ScatterBrain: Unmasking the Shadow of PoisonPlug's Obfuscator

By Mandiant

Published: 2025-01-28 · Archived: 2026-04-05 22:04:12 UTC

Written by: Nino Isakovic

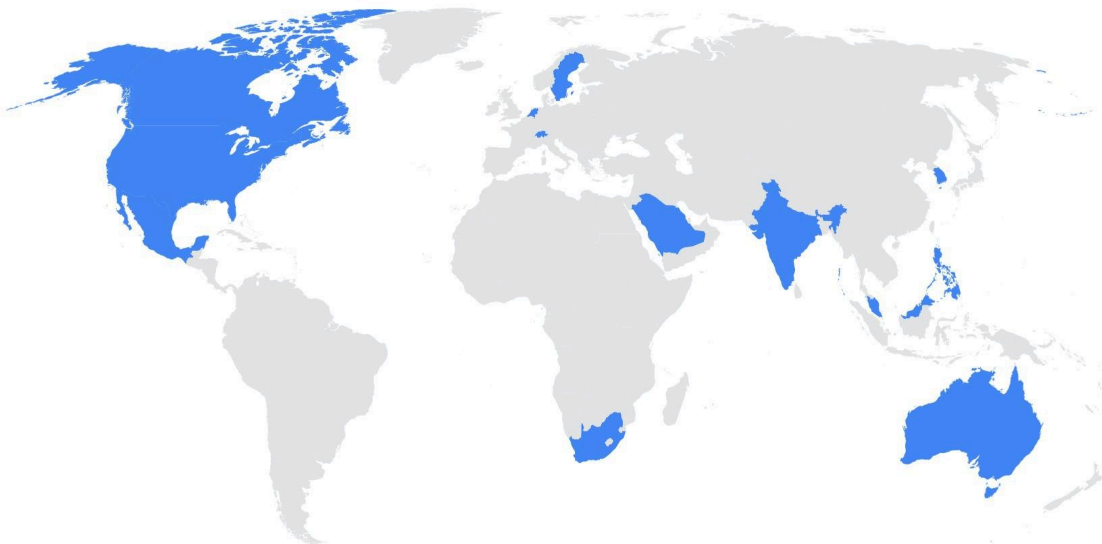
Introduction

Since 2022, Google Threat Intelligence Group (GTIG) has been tracking multiple cyber espionage operations conducted by China-nexus actors utilizing **POISONPLUG.SHADOW**. These operations employ a custom obfuscating compiler that we refer to as "**ScatterBrain**," facilitating attacks against various entities across Europe and the Asia Pacific (APAC) region. ScatterBrain appears to be a substantial evolution of ScatterBee, an obfuscating compiler previously analyzed by PWC.

GTIG assesses that POISONPLUG is an advanced modular backdoor used by multiple distinct, but likely related threat groups based in the PRC, however we assess that POISONPLUG.SHADOW usage appears to be further restricted to clusters associated with APT41.

GTIG currently tracks three known POISONPLUG variants:

- POISONPLUG
- POISONPLUG.DEED
- POISONPLUG.SHADOW



POISONPLUG.SHADOW—often referred to as "Shadowpad," a malware family name first introduced by Kaspersky—stands out due to its use of a custom obfuscating compiler specifically designed to evade detection and analysis. Its complexity is compounded by not only the extensive obfuscation mechanisms employed but also by the attackers' highly sophisticated threat tactics. These elements collectively make analysis exceptionally challenging and complicate efforts to identify, understand, and mitigate the associated threats it poses.

In addressing these challenges, GTIG collaborates closely with the FLARE team to dissect and analyze POISONPLUG.SHADOW. This partnership utilizes state-of-the-art reverse engineering techniques and comprehensive threat intelligence capabilities required to mitigate the sophisticated threats posed by this threat actor. We remain dedicated to advancing methodologies and fostering innovation to adapt to and counteract the ever-evolving tactics of threat actors, ensuring the security of Google and our customers against sophisticated cyber espionage operations.

Overview

In this blog post, we present our in-depth analysis of the ScatterBrain obfuscator, which has led to the development of a complete stand-alone static deobfuscator library independent of any binary analysis frameworks. Our analysis is based solely on the obfuscated samples we have successfully recovered, as we do not possess the obfuscating compiler itself. Despite this limitation, we have been able to comprehensively infer every aspect of the obfuscator and the necessary requirements to break it. Our analysis further reveals that ScatterBrain is continuously evolving, with incremental changes identified over time, highlighting its ongoing development.

This publication begins by exploring the fundamental primitives of ScatterBrain, outlining all its components and the challenges they present for analysis. We then detail the steps required to subvert and remove each protection mechanism, culminating in our deobfuscator. Our library takes protected binaries generated by ScatterBrain as input and produces fully functional deobfuscated binaries as output.

By detailing the inner workings of ScatterBrain and sharing our deobfuscator, we hope to provide valuable insights into developing effective countermeasures. Our blog post is intentionally exhaustive, drawing from our experience in dealing with obfuscation for clients, where we observed a significant lack of clarity in understanding modern obfuscation techniques. Similarly, analysts often struggle with understanding even relatively simplistic obfuscation methods primarily because standard binary analysis tooling is not designed to account for them. Therefore, our goal is to alleviate this burden and help enhance the collective understanding against commonly seen protection mechanisms.

For general questions about obfuscating compilers, we refer to our previous work on the topic, which provides an introduction and overview.

ScatterBrain Obfuscator

Introduction

ScatterBrain is a sophisticated obfuscating compiler that integrates multiple operational modes and protection components to significantly complicate the analysis of the binaries it generates. Designed to render modern binary analysis frameworks and defender tools ineffective, ScatterBrain disrupts both static and dynamic analyses.

- **Protection Modes:** ScatterBrain operates in three distinct modes, each determining the overall structure and intensity of the applied protections. These modes allow the compiler to adapt its obfuscation strategies based on the specific requirements of the attack.
- **Protection Components:** The compiler employs key protection components that include the following:
  - **Selective or Full Control Flow Graph (CFG) Obfuscation:** This technique restructures the program's control flow, making it very difficult to analyze and create detection rules for.
  - **Instruction Mutations:** ScatterBrain alters instructions to obscure their true functionality without changing the program's behavior.
  - **Complete Import Protection:** ScatterBrain employs a complete protection of a binary's import table, making it extremely difficult to understand how the binary interacts with the underlying operating system.

These protection mechanisms collectively make it extremely challenging for analysts to deconstruct and understand the functionality of the obfuscated binaries. As a result, ScatterBrain poses a formidable obstacle for cybersecurity professionals attempting to dissect and mitigate the threats it generates.

Modes of Operation

A mode refers to how ScatterBrain will transform a given binary into its obfuscated representation. It is distinct from the actual core obfuscation mechanisms themselves and is more about the overall strategy of applying protections. Our analysis further revealed a consistent pattern in applying various protection modes at specific stages of an attack chain:

- **Selective:** A group of individually selected functions are protected, leaving the remainder of the binary in its original state. Any import references within the selected functions are also obfuscated. This mode was observed to be used strictly for dropper samples of an attack chain.
- **Complete:** The entirety of the code section and all imports are protected. This mode was applied solely to the plugins embedded within the main backdoor payload.
- **Complete "headerless":** This is an extension of the **Complete** mode with added data protections and the removal of the PE header. This mode was exclusively reserved for the final backdoor payload.

Selective

The selective mode of protection allows users of the obfuscator to selectively target individual functions within the binary for protection. Protecting an individual function involves keeping the function at its original starting address (produced by the original compiler and linker) and substituting the first instruction with a jump to the obfuscated code. The generated obfuscations are stored linearly from this starting point up to a designated "end marker" that signifies the ending boundary of the applied protection. This entire range constitutes a protected function.

The disassembly of a call site to a protected function can take the following from:

```
 .text:180001000 sub     rsp, 28h
.text:180001004 mov     rcx, cs:g_Imagebase
```

```
.text:18000100B call    PROTECTED_FUNCTION ; call to protected func
.text:180001010 mov     ecx, eax
.text:180001012 call    cs:ExitProcess
```

Figure 1: Disassembly of a call to a protected function

The start of the protected function:

```
 .text:180001039 PROTECTED_FUNCTION
.text:180001039   jmp     loc_18000DF97 ; jmp into obfuscated code
.text:180001039 sub_180001039 endp
.text:000000018000103E db  48h ; H.     ; garbage data
.text:000000018000103F db 0FFh
.text:0000000180001040 db 0C1h
```

Figure 2: Disassembly inside of a protected function

The "end marker" consists of two sets of padding instructions, an `int3` instruction and a single `multi-nop` instruction:

```
 END_MARKER:
.text:18001A95C   CC CC CC CC CC CC CC CC CC CC 66
66 0F 1F 84 00 00 00 00 00
.text:18001A95C int     3
.text:18001A95D int     3
.text:18001A95E int     3
.text:18001A95F int     3
.text:18001A960 int     3
.text:18001A961 int     3
.text:18001A962 int     3
.text:18001A963 int     3
.text:18001A964 int     3
.text:18001A965 int     3
.text:18001A966 db      66h, 66h   ; @NOTE: IDA doesn't disassemble properly
.text:18001A966 nop     word ptr [rax+rax+00000000h]
; -------------------------------------------------------------------------
; next, original function
.text:18001A970 ; [0000001F BYTES: COLLAPSED FUNCTION
__security_check_cookie. PRESS CTRL-NUMPAD+ TO EXPAND]
```

Figure 3: Disassembly listing of an end marker

Complete

The complete mode protects every function within the `.text` section of the binary, with all protections integrated directly into a single code section. There are no end markers to signify protected regions; instead, every function is uniformly protected, ensuring comprehensive coverage without additional sectioning.

This mode forces the need for some kind of deobfuscation tooling. Whereas selective mode only protects the selected functions and leaves everything else in its original state, this mode makes the output binary extremely difficult to analyze without accounting for the obfuscation.

Complete Headerless

This complete mode extends the complete approach to add further data obfuscations alongside the code protections. It is the most comprehensive mode of protection and was observed to be exclusively limited to the final payloads of an attack chain. It incorporates the following properties:

- Full PE header of the protected binary is removed.
- Custom loading logic (a loader) is introduced.
  - Becomes the entry point of the protected binary
  - Responsible of ensuring the protected binary is functional
  - Includes the option of mapping the final payload within a separate memory region distinct from the initial memory region it was loaded in
- Metadata is protected via hash-like integrity checks.
  - The metadata is utilized by the loader as part of its initialization sequence.
- Import protection will require relocation adjustments.
  - Done through an "import fixup table"

The loader’s entry routine crudely merges with the original entry of the binary by inserting multiple `jmp` instructions to bridge the two together. The following is what the entry point looks like after running our deobfuscator against a binary protected in headerless mode.

```
140001000 ; __int64 __fastcall start()
140001000 public start
140001000 start proc near
140001000
140001000 killProcess= dword ptr  8
140001000 killThread= dword ptr  10h
140001000
140001000 push    rcx
140001001 push    rdx
140001002 push    r8
140001004 push    r9
140001006 sub     rsp, 28h
14000100A call    LoaderMain  ; initialize the headerless payload in full
14000100F add     rsp, 28h
140001013 pop     r9
140001015 pop     r8
140001017 pop     rdx
140001018 pop     rcx
140001019 jmp     $+5
14000101E jmp     $+5
140001023 jmp     $+5
140001028 mov     rax, rsp
14000102B push    rbx
14000102C sub     rsp, 20h
140001030 and     dword ptr [rax+10h], 0
140001034 and     dword ptr [rax+8], 0
140001038 lea     rdx, [rax+killProcess]
14000103C lea     rcx, [rax+killThread]
140001040 call    shadow::main
140001045 cmp     [rsp+28h+killProcess], 0
14000104A mov     ebx, eax
14000104C jz      loc_140001063
140001052 call    cs:GetCurrentProcess
140001058 mov     edx, ebx                            ; uExitCode
14000105A mov     rcx, rax                            ; hProcess
14000105D call    cs:TerminateProcess
140001063
140001063 loc_140001063:                              ; CODE XREF: start+4C↑j
140001063 cmp     [rsp+28h+killThread], 0
140001068 jz      loc_14000107F
14000106E call    cs:GetCurrentThread
140001074 mov     edx, ebx                            ; dwExitCode
140001076 mov     rcx, rax                            ; hThread
140001079 call    cs:TerminateThread
14000107F
14000107F loc_14000107F:                              ; CODE XREF: start+68↑j
14000107F mov     eax, ebx
140001081 add     rsp, 20h
140001085 pop     rbx
140001086 retn
140001086 start endp
```

Figure 4: Deobfuscated loader entry

The loader's metadata is stored in the `.data` section of the protected binary. It is found via a memory scan that applies bitwise XOR operations against predefined constants. The use of these not only locates the metadata but also serves a dual purpose of verifying its integrity. By checking that the data matches expected patterns when XORed with these constants, the loader ensures that the metadata has not been altered or tampered with.

```
// use the return address of the call the
// function as the starting point for the scan
curr = retaddr;
do {
  do {
    v1 = *(_DWORD *)((char *)curr + 5);
    curr = (_DWORD *)((char *)curr + 1);
  } while ( *curr != (v1 ^ 0x97E8027D) );
}
while ( *curr != (curr[2] ^ 0xF3A300F6) || *curr != (curr[3] ^ 0x858AF28D) );
return curr;
```

Figure 5: Memory scan to identify the loader's metadata inside the .data section

The metadata contains the following (in order):

- Import fixup table (fully explained in the Import Protection section)
- Integrity-hash constants
- Relative virtual address (RVA) of the `.data` section
- Offset to the import fixup table from the start of the `.data` section
- Size, in bytes, of the fixup table
- Global pointer to the memory address that the backdoor is at
- Encrypted and compressed data specific to the backdoor
    - Backdoor config and plugins

```
0001400675AC dd 6252Fh
0001400675B0 dd 659E8h
0001400675B4 dd 62E8Fh
0001400675B8 dd 659F0h
0001400675BC dd 65999h                ; ImpTable entries ^^

0001400675C0 dd 0DD256F8Ah            ; Integrity-hash dwords
0001400675C4 dd 4ACD6DF7h
0001400675C8 dd 2E866F7Ch
0001400675CC dd 58AF9D07h

0001400675D0 dd 62000h                ; RVA of the .data section
0001400675D4 dd 4000h                 ; offset to ImpTable from the .data section
0001400675D8 dd 15C0h                 ; size (in bytes) of the ImpTable

0001400675DC dd 0                     ; mapped, final payload buffer
0001400675E0 dd 0                     ; maxSize
0001400675E4 dd 0                     ; size

0001400675E8 dd 0A000087Bh            ; First data blob: 0A (type), 0x87B (size)
0001400675EC db 45h, 3Ch, 0B8h, 99h, 0F8h, 43h, 7Dh, 0DCh, 0AEh, 0CEh, 78h, 7Bh,
00014006760C db 32h, 0DBh, 0A9h, 42h, 10h, 7Dh, 75h, 57h, 0F4h, 5Bh, 0C2h, 64h,
00014006762C db 82h, 0D7h, 0D6h, 83h, 46h, 91h, 0D8h, 8Fh, 9Fh, 6Ah, 0A3h, 85h,
00014006764C db 31h, 2Ah, 5Ah, 0F9h, 0Ch, 5Ah, 7Ah, 95h, 0BEh, 0E7h, 3, 0FEh, 0F
00014006766C db 0B3h, 0B6h, 9Bh, 70h, 20h, 0A3h, 47h, 7Fh, 0AAh, 2Fh, 15h, 85h,
00014006768C db 2Dh, 6Ah, 0Fh, 50h, 92h, 32h, 0D5h, 0C6h, 47h, 2Eh, 65h, 87h, 60
0001400676AC db 6, 5Bh, 6, 3Dh, 0EFh, 0C5h, 9Ch, 9Fh, 0D4h, 0ACh, 0E3h, 0C2h, 3D
0001400676CC db 16h, 41h, 37h, 98h, 0C8h, 0D2h, 7Eh, 0FDh, 0CFh, 0D7h, 5Eh, 0DFh
0001400676EC db 29h, 0A4h, 22h, 84h, 7Eh, 31h, 1Fh, 1Dh, 2Eh, 0CDh, 1Dh, 4Bh, 0E
```

Figure 6: Loader's metadata

Core Protection Components

Instruction Dispatcher

The **instruction dispatcher** is the central protection component that transforms the natural control flow of a binary (or individual function) into scattered basic blocks that end with a unique dispatcher routine that dynamically guides the execution of the protected binary.

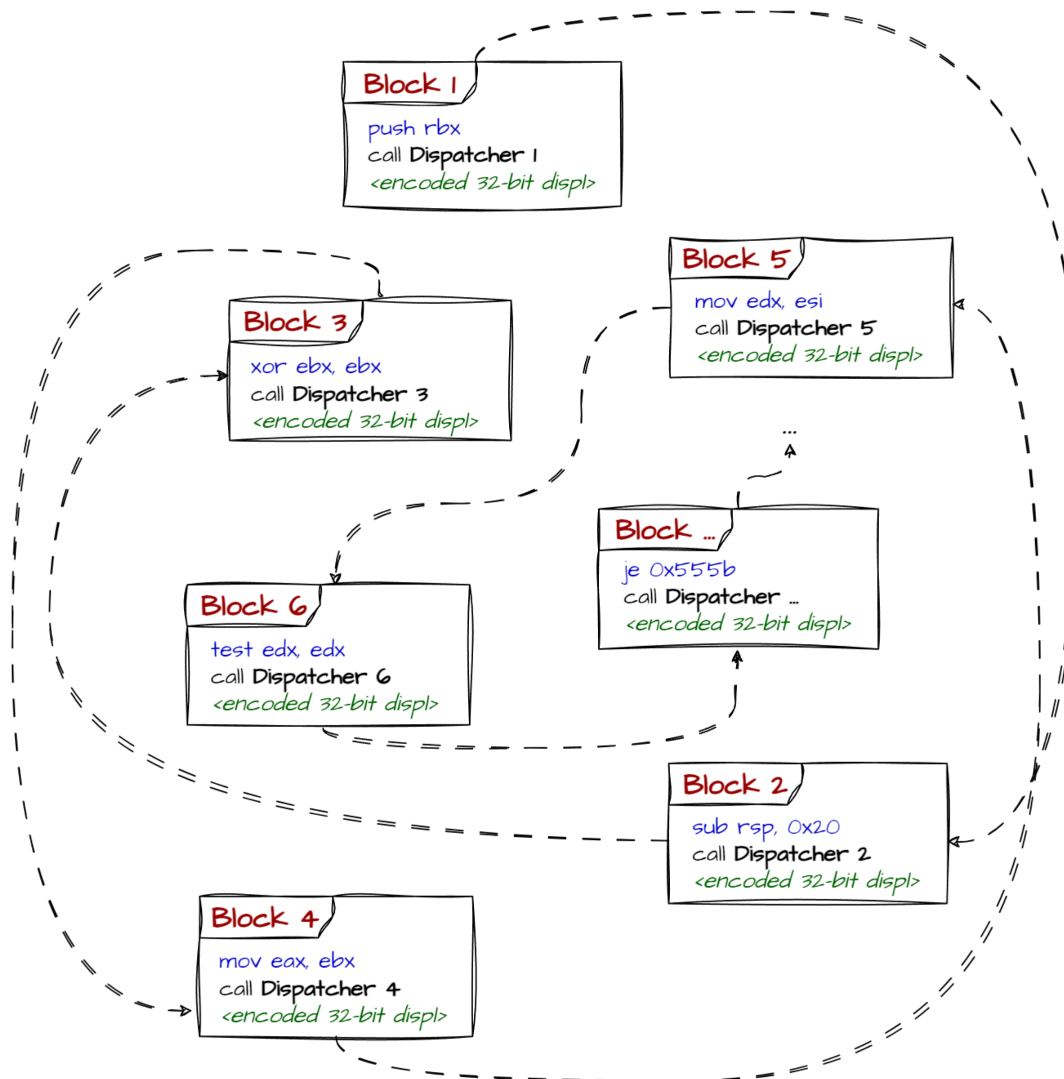


Figure 7: Illustration of the control flow instruction dispatchers induce

Each call to a dispatcher is immediately followed by a 32-bit encoded displacement positioned at what would normally be the return address for the call. The dispatcher decodes this displacement to calculate the destination target for the next group of instructions to execute. A protected binary can easily contain thousands or even tens of thousands of these dispatchers making manual analysis of them practically infeasible. Additionally, the dynamic dispatching and decoding logic employed by each dispatcher effectively disrupts CFG reconstruction methods used by all binary analysis frameworks.

The decoding logic is unique for each dispatcher and is carried out using a combination of `add`, `sub`, `xor`, `and`, `or`, and `lea` instructions. The decoded offset value is then either subtracted from or added to the expected return address of the dispatcher call to determine the final destination address. This calculated address directs execution to the next block of instructions, which will similarly end with a dispatcher that uniquely decodes and jumps to subsequent instruction blocks, continuing this process iteratively to control the program flow.

