Under the hood of Wslink’s multilayered virtual machine

Author: Vladislav Hrčka

March 2022
# CONTENTS

Executive summary ................................................................. 5
  General structure of virtual machines .................................. 5
  Documented techniques for deobfuscation of virtual machines .... 11
  The Miasm framework ........................................................ 12
Wslink’s virtual machine entry – `vm_entry` ................................ 13
Packer ....................................................................................... 14
Junk code .................................................................................. 16
Virtual machine initialization .................................................... 17
  `vm_pre_init()` functions ..................................................... 17
  `vm_init()` function ............................................................ 18
Virtual instructions .................................................................... 20
  Virtual instructions of the second virtual machine ................ 24
    The first virtual instruction .............................................. 24
    The second virtual instruction ......................................... 25
    The third virtual instruction ............................................ 25
    The fourth virtual instruction ......................................... 26
    Automating analysis of the virtual instructions .................. 27
Getting back to the first virtual machine ..................................... 29
The initial virtual instruction ..................................................... 30
  Intro ...................................................................................... 31
  Outro ..................................................................................... 32
Analysis of the virtual context .................................................. 33
Behavior .................................................................................... 34
Initially executed virtual instructions ....................................... 34
  The first executed virtual instruction .................................... 34
  The second executed virtual instruction ............................... 36
  The third executed virtual instruction ................................... 36
  The fourth executed virtual instruction ............................... 38
Automating analysis of the first virtual machine ......................... 39
  Processing the initial bytecode block .................................... 40
  Opaque predicates .................................................................. 41
  Overview ................................................................................. 41
Description of our final VM analyzer code .................................. 45
  Class Wslink ......................................................................... 45
  Class VirtualContext ............................................................ 45
    Method get_next_instr() ..................................................... 45
    Method get_irb_symbs() ...................................................... 45
    Method get_updated_internal_context() ............................. 45
    Method get_state_hash() .................................................... 45
  Class MySymbolicExecutionEngine ......................................... 45
  Class SymbolicCFG ............................................................... 46
  Class Node ............................................................................. 46
Future work .............................................................................. 46
Conclusion ............................................................................... 47
LIST OF TABLES

Table 1. Miasm’s IR semantics ........................................... 13
Table 2. Statistics of the initial virtual instruction ................. 30
Table 3. Statistics of the first executed virtual instruction ....... 36
Table 4. Statistics of the second executed virtual instruction ... 36
Table 5. Statistics of the third executed virtual instruction ..... 37
Table 6. Statistics of the fourth executed virtual instruction ... 38
Table 7. Statistics of the first processed bytecode block ........ 44

LIST OF FIGURES

Figure 1. Illustration of bytecode, where all opcodes and operands are virtual ........................................... 6
Figure 2. Illustration of the relationship between bytecode and the VM’s interpreter ........................................... 7
Figure 3. Illustration of Switch Dispatch, where R0 is a virtual register ......................................................... 8
Figure 4. Illustration of Direct Threading .............................. 9
Figure 5. Overview of the virtualization process ..................... 10
Figure 6. Entry point to the virtual machine ......................... 13
Figure 7. A part of vm_entry of the virtual machine decompiled with Ghidra ............................................. 14
Figure 8. Function used to unpack NsPack in ClamAV ............... 15
Figure 9. A block of code in Miasm’s symbolic execution (left) and a part of the same block in IDA’s decompiler (right) ........... 16
Figure 10. Miasm’s symbolic execution of a vm_pre_init showing parameters supplied to vm_init() .................... 17
Figure 11. Busy-waiting for interpreter in vm_init() ................ 18
Figure 12. vm_init() summary ............................................. 19
Figure 13. Virtual instruction table ....................................... 20
Figure 14. The first virtual instruction in the table ................ 20
Figure 15. One of the vm_pre_init() functions ...................... 21
Figure 16. Miasm’s symbolic execution of the first virtual instruction (function at 0x0FF2DB) ......................... 21
Figure 17. Miasm’s symbolic execution of the first block of vm2_init() ......................................................... 22
Figure 18. Miasm’s symbolic execution of the first block of vm_init() ......................................................... 23
Figure 19. A part of the second virtual instruction table .......... 23
Figure 20. Virtual instructions in the structure of the virtual machines ......................................................... 24
Figure 21. The initial virtual instruction of the second VM ........ 25
EXECUTIVE SUMMARY

ESET researchers recently described Wslink, a unique and previously undocumented malicious loader that runs as a server and that features a virtual-machine-based obfuscator. There are no code, functionality or operational similarities that suggest this is likely to be a tool from a known threat actor; the complete analysis of the malware can be found here.

In this white paper we describe the structure of the virtual machine used in samples of Wslink and suggest a possible approach to see through the obfuscation techniques used in the analyzed samples. We demonstrate our approach on chunks of code of the protected sample. We were not motivated to fully deobfuscate the code, because we discovered a non-obfuscated sample.

