emproof

Unveiling Secrets in Binaries using Code Detection Strategies

Tim Blazytko

@mr_phrazer
synthesis.to
tim@blazytko.to

About Tim

- Chief Scientist, co-founder of emproof
- designs software protections for embedded devices
- trainer for (de)obfuscation and reverse engineering techniques

- Navigating in Large Binaries
- Common Strategies
- Code Detection Heuristics

Large Binary

Analysis Challenges

- locating **complex state machines** and protocol logic

Analysis Challenges

- locating **complex state machines** and protocol logic

vulnerability discovery

Analysis Challenges

- locating **complex state machines** and protocol logic
- detecting **cryptographic implementations**

Analysis Challenges

- locating **complex state machines** and protocol logic
- detecting **cryptographic implementations**

$\boxed{\text{malware \& vulnerability analysis}}$

Analysis Challenges

- locating **complex state machines** and protocol logic
- detecting **cryptographic implementations**
- discovering **C&C server communication** and **string decryption routines**

Analysis Challenges

- locating **complex state machines** and protocol logic
- detecting **cryptographic implementations**
- discovering **C&C serve** … **tion routines**

malware analysis

Analysis Challenges

- locating **complex state machines** and protocol logic
- detecting **cryptographic implementations**
- discovering **C&C server communication** and **string decryption routines**
- pinpointing **obfuscated code** in commercial applications

Analysis Challenges

- locating **complex state machines** and protocol logic
- detecting **cryptographic implementations**
- discovering **C&C server** [illegible]**ption routines**
- pinpointing **obfuscated code** in commercial applications

software piracy

Analysis Challenges

- locating **complex state machines** and protocol logic
- detecting **cryptographic implementations**
- discovering **C&C server communication** and **string decryption routines**
- pinpointing **obfuscated code** in commercial applications
- identifying **API functions** in **statically-linked executables**

Analysis Challenges

- locating **complex state machines** and protocol logic
- detecting **cryptographic implementations**
- discoverin
- pinpointing **obfuscated code** in commercial applications
- identifying **API functions** in **statically-linked executables**

embedded firmware analysis

Analysis Challenges

- locating **complex state machines** and protocol logic
- detecting **cryptographic implementations**
- pinpointing **obfuscated code** in commercial applications
- identifying **API functions** in **statically-linked executables**

Goal: Identifying interesting code locations

Where to start?

Common Approaches

- function symbols

- function symbols

```
validate_serial()
```

Common Approaches

- function symbols
- meaningful strings

Common Approaches

- function symbols
- meaningful strings

“https://evildomain.com”

Common Approaches

- function symbols
- meaningful strings
- interesting API functions

Common Approaches

- function symbols
- meaningful strings
- interesting API functions

`GetAsyncKeyState`

Common Approaches

- function symbols
- meaningful strings
- interesting API functions

Not always applicable

→ Code Detection Heuristics

Code Detection Heuristics

Identification of **interesting** code constructs

Code Detection Heuristics

Identification of **interesting** code constructs

- **guide** manual analysis

Code Detection Heuristics

Identification of **interesting** code constructs

- **guide** manual

False positives will occur

Code Detection Heuristics

Identification of **interesting** code constructs

- **guide** manual analysis
- architecture-agnostic

Identification of **interesting** code constructs

All architectures supported by the disassembler

- architecture-agnostic

Code Detection Heuristics

Identification of **interesting** code constructs

- **guide** manual analysis
- architecture-agnostic
- efficient to compute

Code Detection Heuristics

Identification of **interesting** code constructs

- guide
- architecture-agnostic
- efficient to compute

Applicable to ~100,000 functions

How?

Code Complexity and Statistical Analysis

Interesting code is

- (artificially) complex
- frequently executed
- uncommon

Code Complexity and Statistical Analysis

Interesting code is

- (artificially) complex
- frequently executed
- uncommon

- basic block/function size

Interesting code is

- (artificially) complex
- frequently executed
- uncommon

complex code

- basic block/function size

Code Complexity and Statistical Analysis

Interesting code is

- (artificially) complex
- frequently executed
- uncommon

- basic block/function size
- control-flow graph characteristics

Interesting code is

- (artificially) complex
- frequently executed
- uncommon

underlying code constructs

- basic block/function size
- control-flow graph characteristics

Code Complexity and Statistical Analysis

Interesting code is

- (artificially) complex
- frequently executed
- uncommon

- basic block/function size
- control-flow graph characteristics
- frequency analysis