The following screenshot illustrates what a dispatcher instance looks like when constructed in IDA Pro. Notice the scattered addresses present even within instruction dispatchers, which result from the obfuscator transforming fallthrough instructions—instructions that naturally follow the preceding instruction—into pairs of conditional branches that use opposite conditions. This ensures that one branch is always taken, effectively creating an unconditional jump. Additionally, a `mov` instruction that functions as a no-op is inserted to split these branches, further obscuring the control flow.

```
001A3EF mov     [rsp+8], rbx          ; regular instruction
001A3F4 call    near ptr sub_65EE     ; call to dispatcher at 0x65EE
001A3F4 ; ---------------------------------------------------------------
001A3F9 dd 35FD646Bh                  ; encoded displacement unique to the dispatcher

; ---------------------------------------------------------------
00065EE push    rsi                   ; save rsi as it's used as the "working register"
00065EF jl      loc_34475
00065F5 mov     si, si
00065F8 jge     loc_34475

0034475 mov     rsi, [rsp+8]          ; fetch the return address
003447A jmp     loc_612A5

00612A5 movsxd  rsi, dword ptr [rsi]  ; read the encoded 32-bit displacement
00612A8 js      loc_1DBF
00612AE mov     r9, r9
00612B1 jns     loc_1DBF

0001DBF pushfq                        ; save RFLAGS before decoding
0001DC0 jb      loc_5D29F
0001DC6 xchg    bx, bx
0001DC9 jnb     loc_5D29F

005D29F xor     rsi, 35FCF022h        ; decode displacement using unique 32-bit magic constant
005D2A6 ja      loc_6D01
005D2AC xchg    al, al
005D2AE jbe     loc_6D01

0006D01 sub     [rsp+10h], rsi        ; calculate the destination
0006D06 jo      loc_45D18
0006D0C mov     dl, dl
0006D0E jno     loc_45D18

0045D18 popfq                         ; restore RFLAGS after decoding
0045D19 jg      loc_19D28
0045D1F mov     r11, r11
0045D22 jle     loc_19D28

0019D28 pop     rsi                   ; restore the working register
0019D29 jb      locret_42EC4
0019D2F xchg    cl, cl
0019D31 jnb     locret_42EC4

0042EC4 retn                          ; dispatch to decoded destination
; ---------------------------------------------------------------
```

Figure 8: Example of an instruction dispatcher and all of its components

The core logic for any dispatcher can be categorized into the following four phases:

1. **Preservation of Execution Context**
    - Each dispatcher selects a single working register (e.g., `RSI` as depicted in the screenshot) during the obfuscation process. This register is used in conjunction with the stack to carry out the intended decoding operations and dispatch.

- The `RFLAGS` register in turn is safeguarded by employing `pushfq` and `popfq` instructions before carrying out the decoding sequence.

2. **Retrieval of Encoded Displacement**
    - Each dispatcher retrieves a 32-bit encoded displacement located at the return address of its corresponding call instruction. This encoded displacement serves as the basis for determining the next destination address.
3. **Decoding Sequence**
    - Each dispatcher employs a unique decoding sequence composed of the following arithmetic and logical instructions: `xor` , `sub` , `add` , `mul` , `imul` , `div` , `idiv` , `and` , `or` , and `not` . This variability ensures that no two dispatchers operate identically, significantly increasing the complexity of the control flow.
4. **Termination and Dispatch**
    - The `ret` instruction is strategically used to simultaneously signal the end of the dispatcher function and redirect the program's control flow to the previously calculated destination address.

It is reasonable to infer that the obfuscator utilizes a template similar to the one illustrated in Figure 9 when applying its transformations to the original binary:

```
; --------------------------------------------------------------------------------
; WR:                     working-register
; MAGIC32:                unique 32-bit decoding constant
; DECODING_INSTRUCTION:   one of the 6 known decoding instructions
; CALC_INSTRUCTION:       sub/add to calculate the target after decoding
; --------------------------------------------------------------------------------
push WR {
  mov WR, [rsp+8]
  movsxd  WR, dword ptr [WR]
  pushfq {
     DECODING_INSTRUCTION WR, MAGIC32
     CALC_INSTRUCTION [rsp+10h], WR
  } popfq
} pop WR
retn
```

Figure 9: Instruction dispatcher template

Opaque Predicates

ScatterBrain uses a series of seemingly trivial opaque predicates (OP) that appear straightforward to analysts but significantly challenge contemporary binary analysis frameworks, especially when used collectively. These opaque predicates effectively disrupt static CFG recovery techniques not specifically designed to counter their logic. Additionally, they complicate symbolic execution approaches as well by inducing path explosions and hindering path prioritization. In the following sections, we will showcase a few examples produced by ScatterBrain.

`test` OP

This opaque predicate is constructed around the behavior of the `test` instruction when paired with an immediate zero value. Given that the `test` instruction effectively performs a bitwise AND operation, the obfuscator exploits the fact that any value bitwise AND-ed with zero always invariably results in zero.

Here are some abstracted examples we can find in a protected binary—abstracted in the sense that all instructions are not guaranteed to follow one another directly; other forms of mutations can be between them as can instruction dispatchers.

```
 test    bl, 0
jnp     loc_56C96            ; we never satisfy these conditions
------------------------------
test    r8, 0
jo      near ptr loc_3CBC8
------------------------------
test    r13, 0
jnp     near ptr loc_1A834
------------------------------
test    eax, 0
jnz     near ptr loc_46806
```

Figure 10: Test opaque predicate examples

To grasp the implementation logic of this opaque predicate, the semantics of the `test` instruction and its effects on the processor's flags register are required. The instruction can affect six different flags in the following manner:

- **Overflow Flag (OF):** Always cleared
- **Carry Flag (CF):** Always cleared
- **Sign Flag (SF):** Set if the most significant bit (MSB) of the result is set; otherwise cleared
- **Zero Flag (ZF):** Set if the result is 0; otherwise cleared
- **Parity Flag (PF):** Set if the number of set bits in the least significant byte (LSB) of the result is even; otherwise cleared
- **Auxiliary Carry Flag (AF):** Undefined

Applying this understanding to the sequences produced by ScatterBrain, it is evident that the generated conditions can never be logically satisfied:

Sequence	**Condition Description**
`test <reg>, 0; jo`	`OF` is always cleared

`test <reg>, 0; jnae/jc/jb`	`CF` is always cleared
`test <reg>, 0; js`	Resulting value will always be zero; therefore, `SF` can never be set
`test <reg>, 0; jnp/jpo`	The number of bits in zero is always zero, which is an even number; therefore, `PF` can never be set
`test <reg>, 0; jne/jnz`	Resulting value will always be zero; therefore, `ZF` will always be set

Table 1: Test opaque predicate understanding

`jcc` **OP**

The opaque predicate is designed to statically obscure the original immediate branch targets for conditional branch (`jcc`) instructions. Consider the following examples:

```
 test    eax, eax
ja      loc_3BF9C
ja      loc_2D154

test    r13, r13
jns     loc_3EA84
jns     loc_53AD9

test    eax, eax
jnz     loc_99C5
jnz     loc_121EC

cmp     eax, FFFFFFFF
jz      loc_273EE
jz      loc_4C227
```

Figure 11: jcc opaque predicate examples

The implementation is straightforward: each original `jcc` instruction is duplicated with a bogus branch target. Since both `jcc` instructions are functionally identical except for their respective branch destinations, we can determine with certainty that the first `jcc` in each pair is the original instruction. This original `jcc` dictates the

correct branch target to follow when the respective condition is met, while the duplicated `jcc` serves to confuse analysis tools by introducing misleading branch paths.

Stack-Based OP

The stack-based opaque predicate is designed to check whether the current stack pointer (`rsp`) is below a predetermined immediate threshold—a condition that can never be true. It is consistently implemented by pairing the `cmp rsp` instruction with a `jb` (jump if below) condition immediately afterward.

```
 cmp rsp, 0x8d6e
jb      near ptr unk_180009FDA
```

Figure 12: Stack-based opaque predicate example

This technique inserts conditions that are always false, causing CFG algorithms to follow both branches and thereby disrupt their ability to accurately reconstruct the control flow.

Import Protection

The obfuscator implements a sophisticated import protection layer. This mechanism conceals the binary 's dependencies by transforming each original `call` or `jmp` instruction directed at an import through a unique stub dispatcher routine that knows how to dynamically resolve and invoke the import in question.

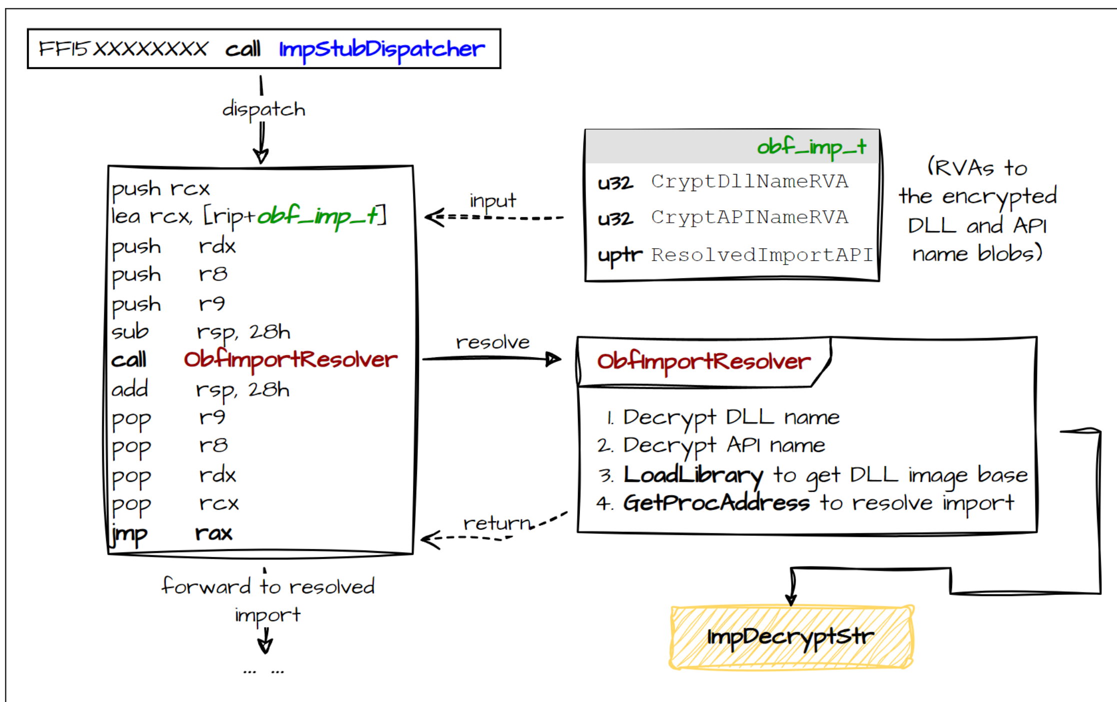


Figure 13: Illustration of all the components involved in the import protection

It consists of the following components:

- **Import-specific encrypted data:** Each protected import is represented by a unique dispatcher stub and a scattered data structure that stores RVAs to both the encrypted dynamic-link library (DLL) and application programming interface (API) names. We refer to this structure as `obf_imp_t`. Each dispatcher stub is hardcoded with a reference to its respective `obf_imp_t`.
- **Dispatcher stub:** This is an obfuscated stub that dynamically resolves and invokes the intended import. While every stub shares an identical template, each contains a unique hardcoded RVA that identifies and locates its corresponding `obf_imp_t`.
- **Resolver routine:** Called from the dispatcher stub, this obfuscated routine resolves the import and returns it to the dispatcher, which facilitates the final call to the intended import. It begins by locating the encrypted DLL and API names based on the information in `obf_imp_t`. After decrypting these names, the routine uses them to resolve the memory address of the API.
- **Import decryption routine:** Called from the resolver routine, this obfuscated routine is responsible for decrypting the DLL and API name blobs through a custom stream cipher implementation. It uses a hardcoded 32-bit salt that is unique per protected sample.
- **Fixup Table:** Present only in headerless mode, this is a relocation fixup table that the loader in headerless mode uses to correct all memory displacements to the following import protection components:
    - Encrypted DLL names
    - Encrypted API names
    - Import dispatcher references

Dispatcher Stub

The core of the import protection mechanism is the **dispatcher stub**. Each stub is tailored to an individual import and consistently employs a `lea` instruction to access its respective `obf_imp_t`, which it passes as the only input to the resolver routine.

```
 push rcx                  ; save RCX
lea rcx, [rip+obf_imp_t]  ; fetch import-specific obf_imp_t
push    rdx               ; save all other registers the stub uses
push    r8
push    r9
sub     rsp, 28h
call    ObfImportResolver ; resolve the import and return it in RAX
add     rsp, 28h
pop     r9                ; restore all saved registers
pop     r8
pop     rdx
pop     rcx
jmp     rax               ; invoke resolved import
```

Figure 14: Deobfuscated import dispatcher stub

Each stub is obfuscated through the mutation mechanisms outlined earlier. This applies to the resolver and import decryption routines as well. The following is what the execution flow of a stub can look like. Note the scattered addresses that while presented sequentially are actually jumping all around the code segment due to the instruction dispatchers.

```
 0x01123a   call InstructionDispatcher_TargetTo_11552
0x011552   push rcx
0x011553   call InstructionDispatcher_TargetTo_5618
0x005618   lea rcx, [rip+0x33b5b]                    ; fetch obf_imp_t
0x00561f   call InstructionDispatcher_TargetTo_f00c
0x00f00c   call InstructionDispatcher_TargetTo_191b5
0x0191b5   call InstructionDispatcher_TargetTo_1705a
0x01705a   push rdx
0x01705b   call InstructionDispatcher_TargetTo_05b4
0x0105b4   push r8
0x0105b6   call InstructionDispatcher_TargetTo_f027
0x00f027   push r9
0x00f029   call InstructionDispatcher_TargetTo_18294
0x018294   test eax, 0
0x01829a   jo 0xf33c
0x00f77b   call InstructionDispatcher_TargetTo_e817
0x00e817   sub rsp, 0x28
0x00e81b   call InstructionDispatcher_TargetTo_a556
0x00a556   call 0x6afa (ObfImportResolver)
0x00a55b   call InstructionDispatcher_TargetTo_19592
0x019592   test ah, 0
0x019595   call InstructionDispatcher_TargetTo_a739
0x00a739   js 0x1935
0x00a73b   call InstructionDispatcher_TargetTo_6eaa
0x006eaa   add rsp, 0x28
0x006eae   call InstructionDispatcher_TargetTo_6257
0x006257   pop r9
0x006259   call InstructionDispatcher_TargetTo_66d6
0x0066d6   pop r8
0x0066d8   call InstructionDispatcher_TargetTo_1a3cb
0x01a3cb   pop rdx
0x01a3cc   call InstructionDispatcher_TargetTo_67ab
0x0067ab   pop rcx
0x0067ac   call InstructionDispatcher_TargetTo_6911
0x006911   jmp rax
```

Figure 15: Obfuscated import dispatcher stub

Resolver Logic

`obf_imp_t` is the central data structure that contains the relevant information to resolve each import. It has the following form:

```
struct obf_imp_t {    // sizeof=0x18
   uint32_t CryptDllNameRVA;    // NOTE: will be 64-bits, due to padding
   uint32_t CryptAPINameRVA;    // NOTE: will be 64-bits, due to padding
   uint64_t ResolvedImportAPI;  // Where the resolved address is stored
};
```

Figure 16: obf_imp_t in its original C struct source form

It is processed by the resolver routine, which uses the embedded RVAs to locate the encrypted DLL and API names, decrypting each in turn. After decrypting each name blob, it uses `LoadLibraryA` to ensure the DLL dependency is loaded in memory and leverages `GetProcAddress` to retrieve the address of the import.

Fully decompiled `ObfImportResolver`:

```
uint64_t __fastcall ObfImportResolver(obf_imp_t *a1) {
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  // check if api already resolved
  ResolvedImportAPI = a1->ResolvedImportAPI;
  if ( !ResolvedImportAPI ) {
    // decrypt current dll name and ensure it's in memory
    dllname = ImpDecryptStr(a1->CryptDllNameRVA);
    LoadLibraryA = CryptResolveLoadLibraryA();
    dll = LoadLibraryA(dllname);
    if ( dll ) {
      // decrypt api name and resolve it
      apiname = ImpDecryptStr(a1->CryptAPINameRVA);
      GetProcAddress = CryptResolveGetProcAddress();
      ResolvedImportAPI = GetProcAddress(dll, apiname);
      a1->ResolvedImportAPI = ResolvedImportAPI;
    }
  }
  // resolved api
  return ResolvedImportAPI;
}
```

Figure 17: Fully decompiled import resolver routine

Import Encryption Logic

The import decryption logic is implemented using a Linear Congruential Generator (LCG) algorithm to generate a pseudo-random key stream, which is then used in a XOR-based stream cipher for decryption. It operates on the following formula:

$X_n + 1 = (a \cdot X_n + c) \bmod 2^{32}$

where:

- `a` is always hardcoded to `17` and functions as the multiplier
- `c` is a unique 32-bit constant determined by the encryption context and is unique per-protected sample
    - We refer to it as the `imp_decrypt_const`
- `mod` 2^{32} confines the sequence values to a 32-bit range

The decryption logic initializes with a value from the encrypted data and iteratively generates new values using the outlined LCG formula. Each iteration produces a byte derived from the calculated value, which is then XOR'ed with the corresponding encrypted byte. This process continues byte-by-byte until it reaches a termination condition.

A fully recovered Python implementation for the decryption logic is provided in Figure 18.

```python
def imp_crypt_str(
    imp_decrypt_const: int,
    encrypted_bytes: bytes
):
    decrypted_bytes = bytearray()

    # extract initial value directly from the encrypted_bytes
    current_value = int.from_bytes(encrypted_bytes[:4], 'little')

    # max length, as specified by the initial algo from the obfuscator
    MAX_LENGTH = 0x400
    for index in range(MAX_LENGTH):
        calculated_value = (17 * current_value - imp_decrypt_const) & 0xFFFFFFFF
        value_bytes = calculated_value.to_bytes(4, 'little')
        sum_value_bytes = sum(value_bytes) & 0xFF

        # chkif end of the encrypted bytes
        if index + 4 >= len(encrypted_bytes): break

        # fetch the corresponding encrypted byte by offsetting index
        encrypted_byte = encrypted_bytes[index + 4]

        # decrypt && append
        decrypted_byte = encrypted_byte ^ sum_value_bytes
        decrypted_bytes.append(decrypted_byte)

        # chkif decrypted byte marks the end of sequence
        if encrypted_byte == sum_value_bytes: break

        # prep next cycle
        current_value = calculated_value
    return decrypted_bytes[:-1].decode("ascii")
```

Figure 18: Complete Python implementation of the import string decryption routine

Import Fixup Table

The import relocation fixup table is a fixed-size array composed of two 32-bit RVA entries. The first RVA represents the memory displacement of where the data is referenced from. The second RVA points to the actual

data in question. The entries in the fixup table can be categorized into three distinct types, each corresponding to a specific import component:

- Encrypted DLL names
- Encrypted API names
- Import dispatcher references

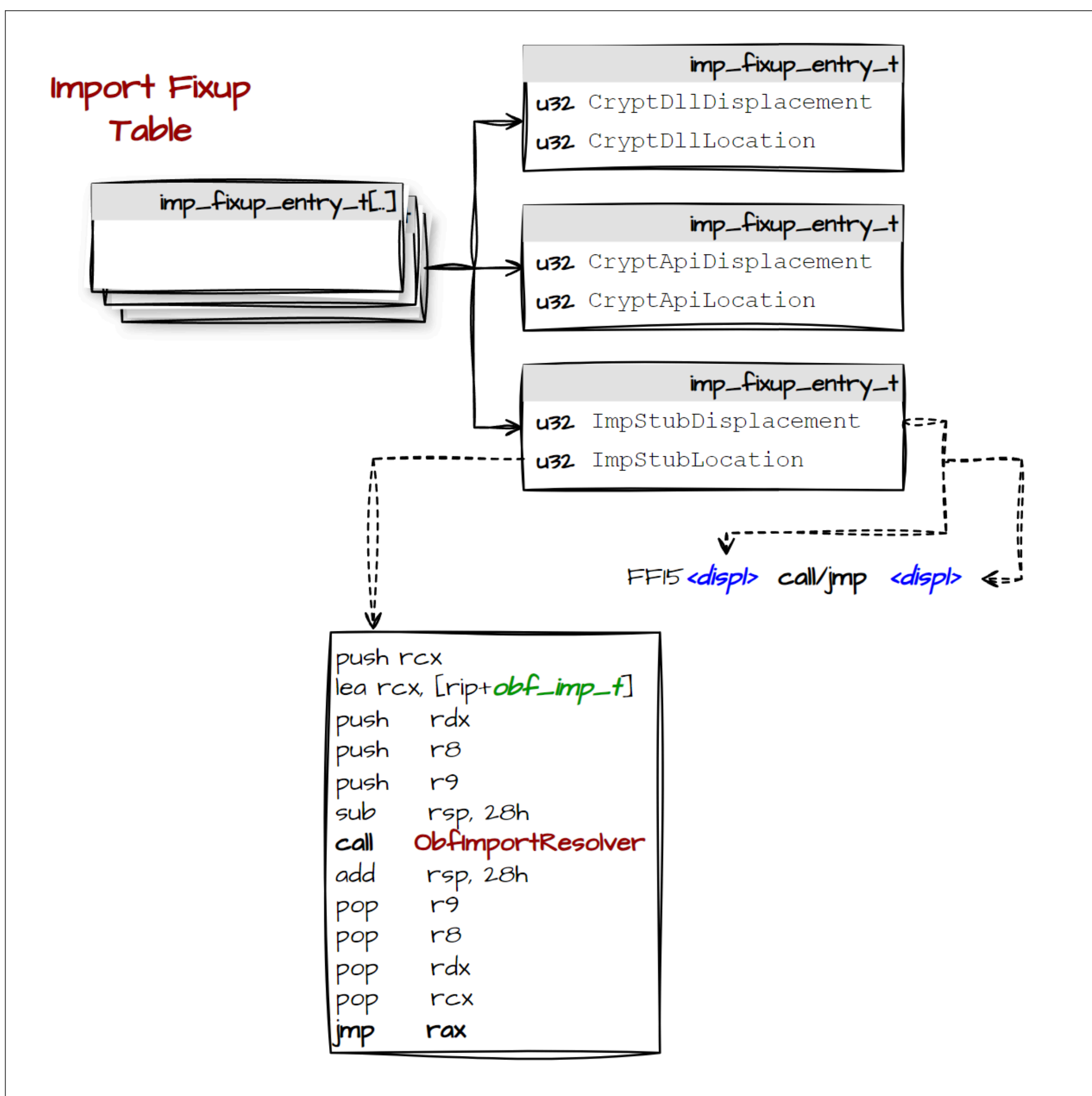


Figure 19: Illustration of the import fixup table

The location of the fixup table is determined by the loader's metadata, which specifies an offset from the start of the `.data` section to the start of the table. During initialization, the loader is responsible for applying the relocation fixups for each entry in the table.

```
0000000140067568 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0000000140067568 dd 65949h
000000014006756C dd 6278Eh
0000000140067570 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0000000140067570 dd 65959h
0000000140067574 dd 2AF5Bh
0000000140067578 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0000000140067578 dd 65961h
000000014006757C dd 35551h
0000000140067580 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0000000140067580 dd 65969h
0000000140067584 dd 65295h
0000000140067588 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0000000140067588 dd 65971h
000000014006758C dd 62863h
0000000140067590 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0000000140067590 dd 65981h
0000000140067594 dd 64ACFh
0000000140067598 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
0000000140067598 dd 65989h
000000014006759C dd 64F73h
00000001400675A0 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00000001400675A0 dd 659BEh
00000001400675A4 dd 65295h
00000001400675A8 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00000001400675A8 dd 659C6h
00000001400675AC dd 6252Fh
00000001400675B0 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00000001400675B0 dd 659E8h
00000001400675B4 dd 62E8Fh
00000001400675B8 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
00000001400675B8 dd 659F0h
00000001400675BC dd 65999h                             ; Import fixup table entries
00000001400675C0 //-------------------------------------------------------------------------
00000001400675C0 dd 0DD256F8Ah                         ; Integrity-hash dwords
00000001400675C4 dd 4ACD6DF7h
00000001400675C8 dd 2E866F7Ch
00000001400675CC dd 58AF9D07h
00000001400675D0 //-------------------------------------------------------------------------
00000001400675D0 dd 62000h                             ; RVA of the .data section
00000001400675D4 dd 4000h                              ; offset to Import Fixup Table from the .data section
00000001400675D8 dd 15C0h                              ; size (in bytes) of the Import Fixup Table
00000001400675DC //-------------------------------------------------------------------------
00000001400675DC dd 0                                  ; mapped, final payload buffer
00000001400675E0 dd 0                                  ; maxSize
00000001400675E4 dd 0                                  ; size
00000001400675E8 //-------------------------------------------------------------------------
00000001400675E8 dd 0A000087Bh                         ; Config data blob: 0A (type), 0x87B (size)
00000001400675EC db 45h, 3Ch, 0B8h, 99h, 0F8h, 43h, 7Dh, 0DCh, 0AEh, 0CEh, 78h, 7Bh, 65h, 4Ah, 79h, 14h, 1Bh,
000000014006760C db 32h, 0DBh, 0A9h, 42h, 10h, 7Dh, 75h, 57h, 0F4h, 5Bh, 0C2h, 64h, 0B6h, 3Fh, 82h, 5Ch, 6Dh,
000000014006762C db 82h, 0D7h, 0D6h, 83h, 46h, 91h, 0D8h, 8Fh, 9Fh, 6Ah, 0A3h, 85h, 1Fh, 7Fh, 0DAh, 3Ah, 7Fh,
```

Figure 20: Loader metadata that shows the Import fixup table entries and metadata used to find it

Recovery

Effective recovery from an obfuscated binary necessitates a thorough understanding of the protection mechanisms employed. While deobfuscation often benefits from working with an intermediate representation (IR) rather than the raw disassembly—an IR provides more granular control in undoing transformations—this obfuscator preserves the original compiled code, merely enveloping it with additional protection layers. Given this context, our deobfuscation strategy focuses on stripping away the obfuscator's transformations from the disassembly to reveal the original instructions and data. This is achieved through a series of hierarchical phases, where each subsequent phase builds upon the previous one to ensure comprehensive deobfuscation.

We categorize this approach into three distinct categories that we eventually integrate:

1. **CFG Recovery**
    - Restoring the natural control flow by removing obfuscation artifacts at the instruction and basic block levels. This involves two phases:
        - **Accounting for instruction dispatchers:** Addressing the core of control flow protection that obscure the execution flow
        - **Function identification and recovery:** Cataloging scattered instructions and reassembling them into their original function counterparts
2. **Import Recovery**
    - **Original Import Table:** The goal is to reconstruct the original import table, ensuring that all necessary library and function references are accurately restored.
3. **Binary Rewriting**
    - **Generating Deobfuscated Executables:** This process entails creating a new, deobfuscated executable that maintains the original functionality while removing ScatterBrain's modifications.

Given the complexity of each category, we concentrate on the core aspects necessary to break the obfuscator by providing a guided walkthrough of our deobfuscator's source code and highlighting the essential logic required to reverse these transformations. This step-by-step examination demonstrates how each obfuscation technique is methodically undone, ultimately restoring the binary's original structure.

Our directory structure reflects this organized approach:

```
 +---helpers
|   |    emu64.py
|   |    pefile_utils.py
|   |--- x86disasm.py
|
\---recover
    |    recover_cfg.py
    |    recover_core.py
    |    recover_dispatchers.py
    |    recover_functions.py
    |    recover_imports.py
    |--- recover_output64.py
```

Figure 21: Directory structure of our deobfuscator library

This comprehensive recovery process not only restores the binaries to their original state but also equips analysts with the tools and knowledge necessary to combat similar obfuscation techniques in the future.

CFG Recovery

The primary obstacle disrupting the natural control flow graph is the use of **instruction dispatchers**. Eliminating these dispatchers is our first priority in obtaining the CFG. Afterward, we need to reorganize the scattered instructions back into their original function representations—a problem known as function identification, which is notoriously difficult to generalize. Therefore, we approach it using our specific knowledge about the obfuscator.

Linearizing the Scattered CFG

Our initial step in recovering the original CFG is to eliminate the scattering effect induced by instruction dispatchers. We will transform all dispatcher call instructions into direct branches to their resolved targets. This transformation linearizes the execution flow, making it straightforward to statically pursue the second phase of our CFG recovery. This will be implemented via brute-force scanning, static parsing, emulation, and instruction patching.

Function Identification and Recovery

We leverage a recursive descent algorithm that employs a depth-first search (DFS) strategy applied to known entry points of code, attempting to exhaust all code paths by "single-stepping" one instruction at a time. We add additional logic to the processing of each instruction in the form of "mutation rules" that stipulate how each individual instruction needs to be processed. These rules aid in stripping away the obfuscator's code from the original.

Removing Instruction Dispatchers

Eliminating instruction dispatchers involves identifying each dispatcher location and its corresponding dispatch target. Recall that the target is a uniquely encoded 32-bit displacement located at the return address of the dispatcher call. To remove instruction dispatchers, it is essential to first understand how to accurately identify them. We begin by categorizing the defining properties of individual instruction dispatchers:

- **Target of a Near Call**
  - Dispatchers are always the destination of a near `call` instruction, represented by the `E8` opcode followed by a 32-bit displacement.
- **References Encoded 32-Bit Displacement at Return Address**
  - Dispatchers reference the encoded 32-bit displacement located at the return address on the stack by performing a 32-bit read from the stack pointer. This displacement is essential for determining the next execution target.
- **Pairing of `pushfq` and `popfq` Instructions to Safeguard Decoding**
  - Dispatchers use a pair of `pushfq` and `popfq` instructions to preserve the state of the `RFLAGS` register during the decoding process. This ensures that the dispatcher does not alter the original execution context, maintaining the integrity of register contents.
- **End with a `ret` Instruction**
  - Each dispatcher concludes with a `ret` instruction, which not only ends the dispatcher function but also redirects control to the next set of instructions, effectively continuing the execution flow.

Leveraging the aforementioned categorizations, we implement the following approach to identify and remove instruction dispatchers:

1. **Brute-Force Scanner for Near Call Locations**
    - Develop a scanner that searches for all near `call` instructions within the code section of the protected binary. This scanner generates a huge array of potential call locations that may serve as dispatchers.
2. **Implementation of a Fingerprint Routine**
    - The brute-force scan yields a large number of false positives, requiring an efficient method to filter them. While emulation can filter out false positives, it is computationally expensive to do it for the brute-force results.
    - Introduce a shallow fingerprinting routine that traverses the disassembly of each candidate to identify key dispatcher characteristics, such as the presence of `pushfq` and `popfq` sequences. This significantly improves performance by eliminating most false positives before concretely verifying them through emulation.
3. **Emulation of Targets to Recover Destinations**
    - Emulate execution starting from each verified call site to accurately recover the actual dispatch targets. Emulating from the call site ensures that the emulator processes the encoded offset data at the return address, abstracting away the specific decoding logic employed by each dispatcher.
    - A successful emulation also serves as the final verification step to confirm that we have identified a dispatcher.
4. **Identification of Dispatch Targets via ret Instructions**
    - Utilize the terminating `ret` instruction to accurately identify the dispatch target within the binary.
    - The `ret` instruction is a definitive marker indicating the end of a dispatcher function and the point at which control is redirected, making it a reliable indicator for target identification.

Brute-Force Scanner

The following Python code implements the brute-force scanner, which performs a comprehensive byte signature scan within the code segment of a protected binary. The scanner systematically identifies all potential `call` instruction locations by scanning for the `0xE8` opcode associated with near `call` instructions. The identified addresses are then stored for subsequent analysis and verification.

```python
def _brute_find_all_calls(
  imgbuffer: bytes,
  data_section_rva: int
) -> list[int]:
  """ Brute-force byte-signature search the provided image buffer for all
  potential near relative call (`e8` only) instructions.
  """
  CALL_BYTE              = bytes([0xe8])
  start_index            = 0
  call_locs: list[int] = []
  #--------------------------------------------------------------------
  MAX_LENGTH = min(len(imgbuffer), data_section_rva)
  while start_index < MAX_LENGTH:
    curr_index = imgbuffer.find(CALL_BYTE, start_index)
    if curr_index == -1: break
      call_locs.append(curr_index)
      start_index = curr_index + 1
  return call_locs
```

Figure 22: Python implementation of the brute-force scanner

Fingerprinting Dispatchers

The fingerprinting routine leverages the unique characteristics of instruction dispatchers, as detailed in the Instruction Dispatchers section, to statically identify potential dispatcher locations within a protected binary. This identification process utilizes the results from the prior brute-force scan. For each address in this array, the routine disassembles the code and examines the resulting disassembly listing to determine if it matches known dispatcher signatures.

This method is not intended to guarantee 100% accuracy, but rather serve as a cost-effective approach to identifying call locations with a high likelihood of being instruction dispatchers. Subsequent emulation will be employed to confirm these identifications.

1. **Successful Decoding of a `call` Instruction**
    - The identified location must successfully decode to a `call` instruction. Dispatchers are always invoked via a `call` instruction. Additionally, dispatchers utilize the return address from the call site to locate their encoded 32-bit displacement.
2. **Absence of Subsequent `call` Instructions**
    - Dispatchers must not contain any `call` instructions within their disassembly listing. The presence of any `call` instructions within a presumed dispatcher range immediately disqualifies the call location as a dispatcher candidate.
3. **Absence of Privileged Instructions and Indirect Control Transfers**

    - Similarly to `call` instructions, the dispatcher cannot include privileged instructions or indirect unconditional `jmps`. Any presence of any such instructions invalidates the call location.

4. **Detection of `pushfq` and `popfq` Guard Sequences**

    - The dispatcher must contain `pushfq` and `popfq` instructions to safeguard the `RFLAGS` register during decoding. These sequences are unique to dispatchers and suffice for a generic identification without worrying about the differences that arise between how the decoding takes place.

Figure 23 is the fingerprint verification routine that incorporates all the aforementioned characteristics and validation checks given a potential call location:

```
#--------------------------------------------------------------------------
def _verify_dispatcher_pushfq(
  d: ProtectedInput64,
  call_offset: int
) -> bool:
  """Verifies instruction dispatchers by identifying their pushfq-popfq
  instruction sequences.

  This implements a raw disassembly pass that detects the guaranteed encoded
  offset decoding and ignoring any instruction-dispatcher-specific mutations
  that exist. The mutations can and do differ between samples and are prolly
  randomly selected form a set that obfuscator has.

  There can be no call instructions within a dispatcher.
  """
  #------------------------------------------------------------------------
  call_instr = d.md.decode(call_offset)
  if not call_instr.is_call(): return False
  target_ea: int = call_instr.get_op1_imm()
  if target_ea not in range(0, len(d.imgbuffer)): return False
  #------------------------------------------------------------------------
  MAX_SCAN_RANGE  = 15 # arbitrary number, but doesn't need to be large
  pushfq_hit      = False
  curr_ea         = target_ea
  count: int      = 0
  while count < MAX_SCAN_RANGE:
    try:
      instr: x86Instr = d.md.decode(curr_ea)
    except Exception as _:
      return False # if we throw then we know it's an invalid site
    #----------------------------------------------------------------------
    if pushfq_hit and instr.is_popfq(): # successful exit condition
      return True
    #----------------------------------------------------------------------
    if instr.is_pushfq():
      pushfq_hit = True
      count = 0
      curr_ea = instr.ea + instr.size
      continue
    if (
      x86.X86_GRP_CALL in instr.groups or
      x86.X86_GRP_PRIVILEGE in instr.groups or
      instr.is_jmp() and (instr.is_op1_reg or instr.is_op1_mem)
    ):
      return False
    #----------------------------------------------------------------------
    if instr.is_jcc() or instr.is_jmp():
      curr_ea = instr.get_op1_imm()
    else:
      curr_ea = instr.ea + instr.size
    count += 1
    #----------------------------------------------------------------------
```

```
    return False
```

Figure 23: The dispatch fingerprint routine

Emulating Dispatchers to Resolve Destination Targets

After filtering potential dispatchers using the fingerprinting routine, the next step is to emulate them in order to recover their destination targets.

```
#------------------------------------------------------------------------------
emu = EmulateIntel64()
emu.map_image(bytes(d.imgbuffer))
emu.map_teb()
snapshot = emu.context_save()
#------------------------------------------------------------------------------
LOG_INSTR: bool = False
MAX_DISPATCHER_RANGE = 45 # dummy range, more than enough for any dispatcher
#------------------------------------------------------------------------------
for call_dispatch_ea in potential_dispatchers:
  emu.context_restore(snapshot)
  emu.pc = call_dispatch_ea
  try:
    for _ in range(MAX_DISPATCHER_RANGE):
      emu.stepi()
      instr = next(emu.dis.disasm(emu.mem[emu.pc:emu.pc+15], emu.pc))
        if x86.X86_GRP_RET in instr.groups:
          next_pc = emu.parse_u64(emu.rsp)
    #------------------------------------------------------
    d.dispatchers_to_target[call_dispatch_ea] = next_pc
    d.dispatcher_locs.append(call_dispatch_ea)
    break
  except Exception:
    continue
```

Figure 24: Emulation sequence used to recover dispatcher destination targets

The Python code in Figure 24 performs this logic and operates as follows:

- **Initialization of the Emulator**
    - Creates the core engine for simulating execution (`EmulateIntel64`), maps the protected binary image (`imgbuffer`) into the emulator's memory space, maps the Thread Environment Block (TEB) as well to simulate a realistic Windows execution environment, and creates an initial snapshot to facilitate fast resets before each emulation run without needing to reinitialize the entire emulator each time.
    - `MAX_DISPATCHER_RANGE` specifies the maximum number of instructions to emulate for each dispatcher. The value 45 is chosen arbitrarily, sufficient given the limited instruction count in dispatchers even with the added mutations.

  - A `try` / `except` block is used to handle any exceptions during emulation. It is assumed that exceptions result from false positives among the potential dispatchers identified earlier and can be safely ignored.
- **Emulating Each Potential Dispatcher**
  - For each potential dispatcher address (`call_dispatch_ea`), the emulator's context is restored to the initial snapshot. The program counter (`emu.pc`) is set to the address of each dispatcher. `emu.stepi()` executes one instruction at the current program counter, after which the instruction is analyzed to determine whether we have finished.
    - If the instruction is a `ret` , the emulation has reached the dispatch point.
    - The dispatch target address is read from the stack using `emu.parse_u64(emu.rsp)` .
  - The results are captured by `d.dispatchers_to_target` , which maps the dispatcher address to the dispatch target. The dispatcher address is additionally stored in the `d.dispatcher_locs` lookup cache.
    - The `break` statement exits the inner loop, proceeding to the next dispatcher.

Patching and Linearization

After collecting and verifying every captured instruction dispatcher, the final step is to replace each call location with a direct branch to its respective destination target. Since both near `call` and `jmp` instructions occupy 5 bytes in size, this replacement can be seamlessly performed by merely patching the `jmp` instruction over the `call` .

```
#-------------------------------------------------------------------------------
# Keep a copy of the protected image with the patches
d.jmppatchedbuffer = d.imgbuffer
def _apply_call_to_jmp_patch(call_ea: int, target_ea: int):
  JMP_SIZE = 5
  rel_offset = (target_ea - (call_ea + JMP_SIZE)) & 0xFFFFFFFF
  rel_offset_slice = rel_offset.to_bytes(4, byteorder='little')
  jmp_instr = b'\xE9' + rel_offset_slice
  d.jmppatchedbuffer[call_ea:call_ea+JMP_SIZE] = jmp_instr
#-------------------------------------------------------------------------------
for call_dispatch_ea, dispatch_target_ea in d.dispatchers_to_target.items():
  _apply_call_to_jmp_patch(call_dispatch_ea, dispatch_target_ea)
#-------------------------------------------------------------------------------
```

Figure 25: Patching sequence to transform instruction dispatcher calls to unconditional jmps to their destination targets

We utilize the `dispatchers_to_target` map, established in the previous section, which associates each dispatcher call location with its corresponding destination target. By iterating through this map, we identify each dispatcher call location and replace the original `call` instruction with a `jmp` . This substitution redirects the execution flow directly to the intended target addresses.

This removal is pivotal to our deobfuscation strategy as it removes the intended dynamic dispatch element that instruction dispatchers were designed to provide. Although the code is still scattered throughout the code segment,

the execution flow is now statically deterministic, making it immediately apparent which instruction leads to the next one.

When we compare these results to the initial screenshot from the Instruction Dispatcher section, the blocks still appear scattered. However, their execution flow has been linearized. This progress allows us to move forward to the second phase of our CFG recovery.

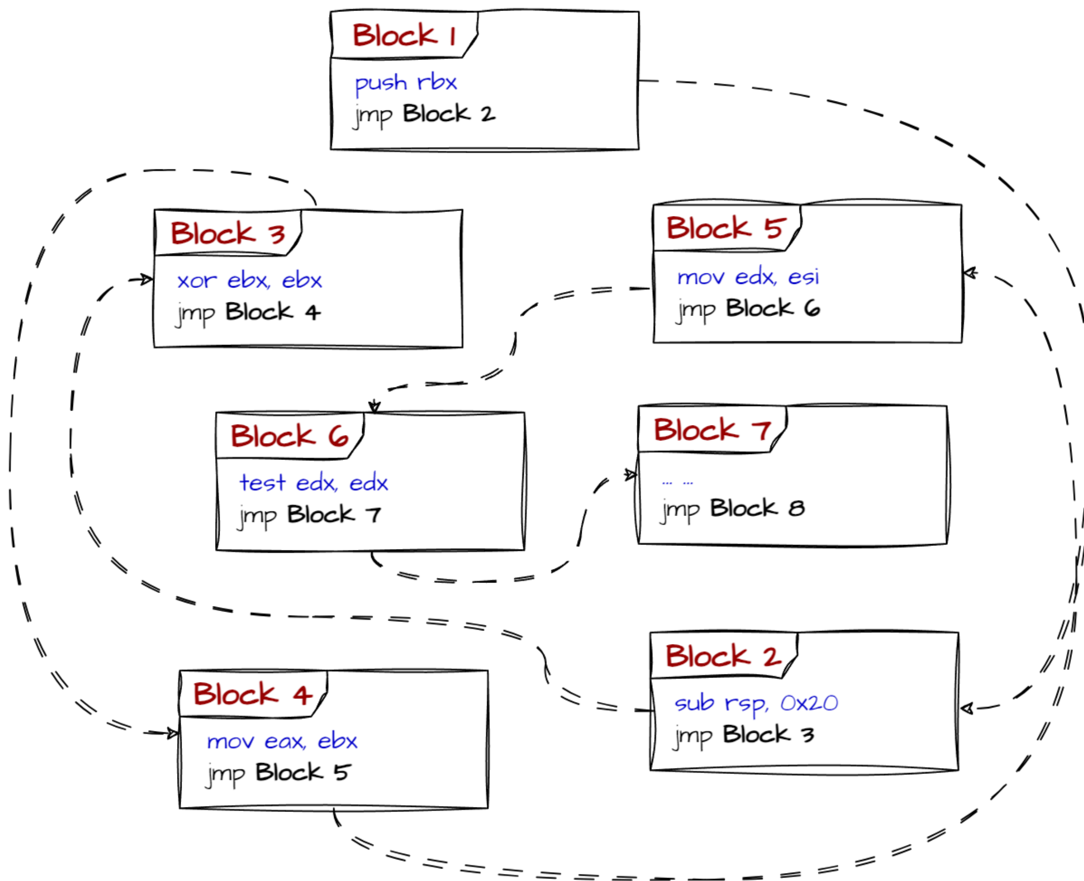


Figure 26: Linearized instruction dispatcher control flow

Function Identification and Recovery

By eliminating the effects of instruction dispatchers, we have linearized the execution flow. The next step involves assimilating the dispersed code and leveraging the linearized control flow to reconstruct the original functions that comprised the unprotected binary. This recovery phase involves several stages, including raw instruction recovery, normalization, and the construction of the final CFG.