Obfuscation techniques are a kind of software protection intended to make code hard to understand and hence conceal its objectives; obfuscating virtual machine techniques have become widely misused for illicit purposes such as obfuscation of malware samples as they hinder both analysis and detection. The ability to analyze malicious code and subsequently improve our detection capabilities is behind our motivation to overcome these techniques.

Virtualized Wslink samples do not contain any clear artifacts, such as specific section names, that easily link it to a known virtualization obfuscator. During our research, we were able to successfully design and implement a semiautomatic solution capable of significantly facilitating analysis of the underlying program’s code. The virtual machine introduced a diverse arsenal of obfuscation techniques, which we were able to overcome to reveal a part of the deobfuscated malicious code that we describe in this document. In the last sections of this analysis, we present parts of the code we developed to facilitate our research.

This white paper also provides an overview of the internal structure of virtual machines in general, and introduces some important terms and frameworks used in our detailed analysis of the Wslink virtual machine.

In the past we described the structure of a custom virtual machine, along with our techniques to devirtualize the machine. That virtual machine contained an interesting anti-disassembly trick, previously utilized by FinFisher – spyware with extensive spying capabilities, such as live surveillance through webcams and microphones, keylogging, and exfiltration of files. We additionally presented an approach for its deobfuscation. You can find more information about that case in this earlier white paper.

OVERVIEW OF VIRTUAL MACHINE STRUCTURES

Before diving into the analysis of Wslink’s virtual machine (VM), we provide an overview of the internal structure of virtual machines in general, describe known approaches to deal with such obfuscation and introduce some important terms and frameworks used in our detailed analysis of the Wslink VM.

General structure of virtual machines

Virtual machines can be divided into two main categories:

1. System virtual machines – support execution of complete operating systems (e.g., various VMWare products, VirtualBox)
2. Process virtual machines – execute individual programs in an OS-independent environment (e.g., Java, the .NET Common Language Runtime)

Here, we are interested only in the second category – process virtual machines – and we will briefly describe certain parts of their internal anatomy necessary to understand the rest of this paper.
Process virtual machines run as normal applications on their host OSes, and in turn run programs whose code is stored as OS-independent **bytecode** (Figure 1) that represents a series of instructions – an application – of a virtual **ISA** (instruction set architecture).

![Figure 1. Illustration of bytecode, where all opcodes and operands are virtual](image)

One can also think about bytecode as a sort of **intermediate representation (IR)**; an abstract representation of code consisting of a specific instruction set that resembles assembly more than a high-level language. It is also known as intermediate language.

The use of IR is convenient in terms of code reusability – when one needs to add support for a new architecture or CPU instruction set, it is easier to convert it to the IR instead of writing all the required algorithms again. Another benefit is that it can simplify the application of some optimization algorithms.

One can generally translate both high- and low-level languages into an IR. Translation of a higher-level language is known as “lowering”, and similarly translation of a lower-level one, “lifting”.

The following example lifts an assembly block `bb0` into a block with the pseudo-IR code `irb0`. All assembly instructions are translated into a set of IR operations and individual operations in sets do not affect each other, where `ZF` stands for zero flag and `CF` for carry flag:

**bb0:**

```
MOV R8, 0x05
SUB AX, DX
XCHG ECX, EDX
```

**irb0:**

```
R8 = 0x05

EAX[:0x10] = EAX[:0x10] - EDX[:0x10]
ZF = EAX[:0x10] - EDX[:0x10] == 0x00
CF = EAX[:0x10] < EDX[:0x10]
...

ECX = EDX
EDX = ECX
```

Modern process VMs usually provide a compiler that can lower code written in a high-level language – one that is easy to understand and comfortable to use – into the respective bytecode.

A VM’s **ISA** generally defines the supported instructions, data types and registers, among other things, that naturally must be implemented by a virtual ISA as well.

Instructions consist of the following parts:

- **opcodes** – operation codes that specify an instruction
- **operands** – parameters of the instructions
ISAs often use two well-known virtual registers:

- **virtual program counter (VPC)** – a pointer to the current position in the bytecode
- **virtual stack pointer** – a pointer to pre-allocated virtual stack space used internally by the VM

The virtual stack pointer does not have to be present in all VMs; it is common only in a certain type of VM – **stack-based ones**.

We will refer to the instructions and their respective parts of a virtual ISA simply as **virtual instructions**, **virtual opcodes**, and **virtual operands**. We sometimes omit the explicit use of “virtual” when it is obvious that we are talking about the virtual representation.

An OS-dependent (Figure 2) executable file – **interpreter** – processes the supplied bytecode and sequentially interprets the underlying virtual instructions thus executing the virtualized program.