Code Complexity and Statistical Analysis

Interesting code is

- (artificially) complex
- frequently executed
- uncommon

(un)common code patterns

- basic block/function size
- control-flow graph characteristics
- frequency analysis

Code Complexity and Statistical Analysis

Interesting code is

- (artificially) complex
- frequently executed
- uncommon

- basic block/function size
- control-flow graph characteristics
- frequency analysis
- usage of intermediate representations

Interesting code is

- (artificially) complex
- frequently executed
- uncommon

architecture-agnostic instruction patterns

- basic block/function size
- control-flow graph characteristics
- frequency analysis
- usage of intermediate representations

Detection Heuristics

Heuristics

1. large basic blocks
2. complex functions
3. frequently called functions
4. state machines
5. uncommon instruction sequences

Detection Heuristics

Heuristics

1. large basic blocks
2. complex functions
3. frequently called functions
4. state machines
5. uncommon instruction sequences

- most heuristics **relative** to all functions in the binary

Detection Heuristics

Heuristics

1. large basic blocks
2. complex functions
3. frequently called functions
4. state machines
5. un

Clear separation between functions

- most heuristics **relative** to all functions in the binary

Detection Heuristics

Heuristics

1. large basic blocks
2. complex functions
3. frequently called functions
4. state machines
5. uncommon instruction sequences

- most heuristics **relative** to all functions in the binary
- each heuristic detects **different** patterns

Detection Heuristics

Heuristics
1. large basic blocks
2. complex functions
3. frequently called functions
4. state machines
5. uncommon i

Know what to use & when

- most heuristics **relative** to all functions in the binary
- each heuristic detects **different** patterns

Large Basic Blocks

Large Basic Blocks

Identification of functions with large basic blocks

Large Basic Blocks

Identification of functions with large basic blocks

- ~5-7 instructions per basic block (on average)

Large Basic Blocks

Identification of functions with large basic blocks

- ~5-7 instructions per basic block (on average)
- larger basic blocks indicate **complex straight-line code**

Large Basic Blocks

Identification of functions with large basic blocks

- ~5-7 instructions per basic block (on average)
- larger basic blocks indicate **complex straight-line code**
- compute per function:

$$\frac{\#\text{instructions}}{\#\text{basic blocks}}$$

Complex Straight-Line Code

- unrolled loops
- cryptographic implementations
- initialization routines
- arithmetic obfuscation

Example: Anti-Cheat I

average #instructions / block per function (in descending order):

1,456
198
68
63
59
55
52
51
49
46

Example: Anti-Cheat I

average #instructions/block per function (in descending order):

1,456
198
68
63
59
55
52
51
49
46

Example: Anti-Cheat I

average #instructions/block per function (in descending order):

1,456
198
68
63

arithmetic and virtualization-based obfuscation

55
52
51
49
46

Example: `ntoskrnl.exe` (Windows Kernel)

functions and their corresponding avergage #instructions/block (in descending order):

`SepInitSystemDacls`	491
`SymCryptSha256AppendBlocks_ul1`	236
`HalpRestoreHvEnlightenment`	147
`MiInitializeDummyPages`	133
`HalpBlkInitializeProcessorState`	103

Example: `ntoskrnl.exe` (Windows Kernel)

functions and their corresponding avergage #instructions/block (in descending order):

`SepInitSystemDacls`	491
`SymCryptSha256AppendBlocks_ul1`	236
`HalpRestoreHvEnlightenment`	147
`MiInitializeDummyPages`	133
`HalpBlkInitializeProcessorState`	103

Example: `ntoskrnl.exe` (Windows Kernel)

functions and their corresponding avergage #instructions/block (in descending order):

`SepInitSystemDacls`	491
`S`	6
`H`	7
`MiInitializeDummyPages`	133
`HalpBlkInitializeProcessorState`	103

initialization routines

Example: `ntoskrnl.exe` (Windows Kernel)

functions and their corresponding avergage #instructions/block (in descending order):

`SepInitSystemDacls`	491
`SymCryptSha256AppendBlocks_ul1`	236
`HalpRestoreHvEnlightenment`	147
`MiInitializeDummyPages`	133
`HalpBlkInitializeProcessorState`	103

