Seeing Through Themida's Code Mutation

Erwan Grelet

June 29th, 2024

About Me

Security researcher at Ubisoft

Interests:

- Reverse Engineering
- Vulnerability Research
- Software Development
- Software Obfuscation

Contacts

- @ergrelet
- @ergrelet@mastodon.social

[0]Disclaimer: this is the result of a personal research project and is not linked to my employer.

Themida

- Commercial **software protector**
- Developed by Oreans Technologies[1]
- **Binary-to-binary** workflow
- Supports **x86 and .NET Windows executables** (EXEs and DLLs)

[1] https://www.oreans.com/

SecureEngine

SecureEngine
ADVANCED ANTI-CRACKING TECHNOLOGY
Advanced Anti-Cracking Techniques
Added to Executable

- Code protection engine used by Themida
- Shared with other Oreans products[2]
- Contains the code mutation engine

[2] Code Virtualizer and WinLicense

Mutation-based Code Obfuscation

In commercial protectors code mutation generally means:

- **No** interpreter or **virtual machine** (VM) involved

Mutation-based Code Obfuscation

In commercial protectors code mutation generally means:

- **No** interpreter or **virtual machine** (VM) involved
- **Light obfuscation** of the code

Mutation-based Code Obfuscation

In commercial protectors code mutation generally means:

- **No** interpreter or **virtual machine** (VM) involved
- **Light obfuscation** of the code
- Adds and **modifies machine code**, preserves original behavior

Mutation-based Code Obfuscation

In commercial protectors code mutation generally means:

- **No** interpreter or **virtual machine** (VM) involved
- **Light obfuscation** of the code
- Adds and **modifies machine code**, preserves original behavior
- Can modify the **control flow graph**

Initial Motivation

The goal

- Develop a deobfuscator for the mutation engine

Initial Plan of Action

The plan

- Fully understand the features of Themida's mutation engine
- Find potential weaknesses we can leverage to deobfuscate the code

Obtaining Themida

Research done on the demo version of Themida (v3.1.1)

- Available on Oreans's web site[3]
- Contains the same mutation engine as the paid version
- We can use the demo as a black box to infer features and behaviors

[3] https://www.oreans.com/download.php

What Mutation Looks Like

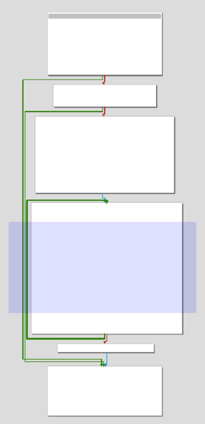


Figure 1: Original CFG (6 basic blocks)

Figure 2: CFG after mutation (74 basic blocks)

What Mutation Looks Like

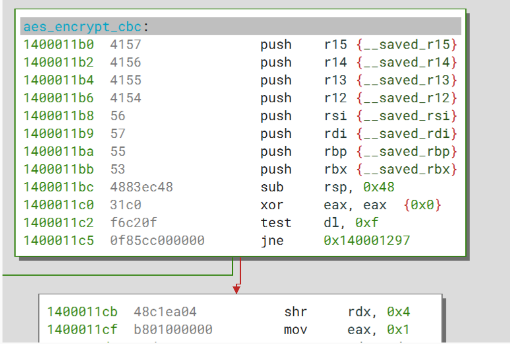


Figure 3: Original code (71 instructions)

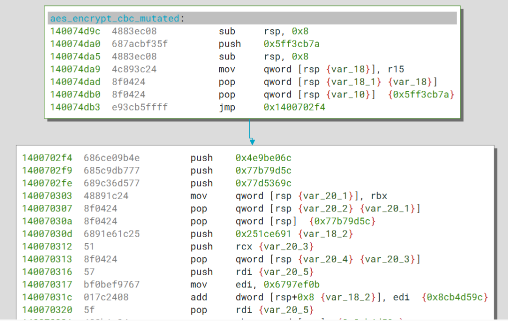
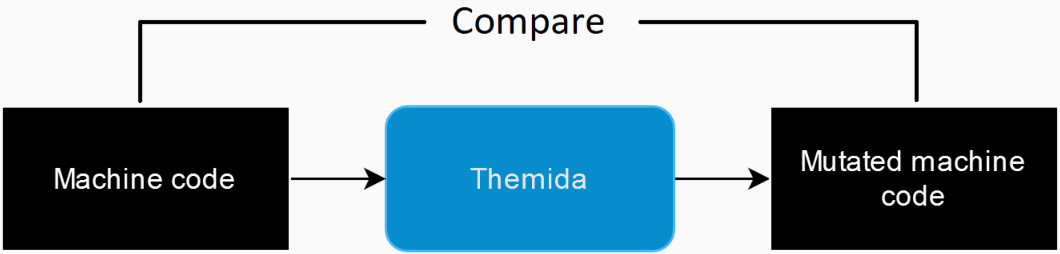