Function identification and recovery is encapsulated in the following two abstractions:

- **Recovered instruction** (`RecoveredInstr`): The fundamental unit for representing individual instructions recovered from an obfuscated binary. Each instance encapsulates not only the raw instruction data but also metadata essential for relocation, normalization, and analysis within the CFG recovery process.

- **Recovered function** (`RecoveredFunc`): The end result of successfully recovering an individual function from an obfuscated binary. It aggregates multiple `RecoveredInstr` instances, representing the sequence of instructions that constitute the unprotected function. The complete CFG recovery process results in an array of `RecoveredFunc` instances, each corresponding to a distinct function within the binary. We will utilize these results in the final Building Relocations in Deobfuscated Binaries section to produce fully deobfuscated binaries.

We do not utilize a basic block abstraction for our recovery approach given the following reasons. Properly abstracting basic blocks presupposes complete CFG recovery, which introduces unnecessary complexity and overhead for our purposes. Instead, it is simpler and more efficient to conceptualize a function as an aggregation of individual instructions rather than a collection of basic blocks in this particular deobfuscation context.

```
@dataclass
class RecoveredInstr:
    """
    The basic primitive for recovery. It represents a recovered instruction from
    the original, protected binary. It will always be associated with a/its
    respective recovered function.

    :func_start_ea:   ref to the parent function the instr is associated with
    :instr:           original, recovered instruction at its original location
    :reloc_instr:     original instruction, at its new, relocated address
    :reloc_ea:        the relocated ea for the instruction (in new image)
    :linear_ea:       the linear ea, after a CFG lineraization (normalization) pass
    :is_obf_import:   ids whether the instr is an obf call/jmp import instr
    :is_boundary_jmp: ids synthetic jmps added to the cfg recovery. These
                      need to be distinguished when relocating the instruction
    :updated_bytes:   the underlying operands that represent an instr can/will
                      be modifed during relocation and certain parts of the
                      deobfuscation.. Keep original and updated bytes separate
    """
    func_start_ea:   int            = -1
    type:            int|None       = None
    instr:           x86Instr|None = None
    reloc_instr:     x86Instr|None = None
    reloc_ea:        int            = 0xffffffff
    linear_ea:       int            = 0xffffffff
    is_obf_import:   bool           = False
    is_boundary_jmp: bool           = False
    updated_bytes:   bytearray      = field(default_factory=bytearray)
```

Figure 27: RecoveredInstr type definition

```python
@dataclass
class RecoveredFunc:
    """
    The second essential primitive for recovery. It represents the recovered
    function from the original, protected binary and constitutes an
    aggregate of recovered instructions (RecoveredInstr). Note, in this case
    it is explicitly an aggregate of instructions, not basic blocks. Extra
    passes can be added to include bb abstractions, if necessary.

    :func_start_ea:    The original starting address of the function.
    :data_section_off: Offset to `.data` section. Keeping it as part of
                       the function allows for easily distinguishing data
                       section cross-references.
    :recovered:        List of recovered instructions.
    :normalized_flow:  List of instructions in normalized flow.
    :ea_to_recovered:  Map of original addresses to their recovered links.
    :obf_backbone:     Backbone links particular to this function.
    :lea_refs:         References collected during LEA instructions processing.
    :sub_calls:        List of instructions that are subroutine calls.
    :reloc_ea:         Relocated function start address.
    :relocs_imports:   Relocation info for imports.
    :relocs_ctrlflow:  Relocation info for control flow instructions.
    :relocs_dataflow:  Relocation info for static data references.
    """

    func_start_ea:    int
    data_section_off: int

    recovered:       list[RecoveredInstr]     = field(default_factory=list)
    normalized_flow: list[RecoveredInstr]     = field(default_factory=list)
    ea_to_recovered: dict[int,RecoveredInstr] = field(default_factory=dict)
    obf_backbone:    dict[int,int]            = field(default_factory=dict)

    sub_calls:       list[x86Instr]           = field(init=False)
    lea_refs:        list[int]                = field(default_factory=list)

    reloc_ea: int = 0xFFFFFFFF
    relocs_imports:  list[RecoveredInstr]     = field(default_factory=list)
    relocs_ctrlflow: list[RecoveredInstr]     = field(default_factory=list)
    relocs_dataflow: list[RecoveredInstr]     = field(default_factory=list)

    def __post_init__(self):
        self.size = sum(len(r.instr.bytes) for r in self.recovered)
        self.sub_calls = [r.instr for r in self.recovered if r.instr.is_call()]
```

Figure 28: RecoveredFunc type definition

DFS Rule-Guided Stepping Introduction

We opted for a recursive-depth algorithm given the following reasons:

- **Natural fit for code traversal:** DFS allows us to infer function boundaries based solely on the flow of execution. It mirrors the way functions call other functions, making it intuitive to implement and reason

about when reconstructing function boundaries. It also simplifies following the flow of loops and conditional branches.

- **Guaranteed execution paths:** We concentrate on code that is definitely executed. Given we have at least one known entry point into the obfuscated code, we know execution must pass through it in order to reach other parts of the code. While other parts of the code may be more indirectly invoked, this entry point serves as a foundational starting point.
    - By recursively exploring from this known entry, we will almost certainly encounter and identify virtually all code paths and functions during our traversal.
- **Adapts to instruction mutations:** We tailor the logic of the traversal with callbacks or "rules" that stipulate how we process each individual instruction. This helps us account for known instruction mutations and aids in stripping away the obfuscator's code.

The core data structures involved in this process are the following: `CFGResult`, `CFGStepState`, and `RuleHandler`:

- `CFGResult`: Container for the results of the CFG recovery process. It aggregates all pertinent information required to represent the CFG of a function within the binary, which it primarily consumes from `CFGStepState`.
- `CFGStepState`: Maintains the state throughout the CFG recovery process, particularly during the controlled-step traversal. It encapsulates all necessary information to manage the traversal state, track progress, and store intermediate results.
    - **Recovered cache:** Stores instructions that have been recovered for a protected function without any additional cleanup or verification. This initial collection is essential for preserving the raw state of the instructions as they exist within the obfuscated binary before any normalization or validation processes are applied after. It is always the first pass of recovery.
    - **Normalized cache:** The final pass in the CFG recovery process. It transforms the raw instructions stored in the recovered cache into a fully normalized CFG by removing all obfuscator-introduced instructions and ensuring the creation of valid, coherent functions.
    - **Exploration stack:** Manages the set of instruction addresses that are pending exploration during the DFS traversal for a protected function. It determines the order in which instructions are processed and utilizes a `visited` set to ensure that each instruction is processed only once.
    - **Obfuscator backbone:** A mapping to preserve essential control flow links introduced by the obfuscator
- `RuleHandler`: Mutation rules are merely callbacks that adhere to a specific function signature and are invoked during each instruction step of the CFG recovery process. They take as input the current protected binary, `CFGStepState`, and the current step-in instruction. Each rule contains specific logic designed to detect particular types of instruction characteristics introduced by the obfuscator. Based on the detection of these characteristics, the rules determine how the traversal should proceed. For instance, a rule might decide to continue traversal, skip certain instructions, or halt the process based on the nature of the mutation.

```
"""
  :func_ea:  function start address the CFG represents
  :recovered_instrs:  all recovered instructions (raw recovery)
  :normalized_flow:   normalized recovery (work with this)
  :ea_to_recovered:   lookup table - instr ea -> recovered instr
  :obf_backbone:      complete backbone links specific to this CFG
"""
CFGResult = collections.namedtuple(  # cannot unpack data class
    "CFGResult", [
        "func_ea",          # int
        "recovered_instrs", # list[RecoveredInstr]
        "normalized_flow",  # list[RecoveredInstr]
        "ea_to_recovered",  # dict[int,RecoveredInstr]
        "obf_backbone"      # dict[int,int]
    ]
)
```

Figure 29: CFGResult type definition

```
@dataclass
class CFGStepState:
  """Represents the recovery state for a controlled-step-CFG traversal.

  :func_start_ea:   the start address of the function to recover
  :rules:           list of rules that dictate how step works
  :recovered:       all recovered, original instructions inside the function boundary
  :ea_to_recovered: easy access map from an ea to its recovered function
  :obf_backbone:    map between an obfuscator's instr ea to its destination target
  :normalized_flow: reconstructed linear "normalized" execution flow of the recovery
  :ea_to_linear:    lookup utilized in linear flow recovery for quick refs to parsed instrs
  :to_explore:      DFS stack
  :visited:         visited locations
  """
  func_start_ea: int

  log: bool = True

  recovered:       List[RecoveredInstr]     = field(default_factory=list)
  ea_to_recovered: Dict[int,RecoveredInstr] = field(default_factory=dict)
  obf_backbone:    Dict[int,int]            = field(default_factory=dict)

  normalized_flow: List[RecoveredInstr]     = field(default_factory=list)
  ea_to_linear:    Dict[int,RecoveredInstr] = field(default_factory=dict)

  to_explore:      List[int]                = field(init=False)
  visited:         Set[int]                 = field(default_factory=set)

  prev_instr: Optional[x86Instr] = None   # Debug helper

  def __post_init__(self):
    self.to_explore = [self.func_start_ea]
```

Figure 30: CFGStepState type definition

```
RuleHandler: TypeAlias = Callable[
  ["ProtectedInput64", "CFGStepState", "x86Instr"], RuleResult
]
```

Figure 31: RuleHandler type definition

The following figure is an example of a rule that is used to detect the patched instruction dispatchers we introduced in the previous section and differentiating them from standard `jmp` instructions:

```
def RULE_HANDLE_DISPATCHER_JMP_AND_STANDARD_JMPS(
  d: ProtectedInput64,
  s: 'CFGStepState',
  instr: x86Instr
) -> RuleResult:
  """Rule that identiifes patched instruction dispatchers and
  differentiates them from standard jmp instructions.
  """
  if instr.is_jmp() and instr.ea in d.dispatcher_locs:
    jmp_dest = instr.get_op1_imm()
    s.obf_backbone[instr.ea] = jmp_dest
    s.to_explore.append(jmp_dest)
    return RuleResult.CONTINUE
  elif instr.is_jmp():
    rinstr = RecoveredInstr(func_start_ea=s.func_start_ea, instr=instr)
    s.recovered.append(rinstr)
    s.ea_to_recovered[instr.ea] = rinstr
    #-----------------------------------------------------------------
    if instr.is_op1_reg or instr.is_op1_mem:
      return RuleResult.CONTINUE
    jmp_dest = instr.get_op1_imm()
    s.to_explore.append(jmp_dest)
    return RuleResult.CONTINUE
  return RuleResult.NEXT_RULE
```

Figure 32: RuleHandler example that identifies patched instruction dispatchers and differentiates them from standard jmp instructions

DFS Rule-Guided Stepping Implementation

The remaining component is a routine that orchestrates the CFG recovery process for a given function address within the protected binary. It leverages the `CFGStepState` to manage the DFS traversal and applies mutation rules to decode and recover instructions systematically. The result will be an aggregate of `RecoveredInstr` instances that constitute the first pass of raw recovery:

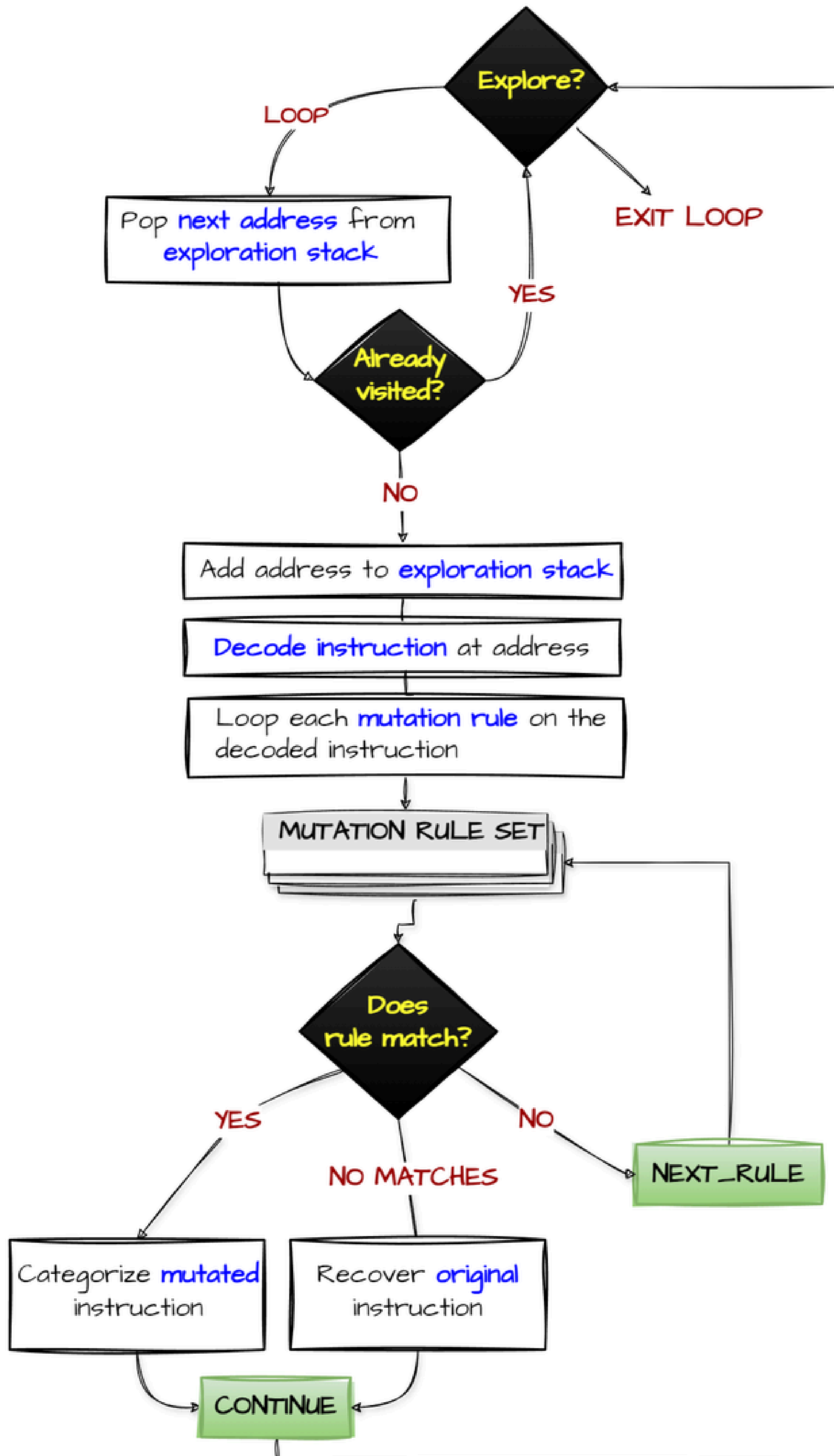

Explore?
LOOP
EXIT LOOP
Pop next address from exploration stack
YES
Already visited?
NO
Add address to exploration stack
Decode instruction at address
Loop each mutation rule on the decoded instruction
MUTATION RULE SET
Does rule match?
YES
NO
NO MATCHES
NEXT_RULE
Categorize mutated instruction
Recover original instruction
CONTINUE

Figure 33: Flow chart of our DFS rule-guided stepping algorithm

The following Python code directly implements the algorithm outlined in Figure 33. It initializes the CFG stepping state and commences a DFS traversal starting from the function's entry address. During each step of the traversal, the current instruction address is retrieved from the `to_explore` exploration stack and checked against the `visited` set to prevent redundant processing. The instruction at the current address is then decoded, and a series of mutation rules are applied to handle any obfuscator-induced instruction modifications. Based on the outcomes of these rules, the traversal may continue, skip certain instructions, or halt entirely.

Recovered instructions are appended to the `recovered` cache, and their corresponding mappings are updated within the `CFGStepState`. The `to_explore` stack is subsequently updated with the address of the next sequential instruction to ensure systematic traversal. This iterative process continues until all relevant instructions have been explored, culminating in a `CFGResult` that encapsulates the fully recovered CFG.

```
def recover_cfg_step(
  d: ProtectedInput64,
  func_start_ea: int
) -> CFGResult:
  """
  Given a start address of a presumed function, recover its control flow using
  the set of mutation rules specified in ProtectedInput64
  """
  #-----------------------------------------------------------------------------
  from enum import Enum
  class StepResult(Enum):
    CONTINUE = 0
    STOP = 1
  #-----------------------------------------------------------------------------
  def _step(d: ProtectedInput64, s: CFGStepState) -> StepResult:
    if not s.to_explore: return False
    #---------------------------------------------------------------------------
    curr_ea: int = s.to_explore.pop()
    if curr_ea in s.visited: return True # Continue
    s.visited.add(curr_ea)
    #---------------------------------------------------------------------------
    try:                              # exceptions can throw on decoding
      s.prev_instr = instr = d.mdp.decode(curr_ea)
    except Exception as e:            # error handling removed for simplicity
        raise ValueError(es)
    #---------------------------------------------------------------------------
    for rule in d.mutation_rules:     # list of rules specified by user
        match rule(d, s, instr):
            case RuleResult.NEXT_RULE: continue
            case RuleResult.CONTINUE:  return StepResult.CONTINUE
            case RuleResult.BREAK:     return StepResult.STOP
    s.recovered.append(
      RecoveredInstr(func_start_ea=s.func_start_ea,
                     instr=instr)
    )
    s.ea_to_recovered[instr.ea] = s.recovered[-1]
    s.to_explore.append(instr.ea + instr.size)
    return StepResult.CONTINUE
  #-----------------------------------------------------------------------------
  # build the stepping state and commence raw recovery
  s: CFGStepState = CFGStepState(func_start_ea=func_start_ea)
  while s.to_explore:
    if not _step(d,s):
      break
  #-----------------------------------------------------------------------------
  # @NOTE: THIS IS THE FINAL PASS AND CAN BE IGNORED
  normalize_raw_recovery(d, s)
  #-----------------------------------------------------------------------------
  return CFGResult(
    func_ea=s.func_start_ea,
    recovered_instrs=s.recovered,
    normalized_flow=s.normalized_flow,
    ea_to_recovered=s.ea_to_recovered,
    obf_backbone=s.obf_backbone
  )
```

Figure 34: DFS rule-guided stepping algorithm Python implementation

Normalizing the Flow

With the raw instructions successfully recovered, the next step is to normalize the control flow. While the raw recovery process ensures that all original instructions are captured, these instructions alone do not form a cohesive and orderly function. To achieve a streamlined control flow, we must filter and refine the recovered instructions—a process we refer to as normalization. This stage involves several key tasks:

- **Updating branch targets:** Once all of the obfuscator-introduced code (instruction dispatchers and mutations) are fully removed, all branch instructions must be redirected to their correct destinations. The scattering effect introduced by obfuscation often leaves branches pointing to unrelated code segments.
- **Merging overlapping basic blocks:** Contrary to the idea of a basic block as a strictly single-entry, single-exit structure, compilers can produce code in which one basic block begins within another. This overlapping of basic blocks commonly appears in loop structures. As a result, these overlaps must be resolved to ensure a coherent CFG.
- **Proper function boundary instruction:** Each function must begin and end at well-defined boundaries within the binary's memory space. Correctly identifying and enforcing these boundaries is essential for accurate CFG representation and subsequent analysis.

Simplifying with Synthetic Boundary Jumps

Rather than relying on traditional basic block abstractions—which can impose unnecessary overhead—we employ synthetic boundary jumps to simplify CFG normalization. These artificial `jmp` instructions link otherwise disjointed instructions, allowing us to avoid splitting overlapping blocks and ensuring that each function concludes at a proper boundary instruction. This approach also streamlines our binary rewriting process when reconstructing the recovered functions into the final deobfuscated output binary.

