbp
Recover the algorithm of a VM
Problem: Given a very secret algorithm obfuscated with a VM. How can we recover the algorithm without fully reversing the VM?
Recover the algorithm of a VM

$ ./vm 1234
3920664950602727424

$ ./vm 326423564
16724117216240346858
Recover the algorithm of a VM

- The VM is too big to be analyzed statically in few minutes
- One trace gives you all information that you need
Recover the algorithm of a VM

- Use taint analysis to isolate VM’s handlers and their goal

Figure 18: VM handler and a taint analysis
Recover the algorithm of a VM

Triton tool

```python
from triton import *

def sym(instruction):
    if instruction.getAddress() == 0x4099B5:
        taintRegister(REG.RAX)

def before(instruction):
    if instruction.isTainted():
        print instruction

if __name__ == '__main__':
    setArchitecture(ARCH.X86_64)
    startAnalysisFromEntry()
    addCallback(sym, CALLBACK.BEFORE_SYMPROC)
    addCallback(before, CALLBACK.BEFORE)
    runProgram()
```

Output

```assembly
mov rdx, qword ptr [rbp + rax*8 - 0x330]
shr rdx, cl ; First handler, RDX = 1234
mov qword ptr [rbp + rax*8 - 0x330], rdx
mov rax, qword ptr [rbp + rax*8 - 0x330]
mov qword ptr [rdx], rax
    ; All others VM’s handlers
mov rdx, qword ptr [rax]
mov rax, qword ptr [rax]
mov ecx, eax
shl rdx, cl ; Last handler, RDX = 3920664950602727424
mov qword ptr [rbp + rax*8 - 0x330], rdx
```
Recover the algorithm of a VM

- Use symbolic execution to extract the expression of the algorithm
  - Create a script \( \text{input} \leftrightarrow \text{hash} \)
Recover the algorithm of a VM

Triton tool

```python
def sym(instruction):
    if instruction.getAddress() == 0x4099B5:
        convertRegisterToSymbolicVariable(REG.RAX)

def before(instruction):
    if instruction.getAddress() == 0x409A0B:
        raxAst = getFullAst(
            getSymbolicExpressionFromId(
                getSymbolicRegisterId(REG.RAX)
            ).getAst())
        print '
[+] Generating input_to_hash.py.
        fd = open('./input_to_hash.py', 'w')
        fd.write(TEMPLATE_GENERATE_HASH % (raxAst))
        fd.close()

        print '
[+] Generating hash_to_input.py.
        fd = open('./hash_to_input.py', 'w')
        fd.write(TEMPLATE_GENERATE_INPUT % (raxAst))
        fd.close()
```

Output

```
$ ./triton ./solve-vm.py ./vm 1234
[+] Generating input_to_hash.py.
[+] Generating hash_to_input.py.
$ python ./input_to_hash.py 1234
3920664950602727424
$ python ./input_to_hash.py 8347324
15528411515173474176
$ python ./hash_to_input.py 15528411515173474176
...
[SymVar_0 = 2095535]
[SymVar_0 = 2093487]
[SymVar_0 = 2027951]
[SymVar_0 = 2029999]
[SymVar_0 = 2060719]
[SymVar_0 = 2062767]
$ ./vm 2093487
15528411515173474176
$ ./vm 2027951
15528411515173474176
$ ./vm 2060719
15528411515173474176
```
Second demo!
Conclusion
• Lots of static protections may be broken from an unique trace
• Taint and symbolic analysis are really useful when reversing obfuscated code
• The best protection is MBA and bitwise operation
  • Hard to detect patterns automatically
  • Hard to simplify
Future Works
Future Works

- **libTriton**
  - Improve the emulation part
  - Paths and expressions merging
    - Restructured DFG/CFG via a Python representation (WIP #282 #287)
    - Trace differential on DNA-based algorithms
  - Pattern matching via formal proof
  - Internal GC to scale the memory consumption
Thanks
Any Questions?
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
S. Bardin and P. Herrmann.

**Structural testing of executables.**


**Automating software testing using program analysis.**


**DART: directed automated random testing.**


J. Newsome and D. X. Song.

**Dynamic taint analysis for automatic detection, analysis, and signature regeneration of exploits on commodity software.**

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.**