Figure 4: Code after mutation (2160 instructions)

Initial Approach



uops.info

uops.info[a] to the rescue!

- Provides descriptions of all(?) x86 instructions
  - Contained in a single XML "database"
- Provides a script to generate assembly code

[a] https://uops.info/xml.html

```
LOCK ADD byte ptr [RAX], DH
LOCK ADD word ptr [RAX], DX
LOCK ADD dword ptr [RAX], EDX
LOCK ADD qword ptr [RAX], RDX
RET
instruction_coverage_ADD_LOCK endp
instruction_coverage_AND proc EXPORT
AND byte ptr [RAX], 0
AND byte ptr [RAX], 2
AND CL, 0
AND CL, 2
AND BPL, 0
AND BPL, 2
AND CH, 0
AND CH, 2
AND word ptr [RAX], 257
```

Figure 5: Assembly file generated from uops.info's database

Input Generation

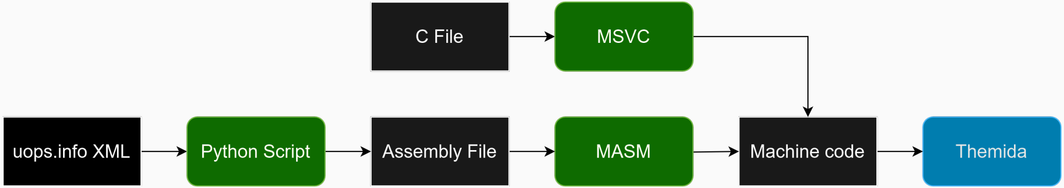


Figure 6: Input generation pipeline

Difficulties

Ended up testing the *SecureEngine*'s instruction handling logic as well:

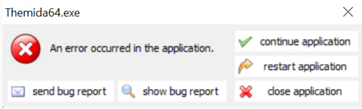
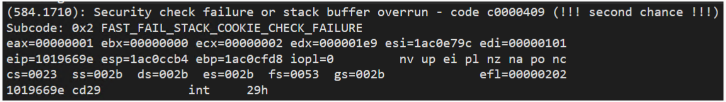


Figure 7: Crash while protecting a function with Themida



```
(584.1710): Security check failure or stack buffer overrun - code c0000409 (!!! second chance !!!)
Subcode: 0x2 FAST_FAIL_STACK_COOKIE_CHECK_FAILURE
eax=00000001 ebx=00000000 ecx=00000002 edx=000001e9 esi=1ac0e79c edi=00000101
eip=1019669e esp=1ac0ccb4 ebp=1ac0cfd8 iopl=0         nv up ei pl nz na po nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
1019669e cd29            int     29h
```

Figure 8: Stack corruption viewed in WinDbg

(Haven't tried to root cause these)

Difficulties

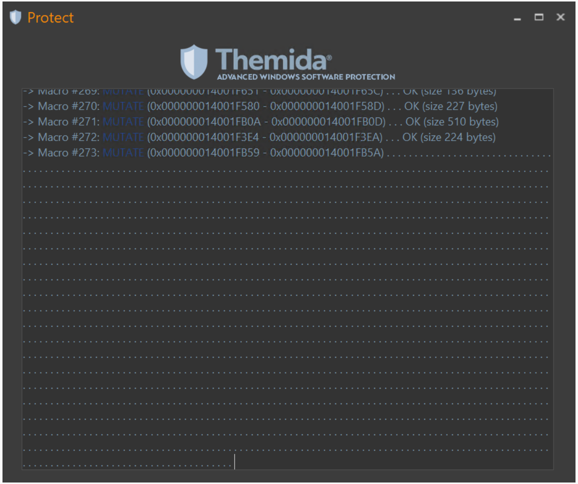


Figure 9: Infinite loop while protecting a function with Themida

Features

SecureEngine's code mutation engine features:

- Opaque function/code entry
- Junk code insertion
- Instruction substitution
    - Constant unfolding
    - Register-to-stack spilling

Opaque Code Entry

- Original code is **redirected to a trampoline**
- Trampoline is used to hinder static analysis
  - Equivalent to obfuscated `push ADDR; ret`
  - Redirects to the actual obfuscated code

Opaque Code Entry

```
aes_encrypt_cbc:
1400011b0   e96d681800        jmp     aes_encrypt_cbc_trampoline
```

Figure 10: Entry of a protected function

Opaque Code Entry

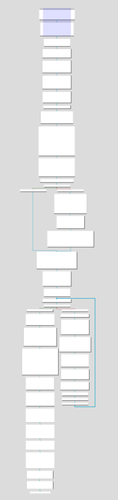


Figure 11: CFG of trampolines generated by Themida to wrap code

Opaque Code Entry

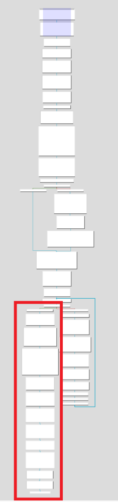


Figure 12: Part of the CFG which computes the obfuscated code's address

Junk Code Insertion

- Junk code insertion is triggered randomly, for **75% of all instructions**
- Junk code can be **inserted before** original instructions **or after or both**
- Junk code **cancels itself** out **within a single basic block**

Junk Code Insertion

Example of `MOV` instruction with junk code inserted around:

```
push eax
add ax, 42
shl eax, 12
mov ebx, ecx ; Original instruction
pop eax
```

Instruction Substitution

The *SecureEngine*'s code mutation engine can substitute the **14** following x86 instruction classes[4]:

AND, **DEC**, **INC**, **JMP**, **MOV**, **MOVZX**, **NEG**, **NOT**, **OR**, **POP**, **PUSH**, **SUB**, **XCHG**, **XOR**

The instruction substitution pass is **always** applied to supported instructions.

[4] In XED, an instruction class is "what is typically thought of as the instruction mnemonic."

Instruction Substitution

Example of `XCHG` instruction substitution:

```
xchg     bl, dh
```

Figure 13: Original instruction

```
xor      bl, dh
xor      dh, bl
xor      bl, dh
```

Figure 14: Mutated instruction

Constant Unfolding

Example of constant unfolding on MOV:

```
mov        rdx, 0x539
```

Figure 15: Original instruction

```
push       r13 {var_8}
push       rbx {var_10}
mov        rbx, 0x5affb935
mov        r13, 0x2f039267
xor        r13, rbx
pop        rbx {var_10}
or         r13, 0x7b59f878
xor        r13, 0x6e7f9195
xor        r13, 0x11826fd6   {0x539}
mov        rdx, r13   {0x539}
pop        r13 {var_8}
```

Figure 16: Mutated instruction

FLAGS Register

To preserve FLAGS register, the engine disables code mutation locally when needed:

Figure 17: "Mutated" intructions when FLAGS are used

Broken Instructions

Interestingly, some instructions can be **randomly** transformed into broken machine code.

Example of a broken `FCMOVNB` instruction:

```
instruction_coverage_FCMOVNB:
14001c299  dbc0            fcmovnb st0, st0
```

Figure 18: Original instruction

```
instruction_coverage_FCMOVNB:
14001c299  1100            adc     dword [rax], eax
14001c29b  684b000248      push    0x4802004b {var_8}
14001c2a0  3cdf            cmp     al, 0xdf
14001c2a2  17              ??
```

Figure 19: “Mutated” instruction

Broken Semantics

But also, semantics can be broken sometimes:

```
xchg      dh, dh
```

Figure 20: Original instruction (`NOP`)

```
xor       dh, dh   {0x0}
xor       dh, dh   {0x0}
xor       dh, dh   {0x0}
```

Figure 21: Mutated instruction (`MOV DH, 0`)

Weaknesses

The obfuscation is annoying enough, but there are some weaknesses:

- **Each basic block** is created from **one original instruction**
- **Each basic block** is **mutated independently**
- The original function's **CFG is preserved**

This means we can **deobfuscate each basic block individually** to recover original instructions.