Merging overlapping basic blocks and ensuring functions have proper boundary instructions amount to the same problem—determining which scattered instructions should be linked together. To illustrate this, we will examine how synthetic jumps effectively resolve this issue by ensuring that functions conclude with the correct boundary instructions. The exact same approach applies to merging basic blocks together.

Synthetic Boundary Jumps to Ensure Function Boundaries

Consider an example where we have successfully recovered a function using our DFS-based rule-guided approach. Inspecting the recovered instructions in the `CFGState` reveals a `mov` instruction as the final operation. If we were to reconstruct this function in memory as-is, the absence of a subsequent fallthrough instruction would compromise the function's logic.

```
cfg_state.recovered
Out[5]:
[0x003468 (48895c2408)        mov qword ptr [rsp + 8], rbx,
 0x00346d (57)                push rdi,
 0x00f920 (4883ec20)          sub rsp, 0x20,
 0x00f924 (488b0d80700000)    mov rcx, qword ptr [rip + 0x7080],
 0x00f937 (4885c9)            test rcx, rcx,
 0x00f93a (740b)              je 0xf947,
 0x00f942 (e80235ffff)        call 0x2e49,
 0x00f947 (488b3db76f0000)    mov rdi, qword ptr [rip + 0x6fb7],
 0x015192 (4885ff)            test rdi, rdi,
 0x00ed32 (0f854f83ffff)      jne 0x7087,
 0x00ba7f (8d4f50)            lea ecx, [rdi + 0x50],
 0x00ba82 (e88fd5ffff)        call 0x9016,
 0x00ba94 (4885c0)            test rax, rax,
 0x00ba97 (0f844e9dffff)      je 0x57eb,
 0x00d1bb (8d570a)            lea edx, [rdi + 0xa],
 0x00d1be (488bc8)            mov rcx, rax,
 0x00d1c1 (e809380000)        call 0x109cf,
 0x0057dc (488bf8)            mov rdi, rax,
 0x0057df (4889051f110100)    mov qword ptr [rip + 0x1111f], rax,
 0x0057e6 (e912d8ffff)        jmp 0x2ffd,
 0x00300a (4885ff)            test rdi, rdi,
 0x00300d (0f848d1a0000)      je 0x4aa0,
 0x007087 (488b5f48)          mov rbx, qword ptr [rdi + 0x48],
 0x00b51d (4883674800)        and qword ptr [rdi + 0x48], 0,
 0x00b52b (83caff)            or edx, 0xffffffff,
 0x00b52e (488bcb)            mov rcx, rbx,
 0x00b531 (ff1594b40000)      call qword ptr [rip + 0xb494],
 0x00b537 (488bcb)            mov rcx, rbx,
 0x00b53a (ff15f9b70000)      call qword ptr [rip + 0xb7f9],
 0x011411 (488b4f30)          mov rcx, qword ptr [rdi + 0x30],
 0x002996 (e8ae040000)        call 0x2e49,
 0x0029a5 (4883673000)        and qword ptr [rdi + 0x30], 0,
 0x004a8f (488bcf)            mov rcx, rdi,
 0x004a92 (ff1555180100)      call qword ptr [rip + 0x11855],
 0x004a98 (488bcf)            mov rcx, rdi,
 0x004a9b (e8a9e3ffff)        call 0x2e49,
 0x00518c (ff15df100100)      call qword ptr [rip + 0x110df],
 0x00101b (488b5c2430)        mov rbx, qword ptr [rsp + 0x30],
 0x001029 (33c0)              xor eax, eax,
 0x00e721 (4883c420)          add rsp, 0x20,
 0x00a2ed (5f)                pop rdi,
 0x00a2ee (c3)                ret ,
 0x002ff4 (33ff)              xor edi, edi,
 0x002ff6 (48893d08390100)    mov qword ptr [rip + 0x13908], rdi]
```

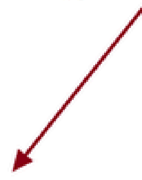


Figure 35: Example of a raw recovery that does not end with a natural function boundary instruction

To address this, we introduce a synthetic jump whenever the last recovered instruction is not a natural function boundary (e.g., `ret`, `jmp`, `int3`).

```
#----------------------------------------------------------------------
def is_boundary_instr(r: RecoveredInstr):
    return (
        r.instr.is_ret() or r.instr.is_jmp() or r.instr.is_int3()
    )
```

Figure 36: Simple Python routine that identifies function boundary instructions

We determine the fallthrough address, and if it points to an obfuscator-introduced instruction, we continue forward until reaching the first regular instruction. We call this traversal "walking the obfuscator's backbone":

```
def walk_backbone(instr_ea: int):
  """
  `obf_backbone` contains all identified instructions added by the obfuscator
  `dispatcher_locs` and `dispatchers_to_target` identify all instruction
  dispatchers within the protected binary.
  """
  curr_ea = instr_ea
  while curr_ea in s.obf_backbone or curr_ea in d.dispatcher_locs:
    curr_ea = (
      d.dispatchers_to_target[curr_ea] if curr_ea in d.dispatcher_locs
      else s.obf_backbone[curr_ea]
  )
  return curr_ea
```

Figure 37: Python routine that implements walking the obfuscator's backbone logic

We then link these points with a synthetic jump. The synthetic jump inherits the original address as metadata, effectively indicating which instruction it is logically connected to.

```
... snip snip ...
0x0057e6 (e912d8ffff)           jmp 0x2ffd,
0x00300a (4885ff)               test rdi, rdi,
0x00300d (0f848d1a0000)         je 0x4aa0,
0x007087 (488b5f48)             mov rbx, qword ptr [rdi + 0x48],
0x00b51d (4883674800)           and qword ptr [rdi + 0x48], 0,
0x00b52b (83caff)               or edx, 0xffffffff,
0x00b52e (488bcb)               mov rcx, rbx,
0x00b531 (ff1594b40000)         call qword ptr [rip + 0xb494],
0x00b537 (488bcb)               mov rcx, rbx,
0x00b53a (ff15f9b70000)         call qword ptr [rip + 0xb7f9],
0x011411 (488b4f30)             mov rcx, qword ptr [rdi + 0x30],
0x002996 (e8ae040000)           call 0x2e49,
0x0029a5 (4883673000)           and qword ptr [rdi + 0x30], 0,
0x004a8f (488bcf)               mov rcx, rdi,
0x004a92 (ff1555180100)         call qword ptr [rip + 0x11855],
0x004a98 (488bcf)               mov rcx, rdi,
0x004a9b (e8a9e3ffff)           call 0x2e49,
0x00518c (ff15df100100)         call qword ptr [rip + 0x110df],
0x00101b (488b5c2430)           mov rbx, qword ptr [rsp + 0x30],
0x001029 (33c0)                 xor eax, eax,
0x00e721 (4883c420)             add rsp, 0x20,
0x00a2ed (5f)                   pop rdi,
0x00a2ee (c3)                   ret ,
0x002ff4 (33ff)                 xor edi, edi,
0x002ff6 (48893d08390100)       mov qword ptr [rip + 0x13908], rdi],
0x002ff6 (eb12)                 jmp 0x300a]
```

Figure 38: Example of adding a synthetic boundary jmp to create a natural function boundary

Updating Branch Targets

After normalizing the control flow, adjusting branch targets becomes a straightforward process. Each branch instruction in the recovered code may still point to obfuscator-introduced instructions rather than the intended destinations. By iterating through the `normalized_flow` cache (generated in the next section), we identify branching instructions and verify their targets using the `walk_backbone` routine.

This ensures that all branch targets are redirected away from the obfuscator's artifacts and correctly aligned with the intended execution paths. Notice we can ignore `call` instructions given that any non-dispatcher `call` instruction is guaranteed to always be legitimate and never part of the obfuscator's protection. These will, however, need to be updated during the final relocation phase outlined in the Building Relocations in Deobfuscated Binaries section.

Once recalculated, we reassemble and decode the instructions with updated displacements, preserving both correctness and consistency.

```python
#-----------------------------------------------------------------------
def update_branch_targets():
  r: RecoveredInstr
  for r in s.normalized_flow:
    if r.instr.is_jcc() or (r.instr.is_jmp() and r.instr.is_op1_imm):
      # targets can still point to backbone
      branch_dest = walk_backbone(r.instr.Op1.imm)
      branch_str  = f'{r.instr.mnemonic} {branch_dest:#08x}'
      nb = d.ks.asm(branch_str, addr=r.instr.ea, as_bytes=True)[0]
      r.instr = d.mdp.decode_buffer(nb, r.instr.ea)
```

Figure 39: Python routine responsible for updating all branch targets

Putting It All Together

Putting it all together, we developed the following algorithm that builds upon the previously recovered instructions, ensuring that each instruction, branch, and block is properly connected, resulting in a completely recovered and deobfuscated CFG for an entire protected binary. We utilize the recovered cache to construct a new, normalized cache. The algorithm employs the following steps:

1. **Iterate Over All Recovered Instructions**
    - Traverse all recovered instructions produced from our DFS-based stepping approach.
2. **Add Instruction to Normalized Cache**
    - For each instruction, add it to the normalized cache, which captures the results of the normalization pass.
3. **Identify Boundary Instructions**
    - Determine whether the current instruction is a boundary instruction.
        - **If it is a boundary instruction**, skip further processing of this instruction and continue to the next one (return to Step 1).
4. **Calculate Expected Fallthrough Instruction**
    - Determine the expected fallthrough instruction by identifying the sequential instruction that follows the current one in memory.
5. **Verify Fallthrough Instruction**
    - Compare the calculated fallthrough instruction with the next instruction in the recovered cache.
        - **If the fallthrough instruction is not the next sequential instruction in memory, check whether it's a recovered instruction we already normalized:**
            - If it is, add a synthetic jump to link the two together in the normalized cache.

        - If it is not, obtain the connecting fallthrough instruction from the recovery cache and append it to the normalized cache.
    - **If the fallthrough instruction matches the next instruction in the recovered cache**:
        - Do nothing, as the recovered instruction already correctly points to the fallthrough. Proceed to Step 6.

6. **Handle Final Instruction**
    - Check if the current instruction is the final instruction in the recovered cache.
        - **If it is the final instruction**:
            - Add a final synthetic boundary jump, because if we reach this stage, we failed the check in Step 3.
            - Continue iteration, which will cause the loop to exit.
        - **If it is not the final instruction**:
            - Continue iteration as normal (return to Step 1).

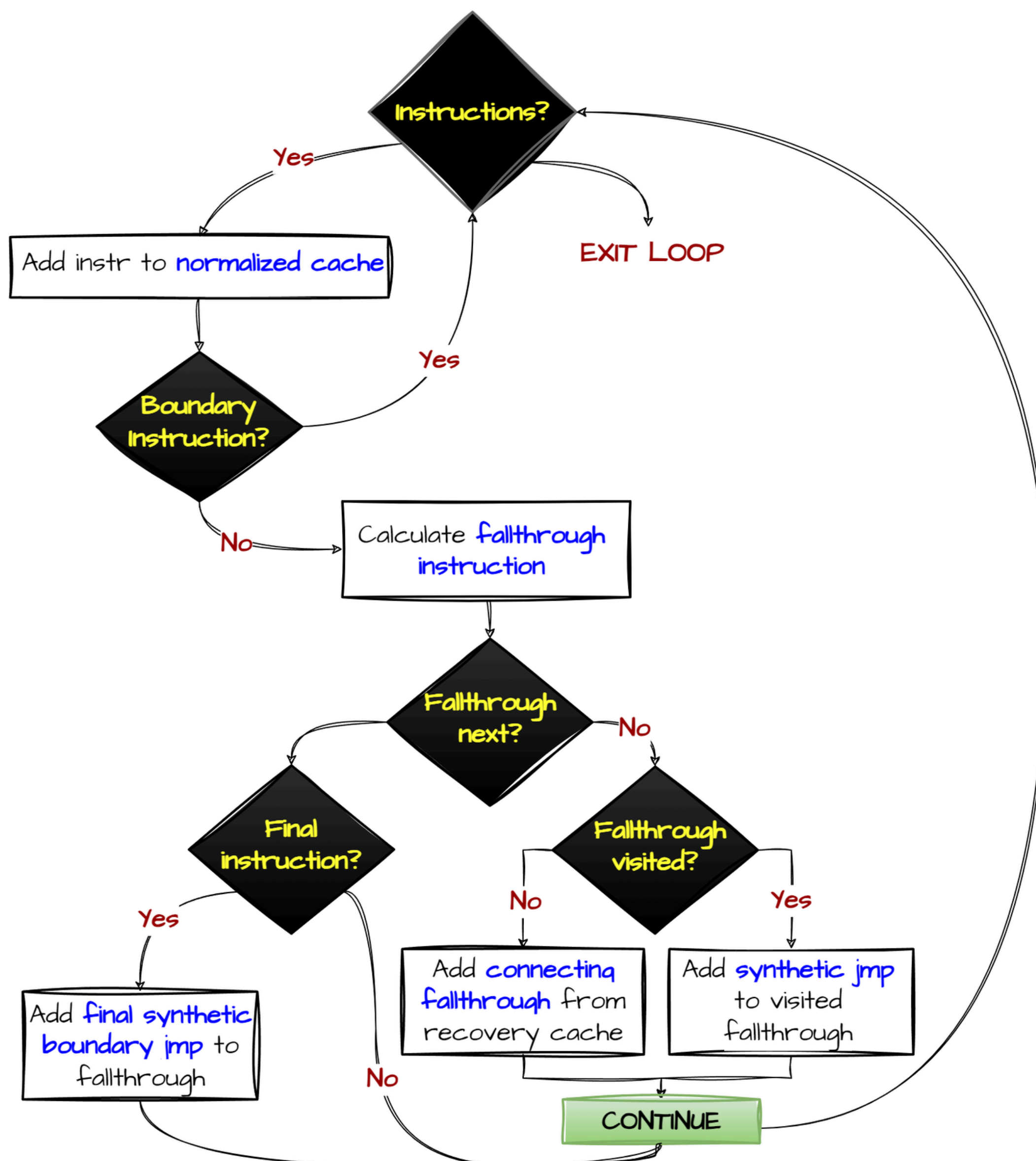


Figure 40: Flow chart of our normalization algorithm

The Python code in Figure 41 directly implements these normalization steps. It iterates over the recovered instructions and adds them to a normalized cache (`normalized_flow`), creates a linear mapping, and identifies where synthetic jumps are required. When a branch target points to obfuscator-injected code, it walks the backbone (`walk_backbone`) to find the next legitimate instruction. If the end of a function is reached without a natural boundary, a synthetic jump is created to maintain proper continuity. After the completion of the iteration, every branch target is updated (`update_branch_targets`), as illustrated in the previous section, to ensure that each instruction is correctly linked, resulting in a fully normalized CFG:

```
#---------------------------------------------------------------------------
# normalization pass pass
for i, r in enumerate(s.recovered):
  s.normalized_flow.append(r)
  s.ea_to_linear[r.instr.ea] = r
  #-----------------------------------------------------------------------
  if is_boundary_instr(r): continue
  #-----------------------------------------------------------------------
  fall_through_ea = walk_backbone(r.instr.ea + r.instr.size)
  if (
    i < len(s.recovered)-1 and
    s.recovered[i+1].instr.ea != fall_through_ea
  ):
    if fall_through_ea in s.ea_to_linear:
      # all synthetics have -1 as their address to signify they are not
      # part of the original binary
      sjr = f'jmp {fall_through_ea:#08x}'
      b = d.ks.asm(sjr, addr=r.instr.ea, as_bytes=True)[0]
      synthetic_jmp = d.mdp.decode_buffer(b, r.instr.ea)
      s.normalized_flow.append(
        RecoveredInstr(func_start_ea=r.func_start_ea,
                       instr=synthetic_jmp,
                       linear_ea=-1,
                       is_boundary_jmp=True))
    else:
      connected_instr = s.ea_to_recovered[fall_through_ea]
      s.normalized_flow.append(connected_instr)
      s.ea_to_linear[connected_instr.instr.ea] = connected_instr
  elif (len(s.recovered) - 1) == i:
      # last instruction is not a known function boundary if this path is
      # reached. Add a synthetic boundary jmp to merge the boundary.
      #-----------------------------------------------------------------
    bjs = f'jmp {fall_through_ea:#08x}'
    b = d.ks.asm(bjs, addr=r.instr.ea, as_bytes=True)[0]
    boundary_jmp = d.mdp.decode_buffer(b, r.instr.ea)
    s.normalized_flow.append(
      RecoveredInstr(func_start_ea=s.func_start_ea,
                     instr=boundary_jmp,
                     linear_ea=-1,
                     is_boundary_jmp=True))
  #-------------------------------------------------------------------------
  # can only be updated after the linear recovery is completed, as updating
  # before or alongside breaks how the fall through address is calculated
  update_branch_targets()
```

Figure 41: Python implementation of our normalization algorithm

Observing the Results

After applying our two primary passes, we have nearly eliminated all of the protection mechanisms. Although import protection remains to be addressed, our approach effectively transforms an incomprehensible mess into a perfectly recovered CFG.

For example, Figure 42 and Figure 43 illustrate the before and after of a critical function within the backdoor payload, which is a component of its plugin manager system. Through additional analysis of the output, we can identify functionalities that would have been impossible to delineate, much less in such detail, without our deobfuscation process.

```
0065EF jl      loc_34475
0065F5 mov     si, si
0065F8 jge     loc_34475
0065FE call    loc_2A5D
006603 movzx   ecx, di
006606 call    loc_3265
00660B mov     [rbx], rax
00660E call    near ptr sub_12B46
006613 scasb
006614 outsd
006615 push    rcx
006616 out     0E8h, eax
006618 test    ax, 0
00661C jnz     near ptr unk_574D1
006622 xor     edx, edx
006624 lea     rax, [rbp+1F8h]
00662B call    sub_533DC
006630 wait
006631 db      47h
006631 repne in eax, 0E8h
006635 call    loc_14E46
00663A call    near ptr sub_35D0C
00663F add     eax, 0E8EA5513h
006644
006644 loc_6644:                           ; CODE XREF: seg000:0000000000027151↓j
006644                                     ; seg000:00000000000A8DB↓j
006644 test    eax, 0
006649
006649 loc_6649:                           ; CODE XREF: seg000:000000000003EBD4↓j
006649 jnz     loc_4E44B
00664F
00664F loc_664F:                           ; CODE XREF: sub_65EE+70↓j
00664F call    near ptr sub_C086
006654 xchg    eax, edx
006654 ; ---------------------------------------------------------------------------
006655 db 0A0h
006656 db  15h
006657 db    7
006658 db 0E8h
006659 ; ---------------------------------------------------------------------------
006659 mov     rcx, rsi
006659 ; ---------------------------------------------------------------------------
00665C db 0FFh
00665D db  15h
00665E ; ---------------------------------------------------------------------------
00665E js      short near ptr loc_664F+3
006660 add     eax, 1C4CE800h
006665 add     al, 0
006667 adc     byte ptr [rsi+rdx], 20h ; ' '
00666B
00666B loc_666B:                           ; CODE XREF: seg000:loc_4CD2↑j
00666B call    near ptr 0FC0EBB8h
006670 test    [rcx], cl
006672 ja      short $+2
```

Figure 42: Original obfuscated shadow::PluginProtocolCreateAndConfigure routine

```
DWORD __fastcall shadow::PluginProtocolCreateAndConfigure(
        plugin_interface_t **Out_pNewInteface,
        unsigned __int16 PluginProtocolType,
        SOCKET ExistingSocket)
{
  plugin_interface_t *NewInteface; // rax MAPDST
  DWORD status; // edi
  mapped_plugin_node_t *plugin; // rax
  protocol_cbs_t *FunctionTable; // rax

  // build new plugin/protocol interface
  NewInteface = mem_alloc_0(sizeof(plugin_interface_t));
  if ( !NewInteface )
    return 8;
  NewInteface->MappedPluginImage = nullptr;
  NewInteface->ProtocolFunctionTable = nullptr;
  NewInteface->PluginSpecificDataType = 0LL;
  NewInteface->Type = 0;
  NewInteface->Type = PluginProtocolType;

  // ensure `PluginManager` is active
  shadow::PluginManagerGetOrCheckIfInit();

  // fetch the `mapped_plugin_img_t` via the protocol/plugin ID
  plugin = shadow::GetActivatePluginByProtocolType(PluginProtocolType);
  NewInteface->MappedPluginImage = plugin;
  if ( !plugin )
  {
    status = 0x7E;
__Check_Plugin_Maped_Image__:
    if ( NewInteface->MappedPluginImage )
    {
      shadow::PluginManagerGetOrCheckIfInit();
      shadow::PluginResetMemoryPermsOnDeactivation(NewInteface->MappedPluginImage);
    }
    win32::LocalFreeIfWrap(NewInteface);
    return status;
  }

  // invoke the initialization routine and configure the protocol (0 - success)
  FunctionTable = plugin->ProtocolFunctionTable;
  NewInteface->ProtocolFunctionTable = FunctionTable;
  status = (FunctionTable->ConfigureProtocol)(
             &NewInteface->PluginSpecificDataType,
             PluginProtocolType,
             ExistingSocket);
  if ( status )
    goto __Check_Plugin_Maped_Image__;
  *Out_pNewInteface = NewInteface;
  return 0;
}
```

Figure 43: Completely deobfuscated and functional shadow::PluginProtocolCreateAndConfigure routine

Import Recovery

Recovering and restoring the original import table revolves around identifying which import location is associated with which import dispatcher stub. From the stub dispatcher, we can parse the respective `obf_imp_t` reference in order to determine the protected import that it represents.