![Diagram showing the relationship between bytecode and VM's interpreter](image)

**Figure 2.** Illustration of the relationship between bytecode and the VM's interpreter

Transfer of control from one virtual instruction to the next during interpretation needs to be performed by every VM. This process is generally known as **dispatching**. There are several documented dispatch techniques such as:
• Switch Dispatch – the simplest dispatch mechanism where virtual instructions are defined as case clauses and a virtual opcode is used as the test expression (Figure 3)
• Direct Call Threading – virtual instructions are defined as functions and virtual opcodes contain addresses of these functions
• Direct Threading – virtual instructions are defined as functions again; however, in comparison to Direct Call Threading, addresses of the functions are stored in a table and virtual opcodes represent offsets to this table. Each function should indirectly call the following one according to the specification (Figure 4)

The body of a virtual opcode in the interpreter’s code is usually called a **virtual handler** because it defines the behavior of the opcode and handles it when the virtual program counter points to a location in the bytecode that contains a virtual instruction with that opcode.

By **context**, regarding VMs, we mean a sort of virtual **process context**: each time a process is removed from access to the processor during process switching, sufficient information on its current operating state – its context – must be stored such that when it is again scheduled to run on the processor, it can resume its operation from an identical position.

**Figure 3.** Illustration of Switch Dispatch, where R0 is a virtual register
Obfuscation techniques are a kind of software protection intended to make code hard to understand and hence conceal its objectives. Such techniques were initially developed to protect the intellectual property of legitimate software, i.e., to hamper reverse engineering.

Virtual machines used as obfuscation engines are based on process virtual machines, as described above. The primary difference is that they are not intended to run cross-platform applications and they usually take machine code compiled or assembled for a known ISA, disassemble it and translate that to their own virtual ISA. It is also usually the case that the VM environment and the virtualized application code are contained in one application, whereas traditional process VMs usually consist of a process that runs as a standalone application that loads separate, virtualized applications.

The strength of this obfuscation technique resides in the fact that the ISA of the VM is unknown to any prospective reverse engineer – a thorough analysis of the VM, which can be very time-consuming, is required to understand the meaning of the virtual instructions and other structures of the VM. Further, if performance is not an issue, the VM’s ISA can be designed to be arbitrarily complex, slowing its execution of virtualized applications, but making reverse engineering even more complex.

Understanding of the VM is necessary for decoding the bytecode and making the virtualized code understandable.

Context has a bit of a different meaning in regard to obfuscating virtual machines: each time we want to switch from the native to virtual ISA or vice-versa, sufficient information – context – on the current operating state must be stored so that when the ISA has to be switched back, execution can resume with only the relevant data and registers modified.

Additionally, obfuscating VMs usually virtualize only certain “interesting” functions – native context is mapped to the virtual one and bytecode, representing the respective function, is chosen beforehand. The built-in interpreter is invoked afterwards (Figure 5). Beginnings of the original functions contain code that prepares and executes the interpreter – entry of the VM (vm_entry); the rest of their code is omitted in Figure 5.

Interpreter, bytecode, and virtual ISA code with data of obfuscating VMs are often all stored in a dedicated section of the executable binary, along with the rest of the partially virtualized program.
Figure 5 shows the way a function, Function 1, in the original application targeting a common ISA can be virtualized for an obfuscating VM's ISA. It needs to be converted into bytecode, for example using a generate_bytecode method. Its body is afterwards overwritten by a call into vm_entry and zeroes. The vm_entry function chooses the respective bytecode, for example, based on the calling function’s address, then conducts a context switch, and next interprets the bytecode. Finally, it returns to the code where the virtualized function, Function 1, would return.

In VMs hosted on x86 architectures, such context switches usually consist of a series of PUSH and POP instructions. For example:

PUSH EAX
PUSH EBX
PUSH ECX
...
MOV ECX, context_addr
POP DWORD PTR [ECX]
POP DWORD PTR [ECX + 4]
POP DWORD PTR [ECX + 8]
...

When the bytecode is fully processed, virtual context is mapped back to native context and execution continues in the non-virtualized code; however, another virtualized function could be executed in the same manner, right away.

Note that several context switches can occur in one virtualized function, for example when a native instruction from the original ISA could not be translated to virtual instructions or an unknown function from the native API needs to be executed.
Documented techniques for deobfuscation of virtual machines

Obfuscating VM techniques have become widely misused for illicit purposes such as obfuscation of malware samples as they hinder both analysis and detection. Hence there is motivation to overcome these obfuscation techniques so as to facilitate analysis of such malicious code and to achieve overall improvement of detection methods.

But first, we want to clarify several terms that are used in this and following sections and might not be known to all readers.

Symbolic execution is a code analysis technique, where specific variables are represented with symbolic values instead of concrete data. Arbitrary operations with these symbolic values produce symbolic expressions. It is usually applied on the code’s IR and the symbolic expressions include flags.