Simplifying The Code

To simplify the code, a couple of ideas came to mind too, but both involve an IR:

- Code Optimization
- Program Synthesis

Simplifying The Code

To simplify the code, a couple of ideas came to mind too, but both involve an IR:

- Code Optimization
- Program Synthesis
  - Symbolic Execution

The Big Picture

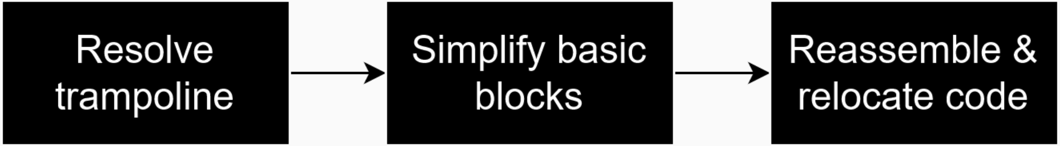


Figure 22: Deobfuscation process, the big picture

The Big Picture

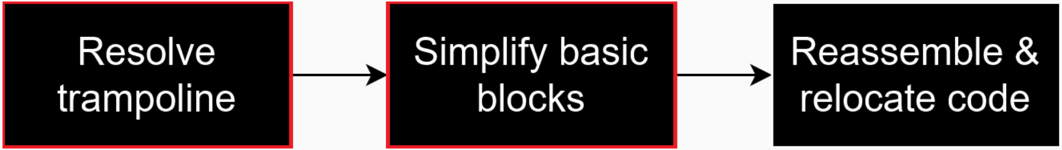


Figure 23: Deobfuscation process, the big picture

Defeating Trampolines

To defeat opaque code entry, we can **symbolically execute trampolines**

- Trampolines contains 2 conditional branches
- Trampoline **logic is always the same**