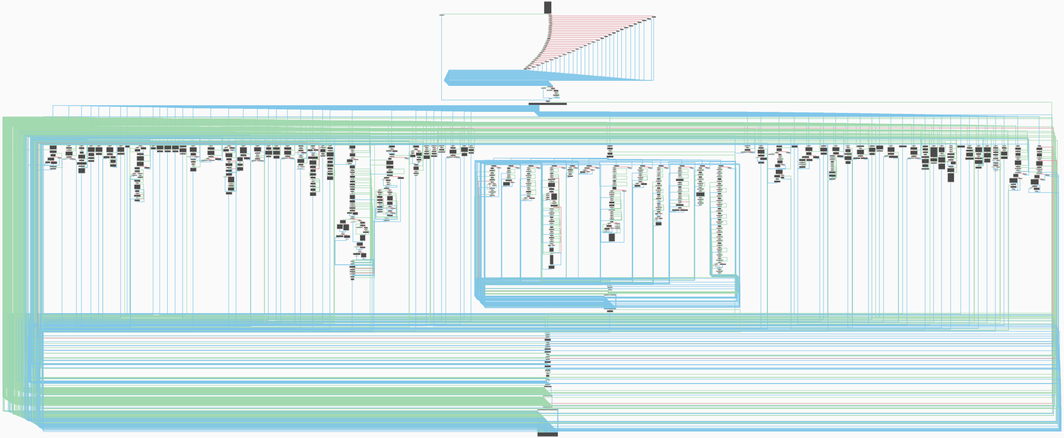
Example: `ntoskrnl.exe` (Windows Kernel)

functions and their corresponding avergage #instructions/block (in descending order):

`SepInitSystemDacls`	491
`MiInitializeDummyPages`	133
`HalpBlkInitializeProcessorState`	103

cryptographic implementations

Complex Functions



Complex Functions

Identification of functions with large control-flow graphs

Complex Functions

Identification of functions with large control-flow graphs

- large functions indicate a **complex code logic**
  - file parsing
  - dispatching routines and network protocols
  - obfuscation

Complex Functions

Identification of functions with large control-flow graphs

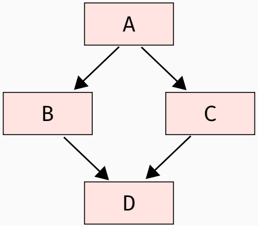
- large functions indicate a **complex code logic**
    - file parsing
    - dispatching routines and network protocols
    - obfuscation
- efficient metric: **cyclomatic complexity**

Cyclomatic Complexity

$$\#\text{edges} - \#\text{basic blocks} + 2$$

Cyclomatic Complexity

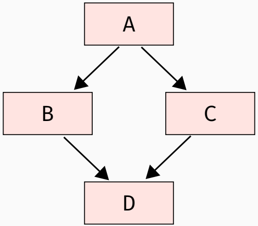
$$\#\text{edges} - \#\text{basic blocks} + 2$$



- 4 basic blocks
- 4 edges

Cyclomatic Complexity

$$\#\text{edges} - \#\text{basic blocks} + 2$$



- 4 basic blocks
- 4 edges

cyclomatic complexity: 2

Example: `ntoskrnl.exe` (Windows Kernel)

cyclomatic complexity per function (in descending order):

2,964
2,371
1,506
718
642
435
414
318
281
274

Example: `ntoskrnl.exe` (Windows Kernel)

cyclomatic complexity per function (in descending order):

2,964
2,371
1,506
718
642
435
414
318
281
274

Example: `ntoskrnl.exe` (Windows Kernel)

cyclomatic complexity per function (in descending order):

2,964
2,371
1,506

related to PatchGuard (anti-tamper protection)

435
414
318
281
274

Frequently Called Functions

```
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef  call 0xdeadbeef
```

Frequently Called Functions

Identification of functions which are **frequently called** from **different functions**

Frequently Called Functions

Identification of functions which are **frequently called** from **different functions**

What kind of functions are called frequently?