One can visualize the symbolic expressions like mathematical formulas as can be seen in the following example, where `irb1` contains a block of pseudo-IR:

```
irb1:
R13 = R13 + 0x027D3930
RBX = RCX + 0x05
R13 = R13 + -RSI
R13 = R13 + RBX

irb1_symb:
RBX = RCX + 0x05
R13 = R13 + RCX + 0x05 + -RSI + 0x027D3930
ZF = R13 + RCX + 0x05 + -RSI + 0x027D3930 == 0x00
```

The state of symbolically executed code consists of:
- Values of all variables
- Program counter
- Accumulated constraints that the program’s inputs need to satisfy to reach the associated location from the entry point

Accumulated constraints can be understood as a theory in logic. In order to find concrete values of the initial variables with symbolic values – inputs – we need to find a satisfying model, which can be done with an SMT (satisfiability modulo theories) solver.

Path coverage is another code analysis technique that determines all possible paths in a piece of code. It is usually implemented using symbolic execution instructed to explore all reachable paths – reachability of newly discovered paths is verified by an SMT solver and already known paths are marked to prevent infinite loops.

Microsoft describes program synthesis as “the task of automatically discovering an executable piece of code given user intent expressed using various forms of constraints such as input-output examples, demonstrations, natural language, etc.”.

Several techniques to deal with VM-based obfuscation have been proposed in the past. Here we briefly walk through them and discuss their advantages and disadvantages.
Rolf Rolles described several standard steps to manually recover the original code, where the drawback is time-complexity:

1. Reverse engineer and understand structures of the VM
2. Detect entries into the VM
3. Develop a disassembler for the instruction set by identifying the purpose of individual virtual opcodes or matching them against already known ones
4. Disassemble the bytecode and convert it into intermediate representation – the semantics of some instructions might be hard to comprehend in basic blocks without further translation (e.g., stack-based VMs would contain a lot of confusing PUSH and POP machinations)
5. Apply compiler optimizations to get rid of additional obfuscation techniques
6. Generate the deobfuscated code

He additionally suggested the use of pure symbolic execution on the virtual opcodes in the fourth step to obtain a representation, where each opcode is a mathematical function that is a map from its input space into itself. The pure symbolic execution technique was later independently implemented in a Miasm blogpost.

Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet proposed a fully automatic approach to overcome obfuscating VM protection on samples with a finite number of executable paths. The approach consists of the following steps:

1. Identification of the sample’s inputs
2. Isolation of pertinent instructions dependent on the identified inputs on an execution trace
3. Performance of a path coverage analysis to reach new paths – traces
4. Reconstruction of the original program from the resulting traces – they are combined and compiler optimizations partially recover the control flow graph

Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Hol produced a semiautomatic approach, based on program synthesis, that uses instruction traces as a black-box oracle to produce random input and output pairs. The I/O pairs are subsequently used to learn the code’s underlying semantics with the synthesizer.

These pairs and semantics are generated for the virtual opcodes that must be identified beforehand – the VM needs to be partially reverse engineered to locate its components.

The approach does not seem to be applicable to some complex (particularly obfuscating) VMs due to its time complexity, as it reportedly took almost three hours to process 36 virtual opcodes of a VM – duplication of handlers, which is a simple and common obfuscation technique, would be a huge issue.

The Miasm framework

Miasm is a free and open-source reverse-engineering framework that aims to analyze, modify and generate binary programs. It has a number of useful features that we use throughout our analysis:

- Opening, modifying and generating binary files – PE and ELF
- Assembling and disassembling of various architectures such as x86, ARM, MIPS...
- Representing assembly semantics using intermediate representation
- Simplification rules for automatic deobfuscation
- Symbolic execution engine
- ...

There are several frameworks for reverse-engineering that provide the features that we needed; we decided to use Miasm in this project simply because it is actively maintained, and we are already familiar and satisfied with it.
The features that we want to use are covered in the example section of its GitHub repository description and its documentation.

We encourage the reader to get familiar at least with semantics of its IR that are summarized in Table 1, since they are going to be used repeatedly.

<table>
<thead>
<tr>
<th>Element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExprId</td>
<td>EAX</td>
</tr>
<tr>
<td>ExprAssign</td>
<td>A=B</td>
</tr>
<tr>
<td>ExprInt</td>
<td>0x18</td>
</tr>
<tr>
<td>ExprLoc</td>
<td>location_1</td>
</tr>
<tr>
<td>ExprCond</td>
<td>A ? B : C</td>
</tr>
<tr>
<td>ExprMem</td>
<td>@16[ESI]</td>
</tr>
<tr>
<td>ExprOp</td>
<td>A + B</td>
</tr>
<tr>
<td>ExprSlice</td>
<td>AH = EAX[8:16]</td>
</tr>
<tr>
<td>ExprCompose</td>
<td>{EAX 0 32, 0X 32 64}</td>
</tr>
</tbody>
</table>

Table 1. Miasm’s IR semantics

The destination address of a symbolic execution performed over a block of code is saved in the respective program counter such as RIP and additionally in a special variable IRDst.