Defeating Trampolines

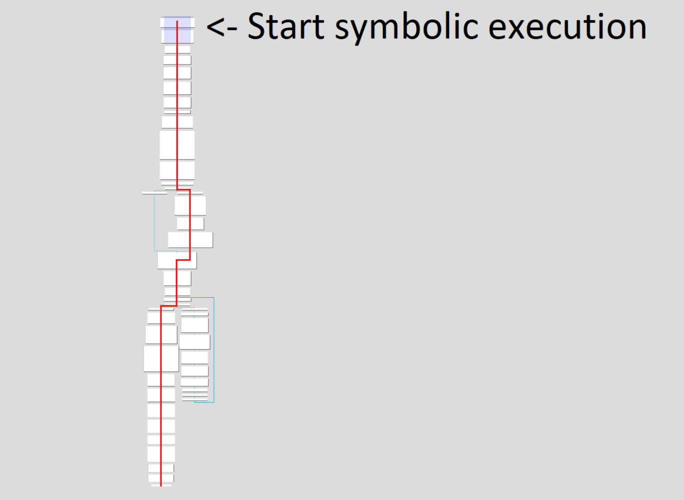


Figure 24: Symbolic Execution Path

Defeating Trampolines

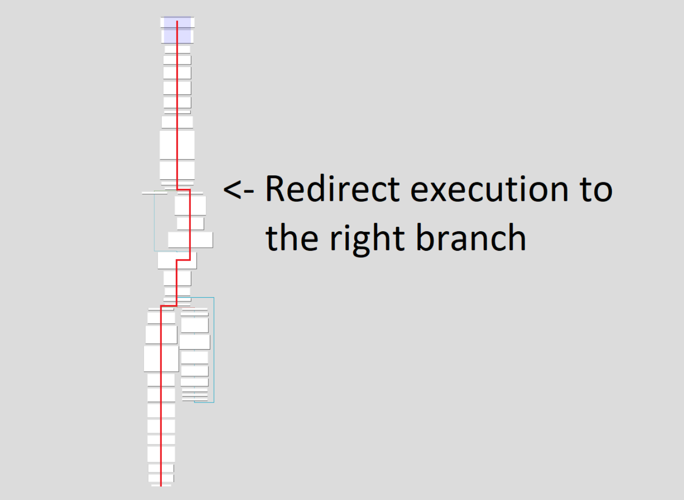


Figure 25: Symbolic Execution Path

Defeating Trampolines

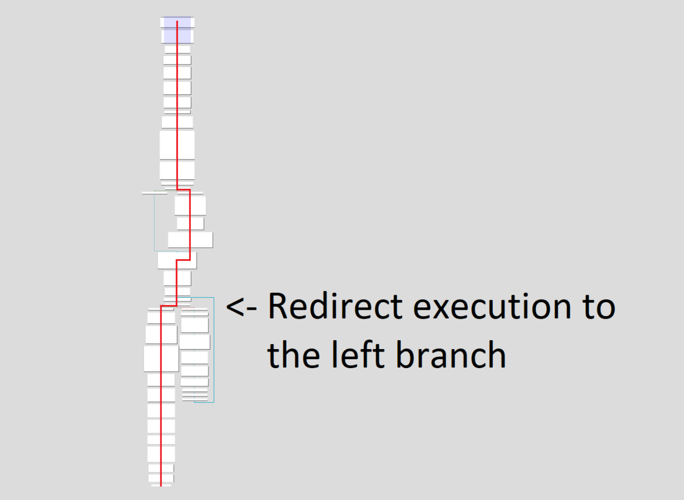


Figure 26: Symbolic Execution Path

Defeating Trampolines

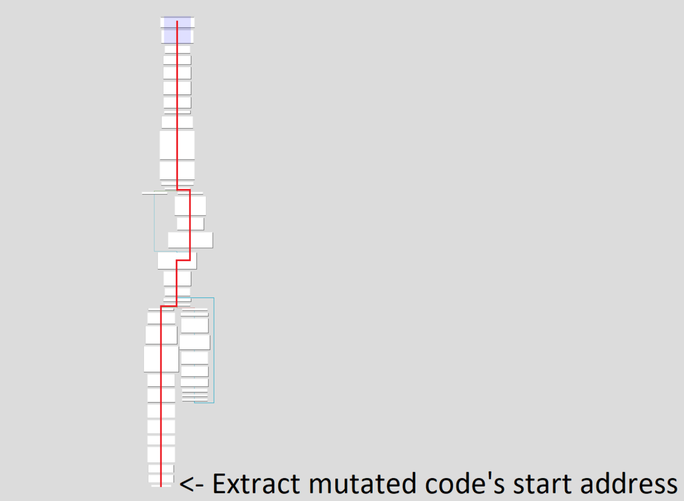


Figure 27: Symbolic Execution Path

Instruction Synthesis

We can differentiate 3 cases for the instruction synthesis process.

Instruction Synthesis (case #1)

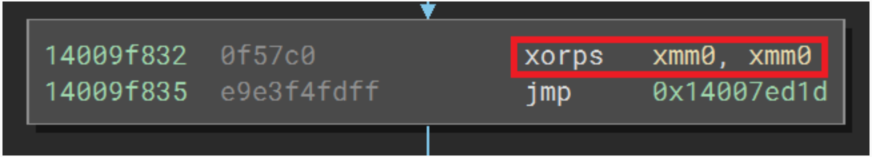


Figure 28: Case #1 (**no junk code**, no substitution)

Instruction Synthesis (case #2)

```
14007ed67  4883c208          add     rdx, 0x8
14007ed6b  4881c208000000    add     rdx, 0x8
14007ed72  48331424          xor     rdx, qword [rsp]
14007ed76  48311424          xor     qword [rsp], rdx
14007ed7a  48331424          xor     rdx, qword [rsp]
14007ed7e  5c                pop     rsp
14007ed7f  0f114720          movups  xmmword [rdi+0x20], xmm0
14007ed83  68c159ff7f        push    0x7fff59c1
14007ed88  4c891424          mov     qword [rsp], r10
14007ed8c  54                push    rsp
14007ed8d  415a              pop     r10
14007ed8f  4983c208          add     r10, 0x8
14007ed93  4981ea08000000    sub     r10, 0x8
14007ed9a  4c871424          xchg    qword [rsp], r10
```