Frequently Called Functions

Identification of functions which are **frequently called** from **different functions**

- allows the identification of **API functions** in **statically-linked executables**

Frequently Called Functions

Identification of functions which are **frequently called** from **different functions**

- allows the identification of **API functions** in **statically-linked executables**
- can sometimes also detect **string decryption & hash functions** in malware

Frequently Called API Functions

- memory management
- data movement
- string operations
- file I/O operations

Example: XOR DDoS (Malware)

Most called functions (from **unique** callers) in the **statically-linked** malware:

`free`	293
`memcpy`	191
`strlen`	184
`memset`	174
`__libc_malloc`	151
`__lll_unlock_wake_private`	148
`__lll_lock_wait_private`	122
`ptmalloc_init`	114
`__strtol_internal`	99
`strcmp`	93

Example: XOR DDoS (Malware)

Most called functions (from **unique** callers) in the **statically-linked** malware:

`free`	293
`memcpy`	191
`strlen`	184
`memset`	174
`__libc_malloc`	151
`__lll_unlock_wake_private`	148
`__lll_lock_wait_private`	122
`ptmalloc_init`	114
`__strtol_internal`	99
`strcmp`	93

Example: XOR DDoS (Malware)

Most called functions (from **unique** callers) in the **statically-linked** malware:

free	293
memcpy	191
strlen	184
__lll_unlock_wake_private	148
__lll_lock_wait_private	122
ptmalloc_init	114
__strtol_internal	99
strcmp	93

frequently called API functions

Example: PlugX (Malware)

Most called functions (from **unique** callers) and their number of calls:

`crc32`	1253
`LoadLibraryA`	1253
`__seterrormode`	320

Example: PlugX (Malware)

Most called functions (from **unique** callers) and their number of calls:

`crc32`	1253
`LoadLibraryA`	1253
`__seterrormode`	320

Most called functions (from **unique** callers) and their number of calls:

hash-based import hiding

`__seterrormode`	320

Example: PlugX (Malware)

Most called functions (from **unique** callers) and their number of calls:

`crc32`	1253
`LoadLibraryA`	1253
`__seterrormode`	320

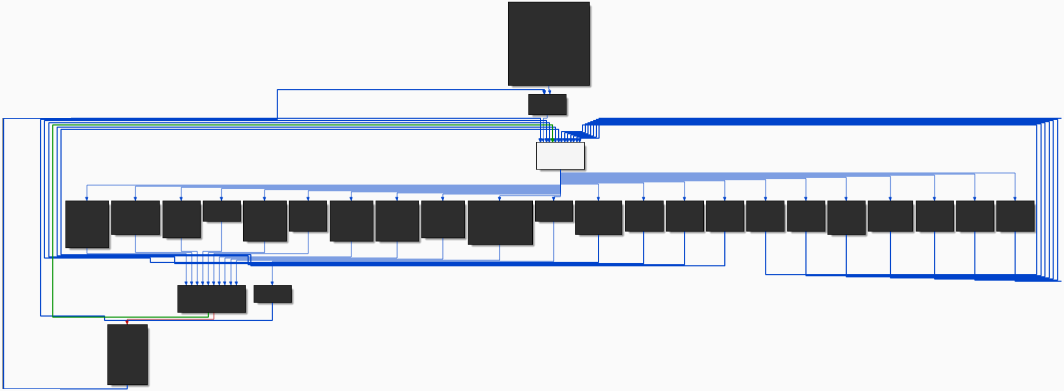
Most called functions (from **unique** callers) and their number of calls:

potential clustering of functions

`__seterrormode` 320

Identification of State Machines

State Machine Heuristic



Identification of functions with **loop-based dispatching routines**

State Machines

Identification of functions with **loop-based dispatching routines**

```
while(true) {
   switch(state) {
      case state_0: ...
      case state_1: ...
      ...
      case state_n: ...
   }
}
```