Note that during Miasm’s symbolic execution: initial values of registers, which are treated as variables, are symbolic and their format is <register name>_init. Simplification rules are applied automatically to the symbolic expressions. For example, the symbolic expression RAX = RCX + 0x2 + 0x3 is automatically simplified into RAX = RCX_init + 0x5.

**WSLINK’S VIRTUAL MACHINE ENTRY – VM_ENTRY**

Let’s get to the analysis of Wslink’s VM now. There are several function calls that enter the VM, all of which are followed by some gibberish data that IDA attempts to disassemble – the data most likely just overwrites the function’s original code before virtualization (Figure 6).
The `vm_entry` of the VM:

- calculates the actual base address by subtracting the expected relative virtual address from the actual virtual address of a place in the code
- unpacks code and data related to the VM on the first run; it uses the calculated base address to determine the location of the packed VM and destination of the unpacked data
- executes an initialization function – one of the `vm_pre_init()` functions to be described is based on the caller’s relative address that is mapped to the respective `vm_pre_init()`

**PACKER**

Wslink’s VM is packed with `NsPack` to reduce the size of the huge executable file; additional obfuscation is probably just a side effect. Similarities between Wslink’s unpacking code and ClamAV’s `unspack()` function are clearly visible (Figure 7 and Figure 8). Note that Ghidra has optimized out calculation of the base address.

```c
vm_pre_init_dispatch = &vm_pre_init_dispatch_table;
if (is_packed != 0) {
    prepare_in_cgi_params();
    if ((in_R8 < 'x02') && (0x0d < (uint)in_P9)) {
        c = Sext1d(in_R8[1], (int)c < 0x11)
        firstbyte = 0;
        allocsz = 0;
        if (8 < (int)c) {
            allocsz = c / 0x2d;
            c = c % 0x2d;
        }
        very_real_unpack((in_R8 + in_P9, 0x300 << 1) | allocsz + (char)c & 0x1fU) + 2 + 0x68c0, c,
        allocsz, firstbyte, in_R8 + 0x1e, (uint)in_P9 - 0xe, in_RCX, in_EDX;
        *in_RDX = ret_addr;
        wVar1 = 0;
    } else {
        wVar1 = 0xffffffff;
    }
    return wVar1;
}
```

Figure 7: A part of `vm_entry` of the virtual machine decompiled with Ghidra
The `vm_pre_init_dispatch_table` in Figure 7 is the structure that maps callers’ addresses of the `vm_entry` to the respective `vm_pre_init()` functions that are to be described.

```c
def function_used_to_unpack_NsPack_in_ClamAV()

    if (c==0x1)
        return 1;
    if (c==0x2d)
        firstbyte = c/0x2d;
    do {c+=0x30;} while (--i);
    else firstbyte = 0;
    if (c==0)
        allocsz = i = c/9;
    do {c+=0xff;} while (--i);
    else allocsz = c;
    tre = c;
    i = allocsz;
    c = (tre+i)&0xff;
    tablesz = (((0x300<<2)+0x730)*sizeof(uint16_t));
    if (cli_checklimits("nspack", ctx, tablesz, 0, VCL_CLEAN))
        return 1; /* Should be ~15KB, if it's so big, it's prolly just not nspacked */
    clidbgmsg("unsp: table size = %d \n", tablesz);
    if (!table = cli_malloc(tablesz))
        clidbgmsg("unspack: Unable to allocate memory for table\n");
    return 1;
}

dsize = cli_readint32(start_of_stuff+8);
ssize = cli_readint32(start_of_stuff+5);
if (ssize <= 15) {
    free(table);
    return 1;
}

tre = very_real_unpack(table, tablesz, tre, allocsz, firstbyte, src, ssize, dst, dsize);
```

**Figure 8.** Function used to unpack NsPack in ClamAV

The `vm_pre_init_dispatch_table` in Figure 7 is the structure that maps callers’ addresses of the `vm_entry` to the respective `vm_pre_init()` functions that are to be described.
JUNK CODE

Each part of the unpacked VM is obfuscated with lots of junk code – unnecessary additional instructions significantly decreasing readability of the code. It often uses instruction pairs with opposite effects.

Neither the IDA nor the Ghidra decompiler is able to deal with such obfuscation; however, Miasm’s symbolic execution was able to make the code easily readable (Figure 9).

Figure 9. A block of code in Miasm’s symbolic execution (left) and a part of the same block in IDA’s decompiler (right)
VIRTUAL MACHINE INITIALIZATION

Initialization of the VM consists of several steps, such as saving values of the native registers on the stack and later moving them to the virtual context, relocation of its internal structures, or preparation of bytecode. We cover these steps more thoroughly in the following subsections.