We pursue the following logic:

- **Identify each valid call/jmp location associated to an import**
  - The memory displacement for these will point to the respective dispatcher stub.
  - For HEADERLESS mode, we need to first resolve the fixup table to ensure the displacement points to a valid dispatcher stub.
- **For each valid location traverse the dispatcher stub to extract the obf_imp_t**
  - The `obf_imp_t` contains the RVAs to the encrypted DLL and API names.
- **Implement the string decryption logic**
  - We need to reimplement the decryption logic in order to recover the DLL and API names.
  - This was already done in the initial Import Protection section.

We encapsulate the recovery of imports with the following `RecoveredImport` data structure:

```
@dataclass
class RecoveredImport:
  """Encapsulates a recovered, protected import.

  :call_instr:       call/jmp site for the import `call qw ptr [rip+0x308f2]`
  :ref_instr:        `lea` fetch inside the stub that ref's the `obf_imp_t`
  :stub_id:          used to lookup into the import fixup table, when present
  :stub_ea:          starting address for the import's impstub dispatcher
  :stub_rfn          the impstub's recovered, deobfuscated CFG. Note, not
                     required for recovery, but keeping it as reference
  :dll_name:         decrypted dll name the import is a part of
  :api_name:         decrypted import name
  :new_rva:          the new, relocated RVA to the added import thunk
  :reloc_call_instr: the new relocated/updated call/jmp import call
  """
  call_instr:       x86Instr|None      = None
  ref_instr:        x86Instr|None      = None
  stub_id:          int                = 0xFFFFFFFF
  stub_ea:          int                = 0xFFFFFFFF
  stub_rfn:         RecoveredFunc|None = None
  dll_name:         str                = ''
  api_name:         str                = ''
  new_rva:          int|None           = -1
  reloc_call_instr: x86Instr|None      = None
```

Figure 44: RecoveredImport type definition

`RecoveredImport` serves as the result produced for each import that we recover. It contains all the relevant data that we will use to rebuild the original import table when producing the deobfuscated image.

Locate Protected Import CALL and JMP Sites

Each protected import location will be reflected as either an indirect near call (`FF/2`) or an indirect near jmp (`FF/4`):

```
FF15 <32-bit_displacement> call ImpStubDispatcher
FF25 <32-bit_displacement> jmp ImpStubDispatcher
```

Figure 45: Disassembly of import calls and jmps representation

Indirect near calls and jmps fall under the `FF` group opcode where the **Reg** field within the **ModR/M** byte identifies the specific operation for the group:

- `/2` : corresponds to `CALL r/m64`
- `/4` : corresponds to `JMP r/m64`

Taking an indirect near call as an example and breaking it down looks like the following:

1. `FF` : group opcode.
2. `15` : **ModR/M** byte specifying `CALL r/m64` with RIP-relative addressing.
    - `15` is encoded in binary as `00010101`
        - **Mod** (bits 6-7): `00`
            - Indicates either a direct RIP-relative displacement or memory addressing with no displacement.
        - **Reg** (bits 3-5): `010`
            - Identifies the call operation for the group
        - **R/M** (bits 0-2): `101`
            - In 64-bit mode with **Mod** `00` and **R/M** `101`, this indicates RIP-relative addressing.
3. `<32-bit displacement>` : added to `RIP` to compute the absolute address.

To find each protected import location and their associated dispatcher stubs we implement a trivial brute force scanner that locates all potential indirect near call/jmps via their first two opcodes.

```
#------------------------------------------------------------------
patterns = [bytes.fromhex('FF15'), bytes.fromhex('FF25')]
pattern_locs: list[int] = []
#------------------------------------------------------------------
# only scanning known code segment
max_length = min(len(d.imgbuffer), d.DATA_SECTION_EA)
for pattern in patterns:
    start_index = 0
    while start_index < max_length:
        curr_index = d.imgbuffer.find(pattern, start_index)
        if curr_index == -1 or curr_index >= max_length: break
        displ = struct.unpack(
            "<I",
            d.imgbuffer[curr_index+2:curr_index+6]
        )[0]
        if displ < len(d.imgbuffer): pattern_locs.append(curr_index)
        start_index = curr_index + len(pattern)
```

Figure 46: Brute-force scanner to locate all possible import locations

The provided code scans the code section of a protected binary to identify and record all locations with opcode patterns associated with indirect call and jmp instructions. This is the first step we take, upon which we apply additional verifications to guarantee it is a valid import site.

Resolving the Import Fixup Table

We have to resolve the fixup table when we recover imports for the HEADERLESS protection in order to identify which import location is associated with which dispatcher. The memory displacement at the protected import site will be paired with its resolved location inside the table. We use this displacement as a lookup into the table to find its resolved location.

Let's take a `jmp` instruction to a particular import as an example.

```
0045503 jmp     cs:qword_63A88
                        qword_63A88     dq 7FF6BE32A5E1h

; Import fixup entry
00669D8 dd 63A88h ; displacement RVA
00669DC dd 295E1h ; RVA to the import dispatcher for the displacement

$_ImpStub_for_63A88:
00295E1 push    rcx
00295E2 call    sub_508BB   ; instruction dispatcher
```

Figure 47: Example of a jmp import instruction including its entry in the import fixup table and the associated dispatcher stub

The `jmp` instruction's displacement references the memory location `0x63A88`, which points to garbage data. When we inspect the entry for this import in the fixup table using the memory displacement, we can identify the location of the dispatcher stub associated with this import at `0x295E1`. The loader will update the referenced data at `0x63A88` with `0x295E1`, so that when the `jmp` instruction is invoked, execution is appropriately redirected to the dispatcher stub.

Figure 48 is the deobfuscated code in the loader responsible for resolving the fixup table. We need to mimic this behavior in order to associate which import location targets which dispatcher.

```
$_Loop_Resolve_ImpFixupTbl:
mov     ecx, [rdx+4]            ; fixup , either DLL, API, or ImpStub
mov     eax, [rdx]              ; target ref loc that needs to be "fixed up"
inc     ebp                     ; update the counter
add     rcx, r13                ; calculate fixup fully (r13 is imgbase)
add     rdx, 8                  ; next pair entry
mov     [r13+rax+0], rcx        ; update the target ref loc w/ full fixup
movsxd  rax, dword ptr [rsi+18h] ; fetch imptbl total size, in bytes
shr     rax, 3                  ; account for size as a pair-entry
cmp     ebp, eax                ; check if done processing all entries
jl      $_Loop_Resolve_ImpTbl
```

Figure 48: Deobfuscated disassembly of the algorithm used to resolve the import fixup table

Resolving the import fixup table requires us to have first identified the data section within the protected binary and the metadata that identifies the import table (`IMPTBL_OFFSET`, `IMPTBL_SIZE`). The offset to the fixup table is from the start of the data section.

```python
#----------------------------------------------------------------------------
assert d.DATA_SECTION_RVA != -1
assert d.IMPTBL_OFFSET    != -1
assert d.IMPTBL_SIZE      != -1
#----------------------------------------------------------------------------
read32 = lambda index:  struct.unpack_from('<I', d.imgbuffer, index)[0]
table_start = d.DATA_SECTION_RVA+d.IMPTBL_OFFSET
table_end   = d.DATA_SECTION_RVA+d.IMPTBL_OFFSET+d.IMPTBL_SIZE
for i in range(table_start, table_end, 8): # sizeof imp_tbl_entry_t
    location = read32(i)
    fixup    = read32(i+4)
    d.imgbuffer[location:location+4] = struct.pack("<I", fixup)  # write
    d.imptbl[location] = fixup                                   # cache
    assert len(d.imptbl) == int(d.IMPTBL_SIZE / 8), (
        "failed to guarantee imptable size"
    )
```

Figure 49: Python re-implementation of the algorithm used to resolve the import fixup table

Having the start of the fixup table, we simply iterate one entry at a time and identify which import displacement (`location`) is associated with which dispatcher stub (`fixup`).

Recovering the Import

Having obtained all potential import locations from the brute-force scan and accounted for relocations in HEADERLESS mode, we can proceed with the final verifications to recover each protected import. The recovery process is conducted as follows:

1. **Decode the location into a valid call or jmp instruction**
    - Any failure in decoding indicates that the location does not contain a valid instruction and can be safely ignored.
2. **Use the memory displacement to locate the stub for the import**
    - In HEADERLESS mode, each displacement serves as a lookup key into the fixup table for the respective dispatcher.
3. **Extract the `obf_imp_t` structure within the dispatcher**
    - This is achieved by statically traversing a dispatcher's disassembly listing.
    - The first `lea` instruction encountered will contain the reference to the `obf_imp_t`.
4. **Process the `obf_imp_t` to decrypt both the DLL and API names**
    - Utilize the two RVAs contained within the structure to locate the encrypted blobs for the DLL and API names.
    - Decrypt the blobs using the outlined import decryption routine.

```
potential_stubs = _brute_find_impstubs(d)
for location in potential_stubs:
  try:
    instr = DECODE_INSTR(location)
    if not (instr.is_call() or instr.is_jmp()): continue
    if 'rip' not in instr.op_str: continue
    #-----------------------------------------------------------------
    stub_ea, stub_id = GET_STUB_DISPLACEMENT(d, instr)
    #-----------------------------------------------------------------
    stub_rfn = recover_import_stub(d, stub_ea)
    #-----------------------------------------------------------------
    ref_instr = _extract_lea_ref_instr(stub_rfn)
    #-----------------------------------------------------------------
    dll_name, api_name = GET_DLL_API_NAMES(d, ref_instr)
    #-----------------------------------------------------------------
    # call site is the only thing that's unique
    recovered_imp = RecoveredImport(
      instr, ref_instr,
      stub_id, stub_ea, stub_rfn,
      dll_name=dll_name,
      api_name=api_name
    )
    #-----------------------------------------------------------------
    d.imports[instr.ea] = recovered_imp        # call/jmp size -> Imp
    d.imp_dict_builder[dll_name].add(api_name) # DllName -> ApiNAme
  except Exception:
    continue  # these don't matter
```

Figure 50: Loop that recovers each protected import

The Python code iterates through every potential import location (`potential_stubs`) and attempts to decode each presumed `call` or `jmp` instruction to an import. A `try` / `except` block is employed to handle any failures, such as instruction decoding errors or other exceptions that may arise. The assumption is that any error invalidates our understanding of the recovery process and can be safely ignored. In the full code, these errors are logged and tracked for further analysis should they arise.

Next, the code invokes a `GET_STUB_DISPLACEMENT` helper function that obtains the RVA to the dispatcher associated with the import. Depending on the mode of protection, one of the following routines is used:

```python
def _get_stub_rva_headerless(
    d: ProtectedInput64,
    instr: x86Instr
):
    """Retrieves the stub address and stub ID for HEADERLESS protection type.
    Headerless protections treat the initial memory displacement as a lookup
    into the fixup table to find the relocated disptacher. We refer to it as
    the `stub_id` here.
    """
    assert instr.is_call() or instr.is_jmp()
    stub_id = instr.get_calljmp_mem_target()
    stub_rva = d.imptbl.get(stub_id)         # actual dispatcher location
    return stub_rva, stub_id

def _get_stub_rva_non_headerless(
    d: ProtectedInput64,
    instr: x86Instr
):
    """Retrieves the stub address for non-HEADERLESS protection type.
    We don't require a stub id (the initial memory displacement that gets
    fixed up) as non-HEADERLESS types don't have to have their imports
    relocated.
    """
    assert instr.is_call() or instr.is_jmp()
    stub_id = 0xffffffff
    stub_loc = instr.get_calljmp_mem_target()
    stub_ea = int.from_bytes(d.imgbuffer[stub_loc:stub_loc+8], 'little')
    base = d.pe.OPTIONAL_HEADER.ImageBase
    stub_ea -= base
    return stub_ea, stub_id
```

Figure 51: Routines that retrieve the stub RVA based on the protection mode

The `recover_import_stub` function is utilized to reconstruct the control flow graph (CFG) of the import stub, while `_extract_lea_ref` examines the instructions in the CFG to locate the `lea` reference to the `obf_imp_t`. The `GET_DLL_API_NAMES` function operates similarly to `GET_STUB_DISPLACEMENT`, accounting for slight differences depending on the protection mode:

```
def _get_dll_api_names_headerless(
  d: ProtectedInput64,
  ref_instr: x86Instr
):
  """Extracts and decrypts the DLL and API names for HEADERLESS
  protection type.
  """
  rva = ref_instr.ea + ref_instr.size + ref_instr.Op2.mem.disp
  dll_name_rva = int.from_bytes(d.imgbuffer[rva:rva+4], "little")
  api_name_rva = int.from_bytes(d.imgbuffer[rva+8:rva+12], "little")
  #------------------------------------------------------------------------
  dll_name = imp_crypt_str(d, d.imgbuffer[dll_name_rva:dll_name_rva+80])
  api_name = imp_crypt_str(d, d.imgbuffer[api_name_rva:api_name_rva+80])
  return dll_name, api_name

def _get_dll_api_names_non_headerless(
  d: ProtectedInput64,
  ref_instr: x86Instr
):
  """Extracts and decrypts the DLL and API names for non-HEADERLESS
  protection type.
  """
  rva = ref_instr.ea + ref_instr.size + ref_instr.Op2.mem.disp
  ea1 = int.from_bytes(d.imgbuffer[rva:rva+8], "little")
  ea2 = int.from_bytes(d.imgbuffer[rva+8:rva+16], "little")
  #------------------------------------------------------------------------
  base = d.pe.OPTIONAL_HEADER.ImageBase
  dll_name_rva = (ea1-base) & 0xFFFFFFFF
  api_name_rva = (ea2-base) & 0xFFFFFFFF
  #------------------------------------------------------------------------
  dll_name = imp_crypt_str(d, d.imgbuffer[dll_name_rva:dll_name_rva+80])
  api_name = imp_crypt_str(d, d.imgbuffer[api_name_rva:api_name_rva+80])
  return dll_name, api_name
```

Figure 52: Routines that decrypt the DLL and API blobs based on the protection mode

After obtaining the decrypted DLL and API names, the code possesses all the necessary information to reveal the import that the protection conceals. The final individual output of each import entry is captured in a `RecoveredImport` object and two dictionaries:

- `d.imports`
  - This dictionary maps the address of each protected import to its recovered state. It allows for the association of the complete recovery details with the specific location in the binary where the import occurs.
- `d.imp_dict_builder`
  - This dictionary maps each DLL name to a set of its corresponding API names. It is used to reconstruct the import table, ensuring a unique set of DLLs and the APIs utilized by the binary.

This systematic collection and organization prepare the necessary data to facilitate the restoration of the original functionality in the deobfuscated output. In Figure 53 and Figure 54, we can observe these two containers to

showcase their structure after a successful recovery:

```
In [10]: d
Out[10]: <recover.recover_core.ProtectedInput64 at 0x1c61f157c90>
In [11]: d.imports
Out[11]:
{4321: ImpStub:
   GetModuleHandleW                    (KERNEL32.dll)
   CallSite: <x86Instr> 0x0010e1 (ff1523540100)           call qword ptr [rip + 0x15423]
   LeaFetch: <x86Instr> 0x011786 (488d0d74540000)         lea rcx, [rip + 0x5474]
   StubId:   0xffffffff
   StubEa:   0x011785,
 4438: ImpStub:
   RegCreateKeyExW                     (ADVAPI32.dll)
   CallSite: <x86Instr> 0x001156 (ff15a6520100)           call qword ptr [rip + 0x152a6]
   LeaFetch: <x86Instr> 0x00d9ea (488d0de4880000)         lea rcx, [rip + 0x88e4]
   StubId:   0xffffffff
   StubEa:   0x008c51,
 5194: ImpStub:
   GetTickCount                        (KERNEL32.dll)
   CallSite: <x86Instr> 0x00144a (ff15f6550100)           call qword ptr [rip + 0x155f6]
   LeaFetch: <x86Instr> 0x003047 (488d0d87380100)         lea rcx, [rip + 0x13887]
   StubId:   0xffffffff
   StubEa:   0x0073d0,
 5813: ImpStub:
   GetLastError                        (KERNEL32.dll)
   CallSite: <x86Instr> 0x0016b5 (ff15b8540100)           call qword ptr [rip + 0x154b8]
   LeaFetch: <x86Instr> 0x00ba61 (488d0dd2ab0000)         lea rcx, [rip + 0xabd2]
   StubId:   0xffffffff
   StubEa:   0x007a45,
 6994: ImpStub:
   InternetSetOptionA                  (WININET.dll)
   CallSite: <x86Instr> 0x001b52 (ff15ba490100)           call qword ptr [rip + 0x149ba]
   LeaFetch: <x86Instr> 0x0055d9 (488d0d7a130100)         lea rcx, [rip + 0x1137a]
   StubId:   0xffffffff
   StubEa:   0x00f20c,
 7726: ImpStub:
   lstrcmpiA                           (KERNEL32.dll)
   CallSite: <x86Instr> 0x001e2e (ff15a04d0100)           call qword ptr [rip + 0x14da0]
```

Figure 53: Output of the d.imports dictionary after a successful recovery

```
In [12]: d.imp_dict_builder
Out[12]:
defaultdict(set,
            {'KERNEL32.dll': {'CloseHandle',
              'CreateEventW',
              'CreateThread',
              'DeleteCriticalSection',
              'EnterCriticalSection',
              'GetCurrentProcessId',
              'GetCurrentThreadId',
              'GetLastError',
              'GetModuleHandleW',
              'GetProcAddress',
              'GetSystemTime',
              'GetTickCount',
              'InitializeCriticalSection',
              'LeaveCriticalSection',
              'LoadLibraryA',
              'LocalAlloc',
              'LocalFree',
              'MultiByteToWideChar',
              'QueryPerformanceCounter',
              'ResetEvent',
              'ResumeThread',
              'SetEvent',
              'SetLastError',
              'Sleep',
              'WaitForSingleObject',
              'WideCharToMultiByte',
              'lstrcmpiA',
              'lstrcmpiW',
              'lstrcpyA',
              'lstrlenA',
              'lstrlenW'},
             'ADVAPI32.dll': {'RegCloseKey',
              'RegCreateKeyExW',
              'RegFlushKey',
              'RegOpenKeyExW',
```

Figure 54: Output of the d.imp_dict_builder dictionary after a successful recovery

Observing the Final Results

This final step—rebuilding the import table using this data—is performed by the `build_import_table` function in the `pefile_utils.py` source file. This part is omitted from the blog post due to its unavoidable length and the numerous tedious steps involved. However, the code is well-commented and structured to thoroughly address and showcase all aspects necessary for reconstructing the import table.

Nonetheless, the following figure illustrates how we generate a fully functional binary from a headerless-protected input. Recall that a headerless-protected input is a raw, headerless PE binary, almost analogous to a shellcode blob. From this blob we produce an entirely new, functioning binary with the entirety of its import protection completely restored. And we can do the same for all protection modes.

Name
decrypted-payload-final-form.dll

Module Name 00086104 0000000D	Imports (n)	Original... 00086000 00000004	TimeDateStamp 00086004 00000004	ForwarderChain 00086008 00000004	Name 0008600C 00000004	FirstThunk 00086010 00000004
KERNEL32.dll	122	000871D8	00000000	00000000	00086104	000871D8
WININET.dll	13	000875B0	00000000	00000000	00086111	000875B0
ADVAPI32.dll	35	00087620	00000000	00000000	0008611D	00087620
WS2_32.dll	22	00087740	00000000	00000000	0008612A	00087740
ole32.dll	5	000877F8	00000000	00000000	00086135	000877F8
USER32.dll	17	00087828	00000000	00000000	0008613F	00087828
OLEAUT32.dll	3	000878B8	00000000	00000000	0008614A	000878B8
WTSAPI32.dll	5	000878D8	00000000	00000000	00086157	000878D8
USERENV.dll	2	00087908	00000000	00000000	00086164	00087908
PSAPI.DLL	5	00087920	00000000	00000000	00086170	00087920
SHLWAPI.dll	1	00087950	00000000	00000000	0008617A	00087950
SHELL32.dll	1	00087960	00000000	00000000	00086186	00087960

OFTs	FTs (IAT)	Hint	Name	Demangled
0000000000086192	0000000000086192	0000	LoadResource	NA
00000000000861A1	00000000000861A1	0000	VirtualAlloc	NA
00000000000861B0	00000000000861B0	0000	WideCharToMultiByte	NA
00000000000861C6	00000000000861C6	0000	GetProcAddress	NA
00000000000861D7	00000000000861D7	0000	FindFirstFileW	NA
00000000000861E8	00000000000861E8	0000	DeleteProcThreadAttributeList	NA
0000000000086208	0000000000086208	0000	SetUnhandledExceptionFilter	NA
0000000000086226	0000000000086226	0000	AttachConsole	NA
0000000000086236	0000000000086236	0000	FindResourceW	NA
0000000000086246	0000000000086246	0000	GetSystemDirectoryW	NA
000000000008625C	000000000008625C	0000	SetFileTime	NA
000000000008626A	000000000008626A	0000	TerminateThread	NA
000000000008627C	000000000008627C	0000	FlushFileBuffers	NA
000000000008628F	000000000008628F	0000	WaitForMultipleObjects	NA
00000000000862A8	00000000000862A8	0000	WriteFile	NA

Figure 55: Display of completely restored import table for a binary protected in HEADERLESS mode

Building Relocations in Deobfuscated Binaries

Now that we can fully recover the CFG of protected binaries and provide complete restoration of the original import tables, the final phase of the deobfuscator involves merging these elements to produce a functional deobfuscated binary. The code responsible for this process is encapsulated within the `recover_output64.py` and the `pefile_utils.py` Python files.