State Machines

Identification of functions with **loop-based dispatching routines**

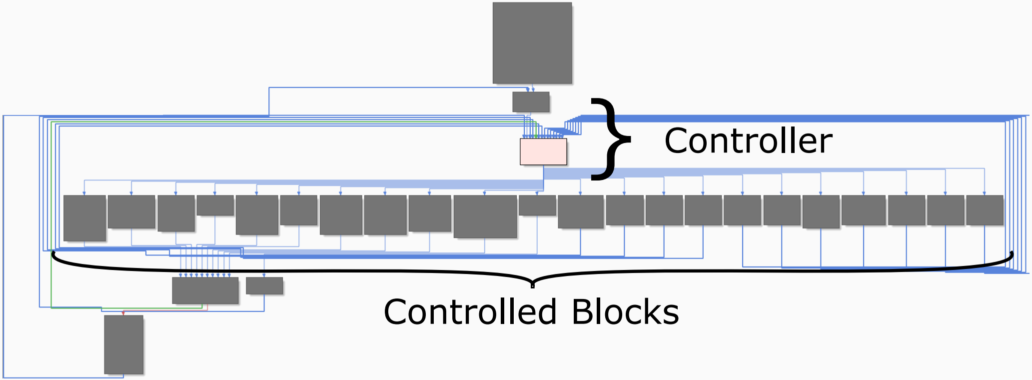
```
while(true) {
   switch(state) {
      case state_0: ...
      case state_1: ...
      ...
      case state_n: ...
   }
}
```

- state machines often implement a **complex program logic**

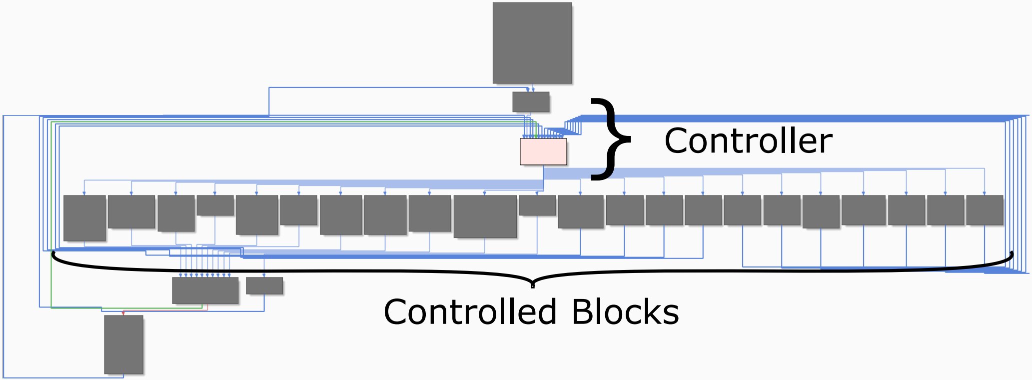
Complex State Machines

- file format parsing
- input validation & sanitization
- network protocol dispatching
- C&C server communication & command dispatching
- data encoding/decoding

State Machine Heuristic



State Machine Heuristic



$$\frac{\#\text{controlled blocks}}{\#\text{blocks in the function}}$$

Examples

PlugX (Malware)
• C&C communication & command dispatching

Examples

PlugX (Malware)

- C&C communication & command dispatching

`ls`

- recursive directory traversal

Examples

PlugX (Malware)

- C&C communication & command dispatching

`ls`

- recursive directory traversal

`gcc`

- file parsing and tokenizing

Uncommon Instruction Sequences

```
mov     eax, dword [rbp]
mov     ecx, dword [rbp + 4]
cmp     r11w, r13w
sub     rbp, 4
not     eax
clc
cmc
cmp     rdx, 0x28b105fa
not     ecx
cmp     r12b, r9b
cmc
and     eax, ecx
jmp     0xc239
mov     word [rbp + 8], eax
pushfq
movzx   eax, r10w
and     ax, di
pop     qword [rbp]
sub     rsi, 4
shld    rax, rdx, 0x1b
xor     ah, 0x4d
mov     eax, dword [rsi]
cmp     ecx, r11d
test    r10, 0x179708d5
xor     eax, ebx
```

```
jmp     0xffffffffff63380
dec     eax
stc
ror     eax, 1
jmp     0xfffffffffff2a70
dec     eax
clc
bswap   eax
test    bp, 0x5124
neg     eax
test    dil, 0xe9
cmp     bx, r14w
cmc
push    rbx
sub     bx, 0x49f8
xor     dword [rsp], eax
and     bh, 0xaf
pop     rbx
movsxd  rax, eax
test    r13b, 0x94
add     rdi, rax
jmp     0xffffffffffc67c7
lea     rax, [rsp + 0x140]
cmp     rbp, rax
ja      0x6557b
jmp     rdi
```