**vm_pre_init() functions**

`vm_pre_init()` functions are meant only to prepare parameters for another stage of initialization (Figure 10). These functions call a single `vm_init()` function (explained in the next section) with specific parameters. The supplied parameters are:

- CPU flags, `RFLAGS`, which are stored on the stack with a `PUSHF` instruction at the beginning of each function
- hardcoded offset to a virtual instruction table that represents the first virtual instruction to be executed (its opcode)
- hardcoded address of the bytecode to be interpreted

![Figure 10: Miasm's symbolic execution of a `vm_pre_init()` showing parameters supplied to `vm_init()`](image)

Under the hood of Wslink's multilayered virtual machine
vm_init() function

vm_init() pushes all the native registers and the supplied CPU flags from parameters (context) onto the stack; one can actually see it in Figure 9. The native context will later be moved to the virtual one that, in addition, holds several internal registers.

One of the internal registers determines whether another instance of the VM is already running – there is only one global virtual context and only one instance of the VM can run at a time. Figure 11 shows the part of the code busy-waiting for the virtual register, where RBP contains the address of the virtual context and RBX the offset of the virtual register – the internal register is stored in [RBX + RBP].

The entire function is summarized in Figure 12.

The bytecode’s address, supplied in the parameters, is added to the virtual context along with the address of the virtual instruction table, which is hardcoded. Both have a dedicated virtual register.

The VM calculates the base address again in the same way as was described for vm_entry; in addition, it stores the address in another internal register that is used later, should an API be called. Then the base address is used to relocate the instruction table, its entries, and the bytecode’s address.

The calculated base address is simply added to all the function addresses if they have not already been relocated.
Figure 12. \texttt{vm\_init()} summary
VIRTUAL INSTRUCTIONS

There are only 45 instructions in the virtual instruction table (Figure 13).

Figure 13. Virtual instruction table

Let us look at the first one in the table. Initially, we need to relocate it; our dump of the VM starts at address 0x00 and it is expected to be at base + 0x0F33F5, so the target address is 0x1EC74E – 0x0F33F5, which is 0x0F9359 (Figure 14).

Figure 14. The first virtual instruction in the table

The JMP in Figure 14 leads us to a function at 0x0FF2DB whose behavior is remarkably similar to vm_pre_init() (Figure 15 and Figure 16 for comparison). The function appears to be pushing another bytecode address, the opcode of the initial virtual instruction, and CPU flags.
Inspecting the function at 0x0F7FFF (Figure 17), into which our virtual instruction jumps, reveals that it appears to be another `vm_init()` (Figure 18). When we compare it to the previous one, we can see that their behaviors are, indeed, the same. We will refer to these functions simply as `vm2_pre_init()` and `vm2_init()`.
Symbolic Execution - 0xf7fff to 0xf888b

RAX = call_func_ret(0xF8004, RSP_init, RCX_init, RDX_init, R8_init, R9_init)
RBX = 0xFF
RCX = 0x1
RSP = call_func_stack(0xF8004, RSP_init) + 0xFFFFFFFFFFFFF80
RBP = @64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFF07FFC
zf = call_func_stack(0xF8004, RSP_init) == 0x80
nf = (call_func_stack(0xF8004, RSP_init) + 0xFFFFFFFFFFFFF80)[63:64]
pf = parity((call_func_stack(0xF8004, RSP_init) + 0xFFFFFFFFFFFFF80) & 0xFF)
of = (((call_func_stack(0xF8004, RSP_init) + 0xFFFFFFFFFFFFF78) ^ (call_func_s)
cf = (((call_func_stack(0xF8004, RSP_init) + 0xFFFFFFFFFFFFF78) ^ (call_func_s
af = (((call_func_stack(0xF8004, RSP_init) + 0xFFFFFFFFFFFFF78) ^ (call_func_s
IRDst = loc_key_3
@64[call_func_stack(0xF8004, RSP_init)] = call_func_ret(0xF8004, RSP_init, RCX
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFF60] = @64[call_func_s
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFF68] = RDI_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFF70] = RSI_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFF78] = RSI_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFF80] = @64[call_func_s
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFF88] = R8_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFF90] = R9_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFF98] = R10_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFA0] = R11_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFA8] = R12_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFB0] = R13_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFB8] = R14_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFC0] = R15_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFC8] = RDI_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFD0] = RSI_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFD8] = RBP_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFE0] = RBX_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFFE8] = RBX_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFFF0] = RDX_init
@64[call_func_stack(0xF8004, RSP_init)] + 0xFFFFFFFFFFFFFFF8] = RCX_init

Figure 17. MiSm's symbolic execution of the first block of vm2_init()
Inspection of the other instructions revealed that they all execute this second VM with different `vm2_pre_init()` functions - this clearly shows that there are two layers of VMs.