Figure 29: Case #2 (**junk code inserted**, no substitution)

Instruction Synthesis (case #2)

```
{
    ExprId('zf', 1): ExprOp('==', ExprId('RSP', 64), ExprInt(0x0, 64)),
    ExprId('af', 1): ExprSlice(ExprOp('^', ExprId('RSP', 64), ExprOp('+', ExprId('RSP', 64
    ExprId('cf', 1): ExprSlice(ExprOp('^', ExprId('RSP', 64), ExprOp('&', ExprOp('^', Expr
    ExprId('of', 1): ExprSlice(ExprOp('&', ExprOp('^', ExprId('RSP', 64), ExprOp('+', Expr
    ExprId('nf', 1): ExprSlice(ExprId('RSP', 64), 63, 64),
    ExprMem(ExprOp('+', ExprId('RDI', 64), ExprInt(0x20, 64)), 128): ExprId('XMM0', 128)
    ExprId('IRDst', 64): ExprInt(0xA2, 64),
    ExprId('pf', 1): ExprOp('parity', ExprOp('&', ExprId('RSP', 64), ExprInt(0xFF, 64)))
}
```

Figure 30: Basic block's symbolic execution

```
{
    ExprId('IRDst', 64): ExprInt(0x4, 64),
    ExprMem(ExprOp('+', ExprId('RDI', 64), ExprInt(0x20, 64)), 128): ExprId('XMM0', 128)
}
```

Figure 31: MOVUPS instruction's symbolic execution

```
{
    ExprId('zf', 1): ExprOp('==', ExprId('RSP', 64), ExprInt(0x0, 64)),
    ExprId('af', 1): ExprSlice(ExprOp('^', ExprId('RSP', 64), ExprOp(':', ExprId('RSP', 64
    ExprId('cf', 1): ExprSlice(ExprOp('^', ExprId('RSP', 64), ExprOp('&', ExprOp('^', Expr
    ExprId('of', 1): ExprSlice(ExprOp('&', ExprOp('^', ExprId('RSP', 64), ExprOp('+', Expr
    ExprId('nf', 1): ExprSlice(ExprId('RSP', 64), 63, 64),
ExprMem(ExprOp('+', ExprId('RDI', 64), ExprInt(0x20, 64)), 128): ExprId('XMM0', 128)
    ExprId('IRDst', 64): ExprInt(0xA2, 64),
    ExprId('pf', 1): ExprOp('parity', ExprOp('&', ExprId('RSP', 64), ExprInt(0xFF, 64)))
}
```

Figure 32: Basic block's symbolic execution (FLAGS removed)

```
{
    ExprId('IRDst', 64): ExprInt(0x4, 64),
    ExprMem(ExprOp('+', ExprId('RDI', 64), ExprInt(0x20, 64)), 128): ExprId('XMM0', 128)
}
```

Figure 33: MOVUPS instruction's symbolic execution

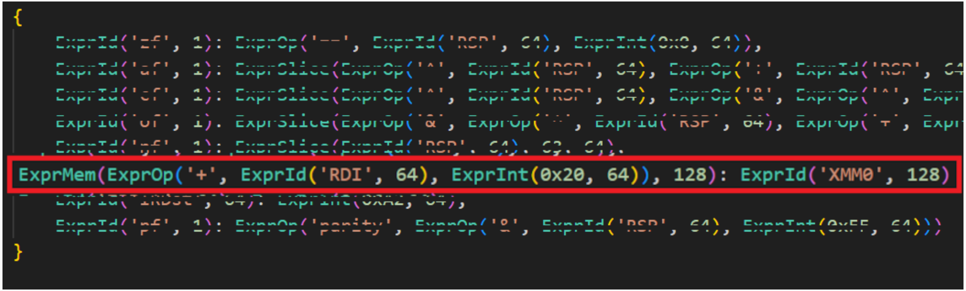


Figure 34: Basic block's symbolic execution (FLAGS removed)

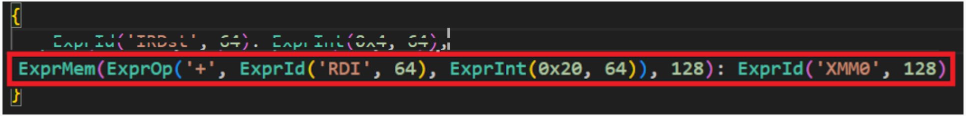


Figure 35: MOVUPS instruction's symbolic execution

Instruction Synthesis (case #3)