Observation

Statistical Analysis of Assembly Code

Common Instruction Sequences

```
push rbp
mov rbp, rsp
push rbx
push rax
mov rbx, qword [rdi+0x30]
mov edi, dword [rdi+0x38]
call _strerror
lea rdi, [0x7bc6]
mov rsi, rbx
mov rdx, rax
xor eax, eax
call _warnx
mov byte [0x8678], 0x1
add rsp, 0x8
pop rbx
pop rbp
retn
```

Common Instruction Sequences

```
push rbp
mov rbp, rsp
push rbx
push rax
mov rbx, qword [rdi+0x30]
mov edi, dword [rdi+0x38]
call _strerror
lea rdi, [0x7bc6]
mov rsi, rbx
mov rdx, rax
xor eax, eax
call _warnx
mov byte [0x8678], 0x1
add rsp, 0x8
pop rbx
pop rbp
retn
```

```
push rbp
mov rbp, rsp
push rbx
push rax
mov rbx, qword [rdi+0x30]
mov edi, dword [rdi+0x38]
```

prologues and epilogues

```
mov rsi, rbx
mov rdx, rax
xor eax, eax
call _warnx
mov byte [0x8678], 0x1
add rsp, 0x8
pop rbx
pop rbp
retn
```

Common Instruction Sequences

```
push rbp
mov rbp, rsp
push rbx
push rax
mov rbx, qword [rdi+0x30]
mov edi, dword [rdi+0x38]
call _strerror
lea rdi, [0x7bc6]
mov rsi, rbx
mov rdx, rax
xor eax, eax
call _warnx
mov byte [0x8678], 0x1
add rsp, 0x8
pop rbx
pop rbp
retn
```

```
push rbp
mov rbp, rsp
push rbx
push rax
mov rbx, qword [rdi+0x30]
mov edi, dword [rdi+0x38]
call _strerror
le
mov rsi, rbx
mov rdx, rax
xor eax, eax
call _warnx
mov byte [0x8678], 0x1
add rsp, 0x8
pop rbx
pop rbp
retn
```

function calls

Common Instruction Sequences

```
push rbp
mov rbp, rsp
push rbx
push rax
mov rbx, qword [rdi+0x30]
mov edi, dword [rdi+0x38]
call _strerror
lea rdi, [0x7bc6]
mov rsi, rbx
mov rdx, rax
xor eax, eax
call _warnx
mov byte [0x8678], 0x1
add rsp, 0x8
pop rbx
pop rbp
retn
```

```
push rbp
mov rbp, rsp
push rbx
push rax
mov rbx, qword [rdi+0x30]
mov edi, dword [rdi+0x38]
call _strerror
```

data movement

```
mov rsi, rbx
mov rdx, rax
xor eax, eax
call _warnx
mov byte [0x8678], 0x1
add rsp, 0x8
pop rbx
pop rbp
retn
```

Uncommon Instruction Sequences

Identification of functions with a large number of **unusual** instruction sequences

- intensive use of **floating-point** instructions
- cryptographic implementations
- obfuscated code

ground truth of the **1,000 most common** instruction sequences:

```
mov  mov   mov
mov  call  mov
mov  mov   call
        ...
sar  mov   mov
```

Statistical Analysis

ground truth of the **1,000 most common** instruction sequences:

```
mov   mov    mov
mov   call   mov
mov   mov    call
        ...
sar   mov    mov
```

How many instruction sequences are **not** in the ground truth?

ground truth of the **1,000 most common** instruction sequences:

```
mov   mov    mov
mov   call   mov
mov   mov    call
```

architecture-agnostic implementation based on intermediate representations

```
sar   mov    mov
```

How many instruction sequences are **not** in the ground truth?

Example: `ci.dll` (Windows Kernel Module)

Functions with the most uncommon instruction sequences (in descending order):

```
MinCryptIsFileRevoked
__security_check_cookie
SymCryptFdefMaskedCopyAsm
SymCryptSha256AppendBlocks_shani
SymCryptFdefRawMulMulx1024
SymCryptParallelSha256AppendBlocks_ymm
SymCryptParallelSha256AppendBlocks_xmm
SymCryptModElementIsZero
SymCryptFdefMontgomeryReduceMulx1024
CipIsSigningLevelRuntimeCustomizable
SymCryptFdefMontgomeryReduceMulx
Gvd5e6c0
```

Example: `ci.dll` (Windows Kernel Module)