Virtual instructions of the first VM execute `vm2_pre_init()` directly without any dispatch table based on the caller's address. The number of virtual instructions in the second VM is significantly higher - 1071 (Figure 19).

```
F5E7  d0 0A17D5h, 0A14Ch, 0A1CFDh, 0A214Dh, 0A2573h, 0A292Eh
F5E7  d0 0A2C1Ch, 0A30F6h, 0A35Ah, 0A3827h, 0A3F7h, 0A3F8h
F5E7  d0 0A43B1h, 0A4772h, 0A4A54h, 0A4E8Dh, 0A5607h, 0A54E7h
F5E7  d0 0A5656h, 0A558Dh, 0A5634h, 0A5676h, 0A52D5h, 0A500Bh
F5E7  d0 0A63A4h, 0A605Ah, 0A60ECh, 0A645Ah, 0A64E9h, 0A61E5h
F5E7  d0 0A81F0h, 0A85A5h, 0A85CDh, 0A84A1h, 0A8691h, 0A8194h
F5E7  d0 0A850Ah, 0A44C8h, 0A45F8h, 0A4635h, 0A4B37h, 0A4A44h
F5E7  d0 0A5B98h, 0A5A82h, 0A5854h, 0A57B6h, 0A50B1h, 0A5068h
F5E7  d0 0A6222h, 0A43C3h, 0A4C7Eh, 0A4C51h, 0A4CF2h, 0A42D9h
F5E7  d0 0A6D2Fh, 0A6D19h, 0A6D6Dh, 0A6E21h, 0A6E08h, 0A6D8Dh
F5E7  d0 0AED55h, 0AF0A6h, 0AF585h, 0AF921h, 0AF40h, 0BF04Ch
F5E7  d0 0B0A1Eh, 0B9725h, 0B0A09h, 0B06CCh, 0B164Ah, 0B153Fh
F5E7  d0 0B1949h, 0B1D12h, 0B20C5h, 0B259Bh, 0B28D0h, 0B2C59h
F5E7  d0 0B3174h, 0B43F9h, 0B44ACh, 0B414Ah, 0B4010h, 0B4359h
F5E7  d0 0B4E9h, 0B4795h, 0B4614h, 0B4D91h, 0B4511h, 0B455Ah
F5E7  d0 0B5E7Eh, 0B5AF6h, 0B5DECh, 0B61E2h, 0B65E1h, 0B6AF8h
F5E7  d0 0B757Eh, 0B7876h, 0B7C07h, 0B7F94h, 0B8332h, 0B8546h
F5E7  d0 0B8544h, 0B8039h, 0B863Dh, 0B8430h, 0B85E6h, 0B84C6h
F5E7  d0 0B8F14h, 0B4A44h, 0B4A5Ch, 0B6A40h, 0B6323h, 0B6323h
F5E7  d0 0B8D7Dh, 0B89E3h, 0B8CF1h, 0B8CC1h, 0B8C7Dh, 0B8D7Dh
F5E7  d0 0B8D5Fh, 0B8D96h, 0B8D6Fh, 0B8DA7h, 0B8D9Fh, 0B82C4h
F5E7  d0 0B8706h, 0B8E08h, 0B8EB2h, 0B8EF4h, 0B8F2Dh, 0B8F6E7h
F5E7  d0 0B8F9Eh, 0B8F92h, 0C6100h, 0C6052h, 0C62AAh, 0C68E3h
F5E7  d0 0C6122h, 0C6179h, 0C1E87h, 0C1E27h, 0C208Ch, 0C23B4h
F5E7  d0 0C2C5Ch, 0C2A32h, 0C2D0Fh, 0C271Ch, 0C354Ah, 0C35DFh
F5E7  d0 0C3D75h, 0C3A80h, 0C45FBh, 0C45C8h, 0C514Dh
F5E7  d0 0C5C6Dh, 0C5C8Dh, 0C5E2Ch, 0C61Ch, 0C65F0h, 0C6A31h
F5E7  d0 0C60B3h, 0C7185h, 0C74A7h, 0C75F7h, 0C7A4Ch, 0C7C8Ch
F5E7  d0 0C8863h, 0C82A7h, 0C85EEh, 0C8880h, 0C8616h, 0C9064h
F5E7  d0 0C9384h, 0C9A86h, 0C9E53h, 0C9A32h, 0C944Ah, 0C9A85h
```

Figure 18. Miasm's symbolic execution of the first block of vm_init().

Figure 19. A part of the second virtual instruction table.
Virtual instructions of the second virtual machine

We start by looking at the first few executed virtual instructions to observe the behavior of the second VM and then try to process the rest of them in a partially automated way.

The diagram in Figure 20 highlights with blue, where the virtual instructions of the second VM are in the structure of the VMs.

![Diagram](Image)

**Figure 20.** Virtual instructions in the structure of the virtual machines

The first virtual instruction

The first virtual instruction is, exceptionally, not obfuscated, as can be seen in Figure 21. Finally, we can see some operations in the virtual context.