For instructions which the mutation **engine can substitute**:

- We only have to manually synthesize **14 instruction classes**
- Development effort is thus **symmetric between attack and defense**
- We can use **pattern matching**

Instruction Synthesis (case #3)

```
{
    ExprId('zf', 1): ExprOp('==', ExprId('RSP', 64), ExprInt(0x0, 64)),
    ExprId('af', 1): ExprSlice(ExprOp('^', ExprId('RSP', 64), ExprOp('+'
    ExprId('pf', 1): ExprOp('parity', ExprOp('&', ExprId('RSP', 64), Expr
    ExprId('of', 1): ExprSlice(ExprOp('&', ExprOp('^', ExprId('RSP', 64)
    ExprId('R13', 64): ExprId('R9', 64),
    ExprId('nf', 1): ExprSlice(ExprId('RSP', 64), 63, 64),
    ExprId('cf', 1): ExprSlice(ExprOp('^', ExprId('RSP', 64), ExprOp('&'
    ExprId('IRDst', 64): ExprInt(0x120, 64)
}
```

Figure 36: Basic block's symbolic execution

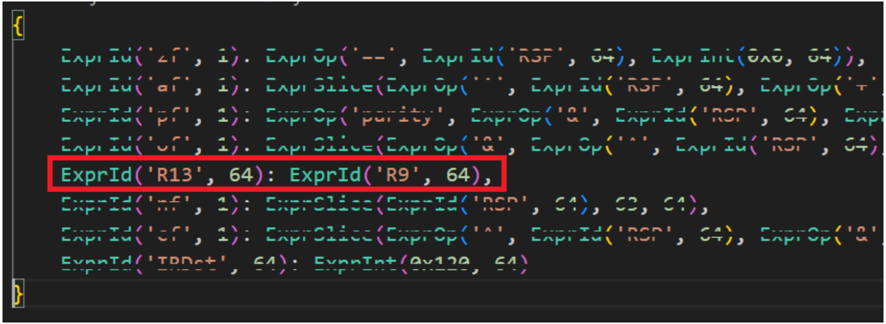


Figure 37: Basic block's symbolic execution (FLAGS removed)

Instruction Synthesis (case #3)

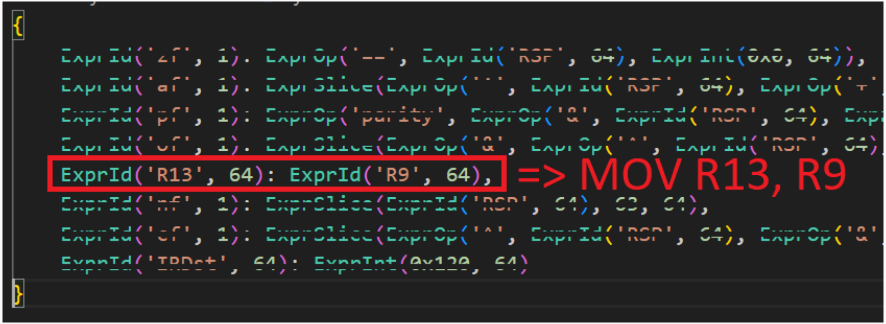


Figure 38: Instruction "synthesized" via pattern matching

Result

```
$ ./sha256_test_protected.exe
SHA-256 tests: SUCCEEDED
$ themida-unmutate ./sha256_test_protected.exe -a 0x1400011d0 0x140001000 0x140001200 0x140001270
-o sha256_test_simplified.exe
INFO - Resolving mutated's functions' addresses...
INFO - Function at 0x1400011d0 jumps to 0x14031f24a
INFO - Function at 0x140001000 jumps to 0x140028532
INFO - Function at 0x140001200 jumps to 0x140211875
INFO - Function at 0x140001270 jumps to 0x1400760b7
INFO - Deobfuscating mutated functions...
INFO - Simplifying function at 0x14031f24a...
INFO - Simplifying function at 0x140028532...
INFO - Simplifying function at 0x140211875...
INFO - Simplifying function at 0x1400760b7...
INFO - Rebuilding binary file...
INFO - Done! You can find your deobfuscated binary at 'sha256_test_simplified.exe'
$ ./sha256_test_simplified.exe
SHA-256 tests: SUCCEEDED
$ |
```