Functions with the most uncommon instruction sequences (in descending order):

```
MinCryptIsFileRevoked
__security_check_cookie
SymCryptFdefMaskedCopyAsm
SymCryptSha256AppendBlocks_shani
SymCryptFdefRawMulMulx1024
SymCryptParallelSha256AppendBlocks_ymm
SymCryptParallelSha256AppendBlocks_xmm
SymCryptModElementIsZero
SymCryptFdefMontgomeryReduceMulx1024
CipIsSigningLevelRuntimeCustomizable
SymCryptFdefMontgomeryReduceMulx
Gvd5e6c0
```

Example: `ci.dll` (Windows Kernel Module)

Functions with the most uncommon instruction sequences (in descending order):

```
MinCryptIsFileRevoked
__security_check_cookie
SymCryptFdefMaskedCopyAsm
SymCryptSha256AppendBlocks_shani
```

cryptographic implementations

```
SymCryptParallelSha256AppendBlocks_xmm
SymCryptModElementIsZero
SymCryptFdefMontgomeryReduceMulx1024
CipIsSigningLevelRuntimeCustomizable
SymCryptFdefMontgomeryReduceMulx
Gvd5e6c0
```

Example: `ci.dll` (Windows Kernel Module)

Functions with the most uncommon instruction sequences (in descending order):

```
MinCryptIsFileRevoked
__security_check_cookie
SymCryptFdefMaskedCopyAsm
SymCryptSha256AppendBlocks_shani
SymCryptFdefRawMulMulx1024
SymCryptParallelSha256AppendBlocks_ymm
SymCryptParallelSha256AppendBlocks_xmm
SymCryptModElementIsZero
SymCryptFdefMontgomeryReduceMulx1024
CipIsSigningLevelRuntimeCustomizable
SymCryptFdefMontgomeryReduceMulx
Gvd5e6c0
```

Example: `ci.dll` (Windows Kernel Module)

Functions with the most uncommon instruction sequences (in descending order):

```
MinCryptIsFileRevoked
__security_check_cookie
SymCryptFdefMaskedCopyAsm
SymCryptSha256AppendBlocks_shani
SymCryptFdefRawMulMulx1024
```

virtualization-based obfuscation

```
SymCryptParallelSha256AppendBlocks_xmm
SymCryptModElementIsZero
SymCryptFdefMontgomeryReduceMulx1024
CipIsSigningLevelRuntimeCustomizable
SymCryptFdefMontgomeryReduceMulx
Gvd5e6c0
```

Conclusion

Takeaways

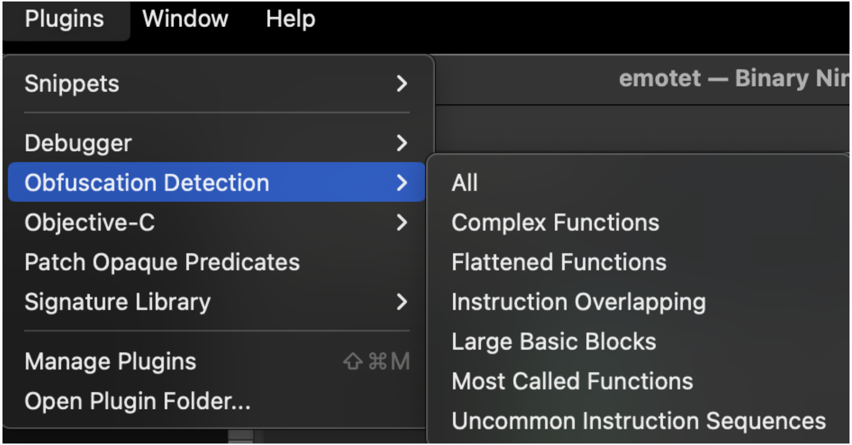
1. efficient and architecture-agnostic heuristics
2. detects a wide range of **interesting** code constructs
3. false positives will occur

Takeaways

1. efficient and architecture-agnostic heuristics
2. detects a wide range of **interesting** code constructs
3. false positives will occur

Useful methods to guide manual analysis in unknown binaries.