The rebuild process comprises two primary steps:

1. Building the Output Image Template
2. Building Relocations

1. Building the Output Image Template

Creating an output image template is essential for generating the deobfuscated binary. This involves two key tasks:

- **Template PE Image:** A Portable Executable (PE) template that serves as the container for the output binary that incorporates the restoration of all obfuscated components. We also need to be cognizant of all

the different characteristics between in-memory PE executables and on-file PE executables.

- **Handling Different Protection Modes:** Different protection modes and input stipulate different requirements.
    - **Headerless variants** have their file headers stripped. We must account for these variations to accurately reconstruct a functioning binary.
    - **Selective protection** preserves the original imports to maintain functionality as well as includes a specific import protection for all the imports leveraged within the selected functions.

2. Building Relocations

Building relocations is a critical and intricate part of the deobfuscation process. This step ensures that all address references within the deobfuscated binary are correctly adjusted to maintain functionality. It generally revolves around the following two phases:

- **Calculating Relocatable Displacements**: Identifying all memory references within the binary that require relocation. This involves calculating the new addresses where these references will point to. The technique we will use is generating a lookup table that maps original memory references to their new relocatable addresses.
- **Apply Fixups:** Modifies the binary's code to reflect the new relocatable addresses. This utilizes the aforementioned lookup table to apply necessary fixups to all instruction displacements that reference memory. This ensures that all memory references within the binary correctly point to their intended locations.

We intentionally omit the details of showcasing the rebuilding of the output binary image because, while essential to the deobfuscation process, it is straightforward enough and just overly tedious to be worthwhile examining in any depth. Instead, we focus exclusively on relocations, as they are more nuanced and reveal important characteristics that are not as apparent but must be understood when rewriting binaries.

Overview of the Relocation Process

Rebuilding relocations is a critical step in restoring a deobfuscated binary to an executable state. This process involves adjusting memory references within the code so that all references point to the correct locations after the code has been moved or modified. On the x86-64 architecture, this primarily concerns instructions that use **RIP-relative addressing**, a mode where memory references are relative to the instruction pointer.

Relocation is necessary when the layout of a binary changes, such as when code is inserted, removed, or shifted during deobfuscation. Given our deobfuscation approach extracts the original instructions from the obfuscator, we are required to relocate each recovered instruction appropriately into a new code segment. This ensures that the deobfuscated state preserves the validity of all memory references and that the accuracy of the original control and data flow is sustained.

Understanding Instruction Relocation

Instruction relocation revolves around the following:

- **Instruction's memory address:** the location in memory where an instruction resides.
- **Instruction's memory memory references:** references to memory locations used by the instruction's operands.

Consider the following two instructions as illustrations:

```
0x1000: E91D3E0000 jmp +0x4E22
0x4E22: 4C8D0557010000 lea r8, ds:[rip+0x157]
```

Figure 56: Illustration of two instructions that require relocation

1. **Unconditional `jmp` instruction**This instruction is located at memory address `0x1000.` It references its branch target at address `0x4E22` . The displacement encoded within the instruction is `0x3E1D` , which is used to calculate the branch target relative to the instruction's position. Since it employs RIP-relative addressing, the destination is calculated by adding the displacement to the length of the instruction and its memory address.

2. `lea` **instruction**This is the branch target for the `jmp` instruction located at `0x4E22` . It also contains a memory reference to the data segment, with an encoded displacement of `0x157` .

When relocating these instructions, we must address both of the following aspects:

- **Changing the instruction's address:** When we move an instruction to a new memory location during the relocation process, we inherently change its memory address. For example, if we relocate this instruction from `0x1000` to `0x2000` , the instruction's address becomes `0x2000` .

- **Adjusting memory displacements:** The displacement within the instruction (`0x3E1D` for the `jmp` , `0x157` for the `lea`) is calculated based on the instruction's original location and the location of its reference. If the instruction moves, the displacement no longer points to the correct target address. Therefore, we must recalculate the displacement to reflect the instruction's new position.

```
0x2000: E91B000000 jmp +0x2020
0x2020: 4C8D05XXXX0000 lea r8, ds:[rip+0xXXXX]
```

Figure 57: Updated illustration demonstration of what relocation would look like

When relocating instructions during the deobfuscation process, we must ensure accurate control flow and data access. This requires us to adjust both the instruction's memory address and any displacements that reference other memory locations. Failing to update these values invalidates the recovered CFG.

What Is RIP-Relative Addressing?

RIP-relative addressing is a mode where the instruction references memory at an offset relative to the **RIP** (instruction pointer) register, which points to the next instruction to be executed. Instead of using absolute

addresses, the instruction encapsulates the referenced address via a signed 32-bit displacement from the current instruction pointer.

Addressing relative to the instruction pointer exists on x86 as well, but only for control-transfer instructions that support a relative displacement (e.g., JCC conditional instructions, near CALLs, and near JMPs). The x64 ISA extended this to account for almost all memory references being RIP-relative. For example, [most data references in x64 Windows binaries are RIP-relative](#).

An excellent tool to visualize the intricacies of a decoded Intel x64 instruction is [ZydisInfo](#). Here we use it to illustrate how a LEA instruction (encoded as `488D151B510600`) references RIP-relative memory at `0x6511b`.

```
ZydisInfo.exe -64 48 8D 15 1B 51 06 00
== [    BASIC ] ============================================================================
   MNEMONIC: lea [ENC: DEFAULT, MAP: DEFAULT, OPC: 0x8D]
     LENGTH:  7
        SSZ: 64
       EOSZ: 64
       EASZ: 64
   CATEGORY: MISC
    ISA-SET: I86
    ISA-EXT: BASE
 EXCEPTIONS: NONE
 ATTRIBUTES: HAS_MODRM HAS_REX IS_RELATIVE
  OPTIMIZED: 48 8D 15 1B 51 06 00

== [ OPERANDS ] ============================================================================
##       TYPE  VISIBILITY  ACTION      ENCODING   SIZE  NELEM  ELEMSZ  ELEMTYPE                        VALUE
--  ---------  ----------  ------  ------------   ----  -----  ------  --------  --------------------------
 0   REGISTER    EXPLICIT       W     MODRM_REG     64      1      64       INT                          rdx
 1     MEMORY    EXPLICIT    NONE      MODRM_RM     64      1       0       INT  TYPE  =                AGEN
                                                                                 SEG   =                  ds
                                                                                 BASE  =                 rip
                                                                                 INDEX =                none
                                                                                 SCALE =                   0
                                                                                 DISP  =  0x000000000006511B
--  ---------  ----------  ------  ------------   ----  -----  ------  --------  --------------------------

== [      ATT ] ============================================================================
   ABSOLUTE: lea 0x000000000065122, %rdx
   RELATIVE: lea 0x6511B(%rip), %rdx

== [    INTEL ] ============================================================================
   ABSOLUTE: lea rdx, qword ptr ds:[0x000000000065122]
   RELATIVE: lea rdx, qword ptr ds:[rip+0x6511B]

== [ SEGMENTS ] ============================================================================
48 8D 15 1B 51 06 00
:  :  :  :..DISP
:  :  :..MODRM
:  :..OPCODE
:..REX
```

Figure 58: ZydisInfo output for the lea instruction

For most instructions, the displacement is encoded in the final four bytes of the instruction. When an immediate value is stored at a memory location, the immediate follows the displacement. Immediate values are restricted to a maximum of 32 bits, meaning 64-bit immediates cannot be used following a displacement. However, 8-bit and 16-bit immediate values are supported within this encoding scheme.

```
ZydisInfo.exe -64 c705d11d060001000000000000010
== [    INTEL ] ==========================================================================
   ABSOLUTE: mov dword ptr ds:[0x0000000000061DDB], 0x01
   RELATIVE: mov dword ptr ds:[rip+0x61DD1], 0x01

== [ SEGMENTS ] ==========================================================================
C7 05 D1 1D 06 00 01 00 00 00
:  :  :           :..IMM
:  :  :..DISP
:  :..MODRM
:..OPCODE
```

Figure 59: ZydisInfo output for the mov instruction storing an immediate operand

Displacements for control-transfer instructions are encoded as immediate operands, with the RIP register implicitly acting as the base. This is evident when decoding a `jnz` instruction, where the displacement is directly embedded within the instruction and calculated relative to the current RIP.

```
ZydisInfo.exe -64 0F 85 CF 00 00 00
== [    BASIC ] ==========================================================================
   MNEMONIC: jnz [ENC: DEFAULT, MAP: 0F, OPC: 0x85]
     LENGTH:  6
        SSZ: 64
       EOSZ: 64
       EASZ: 64
   CATEGORY: COND_BR
    ISA-SET: I86
    ISA-EXT: BASE
 EXCEPTIONS: NONE
 ATTRIBUTES: IS_RELATIVE CPUFLAG_ACCESS ACCEPTS_BND ACCEPTS_BRANCH_HINTS
  OPTIMIZED: 0F 85 CF 00 00 00

== [ OPERANDS ] ==========================================================================
##       TYPE  VISIBILITY  ACTION      ENCODING   SIZE  NELEM  ELEMSZ  ELEMTYPE                        VALUE
--  ---------  ----------  ------  ------------   ----  -----  ------  --------  -------------------------
 0  IMMEDIATE    EXPLICIT       R        JIMM32     32      1      32       INT  [S R 32] 0x00000000000000CF
 1   REGISTER      HIDDEN      RW          NONE     64      1      64       INT                        rip
 2   REGISTER      HIDDEN       R          NONE     64     64       1       INT                     rflags
--  ---------  ----------  ------  ------------   ----  -----  ------  --------  -------------------------

== [    FLAGS ] ==========================================================================
    ACTIONS: [ZF  : T  ]
       READ: 0x00000040
    WRITTEN: 0x00000000

== [      ATT ] ==========================================================================
   ABSOLUTE: jnz 0x00000000000000D5
   RELATIVE: jnz +0xD5

== [    INTEL ] ==========================================================================
   ABSOLUTE: jnz 0x00000000000000D5
   RELATIVE: jnz +0xD5

== [ SEGMENTS ] ==========================================================================
0F 85 CF 00 00 00
:     :..IMM
:..OPCODE
```

Figure 60: ZydisInfo output for the jnz instruction with an immediate operand as the displacement

Steps in the Relocation Process

For rebuilding relocations we take the following approach:

1. **Rebuilding the code section and creating a relocation map**With the recovered CFG and imports, we commit the changes to a new code section that contains the fully deobfuscated code. We do this by:

   - **Function-by-function processing**: rebuild each function one at a time. This allows us to manage the relocation of each instruction within its respective function.

- **Tracking instruction locations**: As we rebuild each function, we track the new memory locations of each instruction. This involves maintaining a global relocation dictionary that maps original instruction addresses to their new addresses in the deobfuscated binary. This dictionary is crucial for accurately updating references during the fixup phase.

2. **Applying fixups**After rebuilding the code section and establishing the relocation map, we proceed to modify the instructions so that their memory references point to the correct locations in the deobfuscated binary. This restores the binary's complete functionality and is achieved by **adjusting memory references** to code or data an instruction may have.

Rebuilding the Code Section and Creating a Relocation Map

To construct the new deobfuscated code segment, we iterate over each recovered function and copy all instructions sequentially, starting from a fixed offset—for example, `0x1000`. During this process, we build a global relocation dictionary (`global_relocs`) that maps each instruction to its relocated address. This mapping is essential for adjusting memory references during the fixup phase.

The `global_relocs` dictionary uses a tuple as the key for lookups, and each key is associated with the relocated address of the instruction it represents. The tuple consists of the following three components:

1. **Original starting address of the function**: The address where the function begins in the protected binary. It identifies the function to which the instruction belongs.

2. **Original instruction address within the function**: The address of the instruction in the protected binary. For the first instruction in a function, this will be the function's starting address.

3. **Synthetic boundary JMP flag**: A boolean value indicating whether the instruction is a synthetic boundary jump introduced during normalization. These synthetic instructions were not present in the original obfuscated binary, and we need to account for them specifically during relocation because they have no original address.

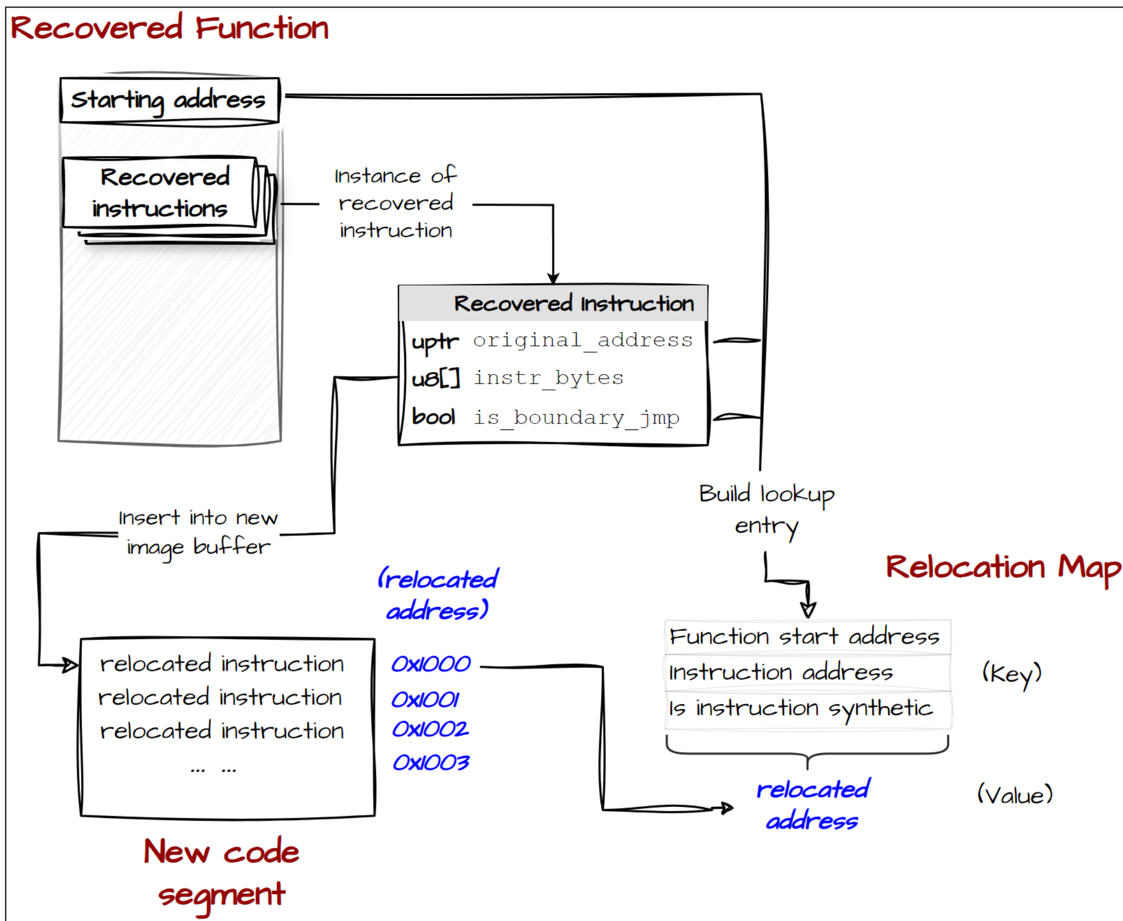


Figure 61: Illustration of how the new code segment and relocation map are generated

The following Python code implements the logic outlined in Figure 61. Error handling and logging code has been stripped for brevity.

```
#-----------------------------------------------------------------------
# `d.global_relocs` is the global relocation lookup map
curr_off = starting_off
for func_ea, rfn in d.cfg.items():  # func_ea: int; rfn: RecoveredFunc
    #-------------------------------------------------------------------
    # process the function first
    rfn.reloc_ea = curr_off         # relocated start ea
    d.global_relocs[(
        func_ea,          # function start address
        func_ea,          # first instruction in function (start address)
        False             # a function is never a boundary instruction
    )] = curr_off
    #-------------------------------------------------------------------
    # process each instruction in the function
    for r: RecoveredInstr in rfn.normalized_flow:
        r.reloc_ea = curr_off   # relocated instr ea
        d.global_relocs[(
            func_ea,            # function instr is a part of
            r.instr.ea,         # original instr location
            r.is_boundary_jmp   # is it a synthetic boundary instruction
        )] = curr_off
        #---------------------------------------------------------------
        ops = r.updated_bytes if r.updated_bytes else r.instr.bytes
        size = len(ops)
        d.newimgbuffer[curr_off:curr_off + size] = ops
        curr_off += size
    curr_off = align_to_16_byte_boundary(curr_off + 8)
```

Figure 62: Python logic that implements the building of the code segment and generation of the relocation map

1. **Initialize current offset**
   Set the starting point in the new image buffer where the code section will be placed. The variable `curr_off` is initialized to `starting_off`, which is typically `0x1000`. This represents the conventional start address of the `.text` section in PE files. For SELECTIVE mode, this will be the offset to the start of the protected function.
2. **Iterate over recovered functions**
   Loop through each recovered function in the deobfuscated control flow graph (`d.cfg`). `func_ea` is the original function entry address, and `rfn` is a `RecoveredFunc` object encapsulating the recovered function's instructions and metadata.
   1. **Handle the function start address first**
      1. **Set function's relocated start address:** Assign the current offset to `rfn.reloc_ea`, marking where this function will begin in the new image buffer.
      2. **Update global relocation map:** Add an entry to the global relocation map `d.global_relocs` to map the original function address to its new location.
   2. **Iterate over each recovered instruction**
      Loop through the normalized flow of instructions within the function. We use the `normalized_flow` as it allows us to iterate over each instruction linearly as we apply it to the new image.
      1. **Set instruction's relocated address:** Assign the current offset to `r.reloc_ea`, indicating where this instruction will reside in the new image buffer.

   2. **Update global relocation map:** Add an entry to `d.global_relocs` for the instruction, mapping its original address to the relocated address.
   3. **Update the output image:** Write the instruction bytes to the new image buffer `d.newimgbuffer` at the current offset. If the instruction was modified during deobfuscation (`r.updated_bytes`), use those bytes; otherwise, use the original bytes (`r.instr.bytes`).
   4. **Advance the offset:** Increment `curr_off` by the size of the instruction to point to the next free position in the buffer and move on to the next instruction until the remainder are exhausted.
3. **Align current offset to 16-byte boundary**After processing all instructions in a function, align `curr_off` to the next 16-byte boundary. We use 8 bytes as an arbitrary pointer-sized value from the last instruction to pad so that the next function won't conflict with the last instruction of the previous function. This further ensures proper memory alignment for the next function, which is essential for performance and correctness on x86-64 architectures. Then repeat the process from step 2 until all functions have been exhausted.

This step-by-step process accurately rebuilds the deobfuscated binary's executable code section. By relocating each instruction, the code prepares the output template for the subsequent fixup phase, where references are adjusted to point to their correct locations.

Applying Fixups

After building the deobfuscated code section and relocating each recovered function in full, we apply fixups to correct addresses within the recovered code. This process adjusts the instruction bytes in the new output image so that all references point to the correct locations. It is the final step in reconstructing a functional deobfuscated binary.

We categorize fixups into three distinct categories, based primarily on whether they apply to control flow or data flow instructions. We further distinguish between two types of control flow instructions: standard branching instructions and those introduced by the obfuscator through the import protection. Each type has specific nuances that require tailored handling, allowing us to apply precise logic to each category.

1. **Import Relocations**: These involve calls and jumps to recovered imports.

2. **Control Flow Relocations**: All standard control flow branching branching instructions.

3. **Data Flow Relocations**: Instructions that reference static memory locations.

Using these three categorizations, the core logic boils down to the following two phases:

1. **Resolving displacement fixups**
   - Differentiate between displacements encoded as immediate operands (branching instructions) and those in memory operands (data accesses and import calls).
   - Calculate the correct fixup values for these displacements using the `d.global_relocs` map generated prior.
2. **Update the output image buffer**

    - Once the displacements have been resolved, write the updated instruction bytes into the new code segment to reflect the changes permanently.