Figure 39: Simplified binaries can be run

Result

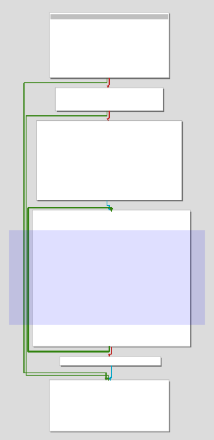


Figure 40: Original (6 BBs)

Figure 41: Obfuscated (74 BBs)

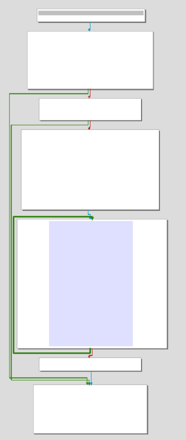


Figure 42: Deobfuscated (7 BBs)

Result

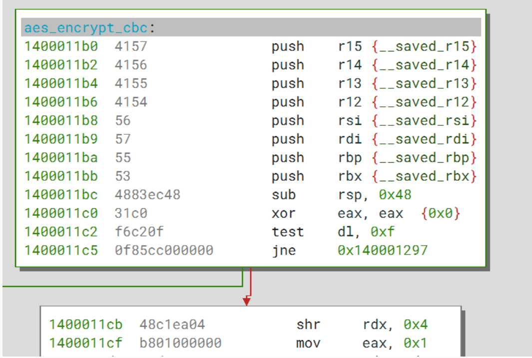


Figure 43: Original (71 instructions)

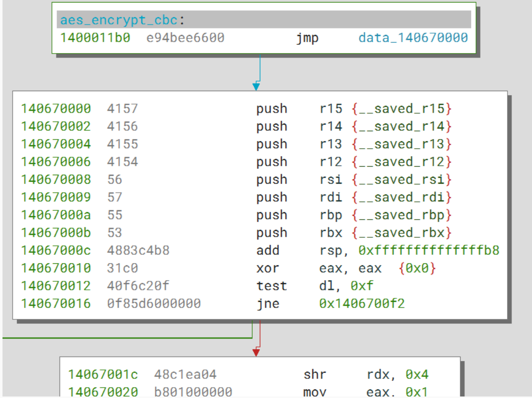


Figure 44: Deobfuscated (74 instructions)

Recap

To recap:

- A few **weaknesses** facilitated the work

Recap

To recap:

- A few **weaknesses** facilitated the work
- Static **symbolic execution** was very **effective**

Recap

To recap:

- A few **weaknesses** facilitated the work
- Static **symbolic execution** was very **effective**
- The attack **scales and works seemlessly on complex functions**
    - Time complexity is roughly linear to the number of basic blocks
    - It can be parallelized

Recap

To recap:

- A few **weaknesses** facilitated the work
- Static **symbolic execution** was very **effective**
- The attack **scales and works seemlessly on complex functions**
  - Time complexity is roughly linear to the number of basic blocks
  - It can be parallelized
- We're able to recover very **close-to-original machine code**

Recap

To recap:

- A few **weaknesses** facilitated the work
- Static **symbolic execution** was very **effective**
- The attack **scales and works seemlessly on complex functions**
  - Time complexity is roughly linear to the number of basic blocks
  - It can be parallelized
- We're able to recover very **close-to-original machine code**
- **Binaries can be patched** to run on the deobfuscated code

Recap

To recap:

- A few **weaknesses** facilitated the work
- Static **symbolic execution** was very **effective**
- The attack **scales and works seemlessly on complex functions**
  - Time complexity is roughly linear to the number of basic blocks
  - It can be parallelized
- We're able to recover very **close-to-original machine code**
- **Binaries can be patched** to run on the deobfuscated code

A blog series will be published soon with more details, stay tuned!

Questions?

Code is available here (GPL-3.0): https://github.com/ergrelet/themida-unmutate

Figure 45: QR Code for the link above