Binary Ninja Plugin



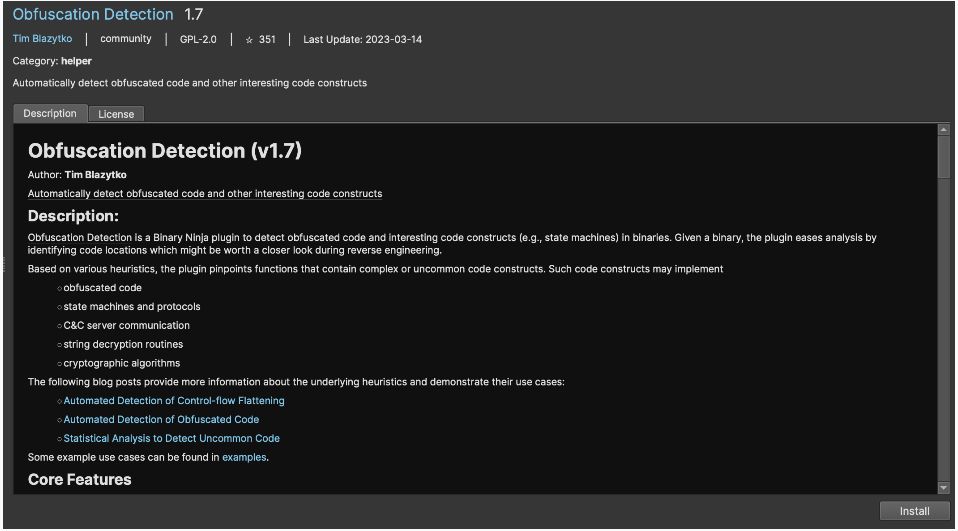
https://github.com/mrphrazer/obfuscation_detection

Binary Ninja Plugin

```
Log    Python Console
Search log
[Default] ==============================================================================
[Default] Control Flow Flattening
[Default] Function 0x4063f0 (sub_4063f0) has a flattening score of 0.9929577464788732.
[Default] Function 0x4012a0 (sub_4012a0) has a flattening score of 0.9855072463768116.
[Default] Function 0x402b60 (sub_402b60) has a flattening score of 0.9855072463768116.
[Default] Function 0x409e20 (sub_409e20) has a flattening score of 0.9846153846153847.
[Default] Function 0x40a4b0 (sub_40a4b0) has a flattening score of 0.9821428571428571.
[Default] Function 0x404f50 (sub_404f50) has a flattening score of 0.9818181818181818.
[Default] Function 0x402210 (sub_402210) has a flattening score of 0.9807692307692307.
[Default] Function 0x4025a0 (sub_4025a0) has a flattening score of 0.9787234042553191.
[Default] Function 0x40a9d0 (sub_40a9d0) has a flattening score of 0.9772727272727273.
[Default] Function 0x409530 (sub_409530) has a flattening score of 0.9761904761904762.
[Default] Function 0x407060 (sub_407060) has a flattening score of 0.975609756097561.
[Default] Function 0x401fa0 (sub_401fa0) has a flattening score of 0.975609756097561.
[Default] Function 0x406080 (sub_406080) has a flattening score of 0.975.
[Default] Function 0x4038b0 (sub_4038b0) has a flattening score of 0.975.
[Default] Function 0x401940 (sub_401940) has a flattening score of 0.9736842105263158.
[Default] Function 0x408660 (sub_408660) has a flattening score of 0.972972972972973.
[Default] Function 0x408f30 (sub_408f30) has a flattening score of 0.972972972972973.
[Default] Function 0x409860 (sub_409860) has a flattening score of 0.9714285714285714.
```

https://github.com/mrphrazer/obfuscation_detection

Plugin Manager



Summary

- common approaches to navigate in large binaries
- architecture-agnostic detection heuristics to pinpoint intesting code constructs
- useful in many reverse engieering scenarios

`https://github.com/mrphrazer/obfuscation_detection/`

Tim Blazytko

`@mr_phrazer`

`synthesis.to`

tim@blazytko.to

References

- "Automated Detection of Obfuscated Code" by Tim Blazytko
  `https://synthesis.to/2021/08/10/obfuscation_detection.html`
- "Automated Detection of Control-flow Flattening" by Tim Blazytko
  `https://synthesis.to/2021/03/03/flattening_detection.html`
- "Statistical Analysis to Detect Uncommon Code" by Tim Blazytko
  `https://synthesis.to/2023/01/26/uncommon_instruction_sequences.html`