To achieve this, we utilize several helper functions and lambda expressions. The following is a step-by-step explanation of the code responsible for calculating the fixups and updating the instruction bytes.

```
PACK_FIXUP = lambda fixup: bytearray(struct.pack("<I", fixup))
CALC_FIXUP = lambda dest,size: (dest-(r.reloc_ea+size)) & 0xFFFFFFFF
IS_IN_DATA = lambda dest: dest in d.data_range_rva
#-----------------------------------------------------------------------
def resolve_disp_fixup_and_apply(
    r: RecoveredInstr,
    dest: int
):
    assert r.instr.disp_size == 4
    fixup = CALC_FIXUP(dest, r.instr.size)
    offset = r.instr.disp_offset
    r.updated_bytes[offset:offset+4] = PACK_FIXUP(fixup)
#-----------------------------------------------------------------------
def resolve_imm_fixup_and_apply(
    r: RecoveredInstr,
    reloc_dest: int,
):
    if r.instr.is_call():
        r.updated_bytes = d.ks.asm(
            f'{r.instr.mnemonic} {reloc_dest:#08x}',
            r.reloc_ea)[0]
    else:
        fixup = CALC_FIXUP(reloc_dest, len(r.updated_bytes))
        if r.instr.is_jmp():
            assert len(r.updated_bytes) == 5 # placeholders
            r.updated_bytes[1:5] = PACK_FIXUP(fixup)
        elif r.instr.is_jcc():
            assert len(r.updated_bytes) == 6 # placeholders
            r.updated_bytes[2:6] = PACK_FIXUP(fixup)
#-----------------------------------------------------------------------
def update_reloc_in_img(
    r: RecoveredInstr,
    tag: str # tag to log what reloc type
):
    r.reloc_instr = d.md.decode_buffer(bytes(r.updated_bytes),r.reloc_ea)
    d.newimgbuffer[r.reloc_ea:r.reloc_ea+r.reloc_instr.size] = (
        r.updated_bytes
    )
```

Figure 63: Helper routines that aid in applying fixups

- **Define lambda helper expressions**
    - `PACK_FIXUP` : packs a 32-bit fixup value into a little-endian byte array.
    - `CALC_FIXUP` : calculates the fixup value by computing the difference between the destination address (`dest`) and the end of the current instruction (`r.reloc_ea + size`), ensuring it fits within 32 bits.
    - `IS_IN_DATA` : checks if a given address is within the data section of the binary. We exclude relocating these addresses, as we preserve the data section at its original location.
- **Resolve fixups for each instruction**

- Import and data flow relocations
  - Utilize the `resolve_disp_fixup_and_apply` helper function as both encode the displacement within a memory operand.
- Control flow relocations
  - Use the `resolve_imm_fixup_and_apply` helper as the displacement is encoded in an immediate operand.
  - During our CFG recovery, we transformed each `jmp` and `jcc` instruction to its near jump equivalent (from 2 bytes to 6 bytes) to avoid the shortcomings of 1-byte short branches.
    - We force a 32-bit displacement for each branch to guarantee a sufficient range for every fixup.

- **Update the output image buffer**
  - Decode the updated instruction bytes to have it reflect within the `RecoveredInstr` that represents it.
  - Write the updated bytes to the new image buffer
    - `updated_bytes` reflects the final opcodes for a fully relocated instruction.

With the helpers in place, the following Python code implements the final processing for each relocation type.

```
#-----------------------------------------------------------------------
# IMPORTS
for r in rfn.relocs_imports: # r: RecoveredInstr
    r.updated_bytes = bytearray(r.instr.bytes)
    imp_entry = d.imports.get(r.instr.ea)
    imp_entry.new_rva = d.import_to_rva_map.get(imp_entry.api_name)
    resolve_disp_fixup_and_apply(r, imp_entry.new_rva)
    update_reloc_in_img(r, "Import")
#-----------------------------------------------------------------------
# CONTROL FLOW
for r in rfn.relocs_ctrlflow:
    dest = r.instr.get_op1_imm()
    reloc_dest = (
        d.global_relocs.get((dest, dest, False)) if r.instr.is_call()
        else d.global_relocs.get((rfn.func_start_ea, dest, False))
    )
    resolve_imm_fixup_and_apply(r, reloc_dest)
    update_reloc_in_img(r, tag="CtrlFlow")
#-----------------------------------------------------------------------
# DATA ACCESS
KNOWN = [
    x86.X86_INS_INC, x86.X86_INS_LEA, x86.X86_INS_CMOVE,
    x86.X86_INS_MOV, x86.X86_INS_CMP, x86.X86_INS_AND
]
for r in rfn.relocs_dataflow:
    if not r.instr.id in KNOWN:
        insert_new_into_known(r)
    r.updated_bytes = bytearray(r.instr.bytes)
    instr_tag = r.instr.mnemonic.upper()
    reloc_dest = r.instr.disp_dest
    if not IS_IN_DATA(reloc_dest):
        reloc_dest = d.global_relocs.get((reloc_dest,reloc_dest,False))
    resolve_disp_fixup_and_apply(r, reloc_dest)
    update_reloc_in_img(r, "DataFlow")
```

Figure 64: The three core loops that address each relocation category

- **Import Relocations:** The first for loop handles fixups for import relocations, utilizing data generated during the Import Recovery phase. It iterates over every recovered instruction `r` within the `rfn.relocs_imports` cache and does the following:
  - **Prepare updated instruction bytes:** initialize `r.updated_bytes` with a mutable copy of the original instruction bytes to prepare it for modification.
  - **Retrieve import entry and displacement:** obtain the import entry from the imports dictionary `d.imports` and retrieve the new RVA from `d.import_to_rva_map` using the import's API name.
  - **Apply fixup:** use the `resolve_disp_fixup_and_apply` helper to calculate and apply the fixup for the new RVA. This adjusts the instruction's displacement to correctly reference the imported function.
  - **Update image buffer:** write `r.updated_bytes` back into the new image using `update_reloc_in_img`. This finalizes the fixup for the instruction in the output image.
- **Control Flow Relocations:** The second for loop handles fixups for control flow branching relocations (`call`, `jmp`, `jcc`). Iterating over each entry in `rfn.relocs_ctrlflow`, it does the following:
  - **Retrieve destination**: extract the original branch destination target from the immediate operand.
  - **Get relocated address**: reference the relocation dictionary `d.global_relocs` to obtain the branch target's relocated address. If it's a call target, then we specifically look up the relocated address for the start of the called function.
  - **Apply fixup**: use `resolve_imm_fixup_and_apply` to adjust the branch target to its relocated address.
  - **Update buffer**: finalize the fixup by writing `r.updated_bytes` back into the new image using `update_reloc_in_img`.
- **Data Flow Relocations:** The final loop handles the resolution of all static memory references stored within `rfn.relocs_dataflow`. First, we establish a list of `KNOWN` instructions that require data reference relocations. Given the extensive variety of such instructions, this categorization simplifies our approach and ensures a comprehensive understanding of all possible instructions present in the protected binaries. Following this, the logic mirrors that of the import and control flow relocations, systematically processing each relevant instruction to accurately adjust their memory references.

After reconstructing the code section and establishing the relocation map, we proceeded to adjust each instruction categorized for relocation within the deobfuscated binary. This was the final step in restoring the output binary's full functionality, as it ensures that each instruction accurately references the intended code or data segments.

Observing the Results

To demonstrate our deobfuscation library for ScatterBrain, we conduct a test study showcasing its functionality. For this test study, we select three samples: a **POISONPLUG.SHADOW** headerless backdoor and two embedded plugins.

We develop a Python script, `example_deobfuscator.py`, that consumes from our library and implements all of the recovery techniques outlined earlier. Figure 65 and Figure 66 showcase the code within our example deobfuscator:

```python
from recover.recover_core import *
from recover.recover_dispatchers import recover_instruction_dispatchers
from recover.recover_imports     import recover_imports, recover_imports_merge
from recover.recover_functions   import recover_recursive_in_full
from recover.recover_output64    import rebuild_output
#--------------------------------------------------------------------------------
def CASE_F():
    """
    samples/F:
        embedded_plugin_2000CC24.dll   (complete obfuscated binaries)
        embedded_plugin_2000FE24.dll   (complete obfuscated binaries)
        780EBC3FAE807DD6E2039A2354C50388-decrypted-backdoor.bin
    """
    PATH =  r"C:/tmp/poison-plug-shadow/samples/case_f/"
    IMP_CONST = 0x6817FD83
    #--------------------------------------------------------------------------------
    def TEST_HEADERLESS_BACKDOOR():
        d = ProtectedInput64(
            "".join([PATH, "780EBC3FAE807DD6E2039A2354C50388-backdoor-decrypted.bin"]),
            ProtectionType.HEADERLESS,
            imp_decrypt_const=IMP_CONST,
            mutation_rules=RUL e_SET_1)
        #--------------------------------------------------------------------------------
        recover_instruction_dispatchers(d)
        assert len(d.dispatcher_locs) == 0x4090, f'length is {len(d.dispatcher_locs)}'
        #--------------------------------------------------------------------------------
        recover_imports_merge(d)
        assert len(d.imports) == 0x46f, f'import length is {len(d.imports)}'
        d.log.info(f"Recovered {len(d.imports)} protected imports")
        #--------------------------------------------------------------------------------
        d.cfg = recover_recursive_in_full(d, 0)
        assert len(d.cfg.items()) == 495, "failed to recover known function amount"
        d.log.info(f"Recovered {len(d.cfg.items())} protected functions.")
        #--------------------------------------------------------------------------------
        rebuild_output(d)
        d.dump_newimgbuffer_to_disk()
        #--------------------------------------------------------------------------------
        d.log.info("Done\n" + "="*90)
    #--------------------------------------------------------------------------------
```

Figure 65: The first half of the Python code in example_deobfuscator.py

```
    #---------------------------------------------------------------------------
    def TEST_PLUGINS():
        d1: ProtectedInput64 = ProtectedInput64(
            "".join([PATH, "embedded_plugin_2000CC24.dll"]),
            ProtectionType.FULL,
            imp_decrypt_const=IMP_CONST,
            mutation_rules=RULE_SET_1)
        d2: ProtectedInput64 = ProtectedInput64(
            "".join([PATH, "embedded_plugin_2000FE24.dll"]),
            ProtectionType.FULL,
            imp_decrypt_const=IMP_CONST,
            mutation_rules=RULE_SET_1)
        #-----------------------------------------------------------------------
        recover_instruction_dispatchers(d1)
        recover_instruction_dispatchers(d2)
        assert len(d1.dispatcher_locs) == 1332
        assert len(d2.dispatcher_locs) == 1883
        #-----------------------------------------------------------------------
        recover_imports_merge(d1)
        recover_imports_merge(d2)
        assert len(d1.imports) == 76
        assert len(d2.imports) == 84
        #-----------------------------------------------------------------------
        d1.cfg = recover_recursive_in_full(d1, d1.pe.OPTIONAL_HEADER.AddressOfEntryPoint)
        d2.cfg = recover_recursive_in_full(d2, d2.pe.OPTIONAL_HEADER.AddressOfEntryPoint)
        assert len(d1.cfg.items()) == 35
        assert len(d2.cfg.items()) == 60
        #-----------------------------------------------------------------------
        rebuild_output(d1)
        rebuild_output(d2)
        d1.dump_newimgbuffer_to_disk()
        d2.dump_newimgbuffer_to_disk()
        #-----------------------------------------------------------------------
        d1.log.info("Done\n" + "="*90)

    TEST_HEADERLESS_BACKDOOR()
    TEST_PLUGINS()

CASE_F()
```

Figure 66: The second half of the Python code in example_deobfuscator.py

Running `example_deobfuscator.py` we can see the following. Note, it takes a bit given we have to emulate more than 16,000 instruction dispatchers that were found within the headerless backdoor.

```
kd ❯ ls ..\..\samples\case_f\

    Directory: C:\tmp\poison-plug-shadow\samples\case_f

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---           1/22/2025  2:05 PM         548579 780EBC3FAE807DD6E2039A2354C50388-backdoor-decrypted.bin
-a---           1/22/2025  2:05 PM          52224 embedded_plugin_2000CC24.dll
-a---           1/22/2025  2:05 PM          65024 embedded_plugin_2000FE24.dll

❯ deobfuscator
kd ❯ C:\Python311\python.exe .\example_deobfuscator.py
[ProtectedImage64::INFO]: processing input:
        Filepath: C:\tmp\poison-plug-shadow\samples\case_f\780EBC3FAE807DD6E2039A2354C50388-backdoor-decrypted.bin
        Basename: 780EBC3FAE807DD6E2039A2354C50388-backdoor-decrypted
        SHA256:   23016B576FB3EC5A080870B9CB084E3FFDE3C091907308968A253FFF1FD61628
        MD5:      780EBC3FAE807DD6E2039A2354C50388
        Mode:     ProtectionType.HEADERLESS
[ProtectedImage64::INFO]: successfully recovered all known blob data:
        blob 0xa0 at 0x0675e8 with size 0x87b
        blob 0x20 at 0x067e67 with size 0x1624
        blob 0x20 at 0x06948f with size 0xcc24
        blob 0x20 at 0x0760b7 with size 0xfe24
[ProtectedImage64::INFO]: headerless variant metadata header recovered and verified:
        .data section RVA:  0x062000
        .data section size: 0x023ee3
        ImportTable offset: 0x004000 (from start of .data section)
        ImportTable size:   0x0015c0 (in bytes)
[ProtectedImage64::INFO]: Starting instruction dispatcher recovery
```

Figure 67: The three core loops that address each relocation category

Focusing on the headerless backdoor both for brevity and also because it is the most involved in deobfuscating, we first observe its initial state inside the IDA Pro disassembler before we inspect the output results from our deobfuscator. We can see that it is virtually impenetrable to analysis.

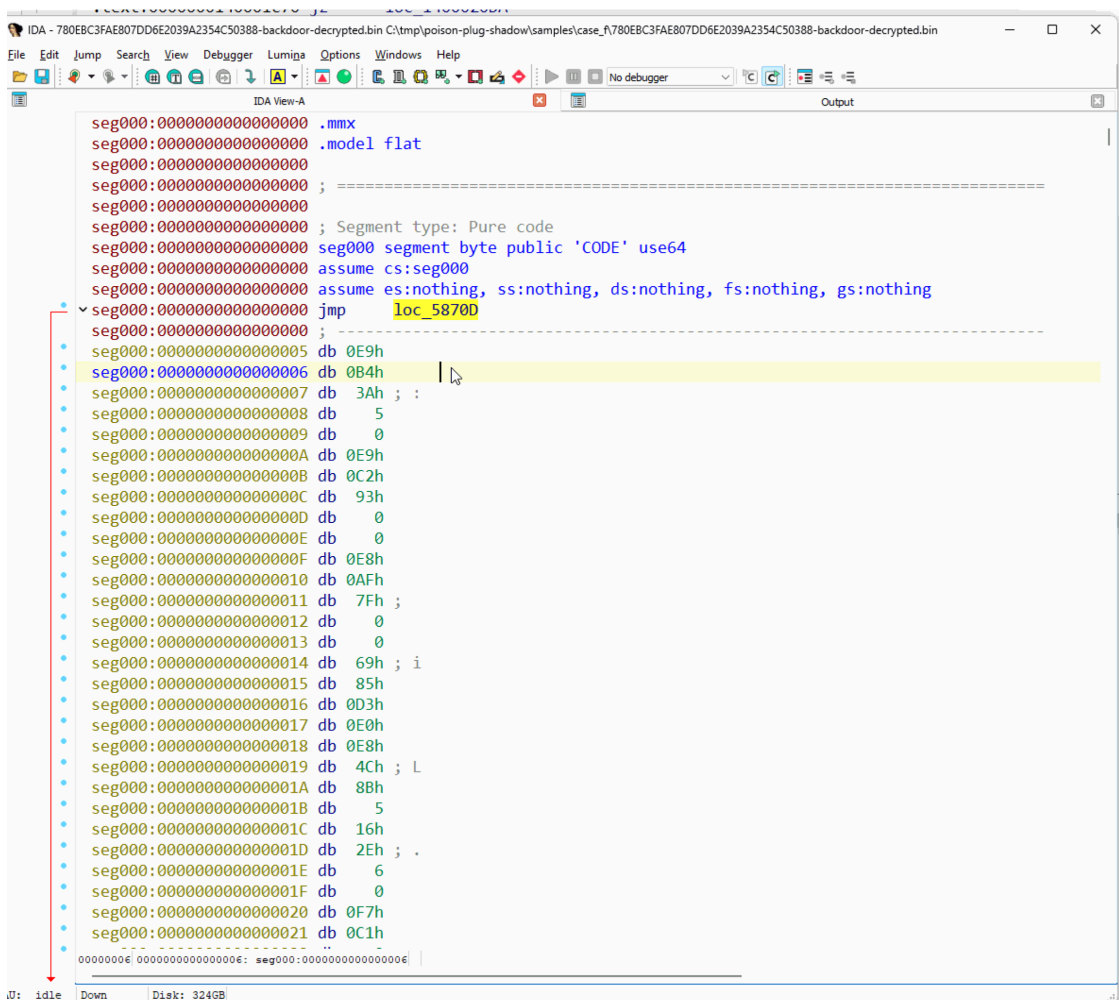


Figure 68: Observing the obfuscated headerless backdoor in IDA Pro

After running our example deobfuscator and producing a brand new deobfuscated binary, we can see the drastic difference in output. All the original control flow has been recovered, all of the protected imports have been restored, and all required relocations have been applied. We also account for the deliberately removed PE header of the headerless backdoor that ScatterBrain removes.

```
Pseudocode-A
1 // attributes: thunk
2 __int64 start()
3 {
4   return start_0();
5 }

===========================================================================
RET 0008 rax         __int64;
TOTAL STKARGS SIZE: 32
---------------------------------------------------------------------------
void sub_1400010A0( void );
__int64 sub_140001B80( &v5, &v4 );
HANDLE GetCurrentProcess( void );
void TerminateProcess( CurrentProcess, v0 );
HANDLE GetCurrentThread( void );
void TerminateThread( CurrentThread, v0 );
---------------------------------------------------------------------------
__int64 start_0()
{
  UINT v0; // ebx
  HANDLE CurrentProcess; // rax
  HANDLE CurrentThread; // rax
  int v4; // [rsp+10h] [rbp+8h] BYREF
  int v5; // [rsp+18h] [rbp+10h] BYREF

  sub_1400010A0();
  v5 = 0;
  v4 = 0;
  v0 = sub_140001B80(&v5, &v4);
  if ( v4 )
  {
    CurrentProcess = GetCurrentProcess();
    TerminateProcess(CurrentProcess, v0);
  }
  if ( v5 )
  {
    CurrentThread = GetCurrentThread();
    TerminateThread(CurrentThread, v0);
  }
  return v0;
}
```

```
IDA View-A
.text:0000000140001000 ; ===========================================================================
.text:0000000140001000
.text:0000000140001000 ; Segment type: Pure code
.text:0000000140001000 ; Segment permissions: Read/Execute
.text:0000000140001000 _text segment para public 'CODE' use64
.text:0000000140001000 assume cs:_text
.text:0000000140001000 ;org 140001000h
.text:0000000140001000 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
.text:0000000140001000
.text:0000000140001000 ; ============== S U B R O U T I N E =======================================
.text:0000000140001000
.text:0000000140001000 ; Attributes: thunk
.text:0000000140001000
.text:0000000140001000 public start
.text:0000000140001000 start proc near
.text:0000000140001000 jmp     $+5
.text:0000000140001000 start endp
.text:0000000140001000
00001000 0000000140001000: start (Synchronized with Pseudocode-A)
```

Figure 69: Observing the deobfuscated headerless backdoor in IDA Pro

Given we produce functional binaries as part of the output, the subsequent deobfuscated binary can be either run directly or debugged within your favorite debugger of choice.

Figure 70: Debugging the deobfuscated headerless backdoor in everyone's favorite debugger

Conclusion

In this blog post, we delved into the sophisticated ScatterBrain obfuscator used by POISONPLUG.SHADOW, an advanced modular backdoor leveraged by specific China-nexus threat actors GTIG has been tracking since 2022. Our exploration of ScatterBrain highlighted the intricate challenges it poses for defenders. By systematically outlining and addressing each protection mechanism, we demonstrated the significant effort required to create an effective deobfuscation solution.

Ultimately, we hope that our work provides valuable insights and practical tools for analysts and cybersecurity professionals. Our dedication to advancing methodologies and fostering collaborative innovation ensures that we remain at the forefront of combating sophisticated threats like POISONPLUG.SHADOW. Through this exhaustive examination and the introduction of our deobfuscator, we contribute to the ongoing efforts to mitigate the risks posed by highly obfuscated malware, reinforcing the resilience of cybersecurity defenses against evolving adversarial tactics.

Indicators of Compromise

A [Google Threat Intelligence Collection] featuring indicators of compromise (IOCs) related to the activity described in this post is now available.

Host-Based IOCs

MD5	Associated Malware Family

`5C62CDF97B2CAA60448619E36A5EB0B6`	POISONPLUG.SHADOW
`0009F4B9972660EEB23FF3A9DCCD8D86`	POISONPLUG.SHADOW
`EB42EF53761B118EFBC75C4D70906FE4`	POISONPLUG.SHADOW
`4BF608E852CB279E61136A895A6912A9`	POISONPLUG.SHADOW
`1F1361A67CE4396C3B9DBC198207EF52`	POISONPLUG.SHADOW
`79313BE39679F84F4FCB151A3394B8B3`	POISONPLUG.SHADOW
`704FB67DFFE4D1DCE8F22E56096893BE`	POISONPLUG.SHADOW

Acknowledgements

Special thanks to Conor Quigley and Luke Jenkins from the Google Threat Intelligence Group for their contributions to both Mandiant and Google's efforts in understanding and combating the POISONPLUG threat. We also appreciate the ongoing support and dedication of the teams at Google, whose combined efforts have been crucial in enhancing our cybersecurity defenses against sophisticated adversaries.

Posted in

- [Threat Intelligence](#)

Source: https://cloud.google.com/blog/topics/threat-intelligence/scatterbrain-unmasking-poisonplug-obfuscator