By inspecting the modified memory and calculated destination address of the instruction, it is clear that the instruction does three things:

1. Zeroes out a virtual 32-bit register at offset \(0xB5\) in the virtual context (highlighted in gray in Figure 21), which is stored in the \(RBP\) register.
2. A virtual 64-bit register at offset \(0x28\) is increased by \(0x04\): it is the pointer to the bytecode – virtual program counter. The size of the virtual instruction is hence four bytes (highlighted in red in Figure 21).
3. The next virtual instruction is prepared to be executed, the offset to the virtual instruction table – virtual opcode – is fetched from the virtual program counter. The virtual instruction table is at offset \(0xA4\) (highlighted in green in Figure 21). This means that the VM uses the Direct Threading Dispatch technique.
Note that the size of the next instruction’s opcode is only two bytes and the remaining word is left unused. We can see that it is just a zero when we look at virtual operands (Figure 22). Sizes of the other instructions differ – it is not just padding that preserves the same size for all instructions.

The second virtual instruction does not do anything special; it just zeroes out several virtual registers and jumps to the next instruction (Figure 23).

The third virtual instruction stores the address of the stack pointer in a virtual register (Figure 24); the offset of the register is determined by one of the operands, and its offset is 0x0141 in our case.
The fourth virtual instruction

The fourth instruction contains two immediately visible anomalies in comparison with previous instructions – the stack pointer’s delta is lower at the end of the function and it contains a conditional branch (Figure 25).

Symbolic execution of the first block reveals that a value is popped from the stack into a virtual register (Figure 26), which makes sense as the values of the native registers remain on the stack after being saved there by `vm2_init()`. They are now being moved to the virtual context – the context switch is partially performed by a number of virtual instructions, each of which pops one value off the stack into a different register.

The virtual register, where the value of the native register is to be saved, is determined by an operand and two other virtual registers at offsets `0x0B` and `0x70`. However, their initial value is already known: they were set to zero by the second virtual instruction (Figure 23), which means that we can calculate the offset of the register and simplify the expressions – they are used just to obfuscate the code.

Figure 25. The conditional branch and delta of the stack pointer of the fourth virtual instruction

Figure 26. Destination address and memory modified by the fourth virtual instruction
Rolling decryption

Analysis of other virtual instructions confirmed that the virtual registers at offsets 0x0B and 0x70 are meant just to encode operands. This technique is called rolling decryption and it is known to be used by the VMProtect obfuscator. However, it is the only overlap with that obfuscator and we are highly confident that this VM is different.

The obfuscation technique is certainly one of the reasons for the enormous number of virtual instructions – use of the technique requires duplication of individual instructions since each uses a different key to decode the operands.

Simplification

The expressions can be simplified to the following when we apply the known values of the virtual registers:

\[
\text{IRDst} = (-@16[@64[RBP_{init} + 0x28] + 0x4] ^ 0x3038 == @16[@64[RBP_{init} + 0x28] + 0x6])?(0x7FEC91ABD1C,0x7FEC91ABC6)\\
@64[RBP_{init} + {@16[@64[RBP_{init} + 0x28] + 0x6], 0, 16, 0x0, 16, 64}] = @64[RBP_{init}]
\]

Now let us take a look at the expression in the conditional block:

\[
@64[RBP_{init} + {@16[@64[RBP_{init} + 0x28] + 0x6], 0, 16, 0x0, 16, 64}] = @64[RBP_{init} + {@16[@64[RBP_{init} + 0x28] + 0x6], 0, 16, 0x0, 16, 64}] + 0x8
\]

We can now see that the virtual instruction is definitely POP – it moves a value off the top of the stack to a virtual register, whose offset is still obfuscated with a simple XOR; it additionally increases the stack pointer when the destination register is not the stack pointer.

As values in the bytecode are known too, we can apply them and simplify the instruction even further into the following final unconditional expressions:

\[
\text{IRDst} = @64[@64[RBP_{init} + 0xA4] + 0x5A8]\\
@64[RBP_{init} + 0x28] = @64[RBP_{init} + 0x28] + 0x8\\
@64[RBP_{init} + 0x141] = @64[RBP_{init} + 0x141] + 0x8\\
@64[RBP_{init} + 0x12A] = @64[RSP_{init}]
\]

Automating analysis of the virtual instructions

As doing this for more than 1000 instructions would be very time consuming, we wrote a Python script with Miasm that collects this information for us so we can get a better overview of what is going on. We are particularly interested in modified memory and destination addresses.

Just as in the fourth virtual instruction, we will treat certain virtual registers as concrete values to retrieve clear expressions. These registers are dedicated to the rolling decryption and perform memory accesses that are relative to the bytecode pointer, e.g. [\text{oobf}\_\text{reg}_1\text{]} = [(\text{obfuscator} + 0x05) ^ 0xABCD.

Subsequently we concretize the pointer to the virtual instruction table too and, by the end of the virtual instruction: calculate addresses of the next ones, clear the symbolic state, and start with the following virtual instructions.

We additionally save aside memory assignments that are not related to the internal registers of the VM and gradually build a graph based on the virtual program counter (Figure 27).
We stop when we cannot unambiguously determine the next virtual instructions to be executed; one can automatically process most of the virtual instructions in this way.

Note that instructions featuring complex loops cannot be processed with certainty and need to be addressed individually due to the path explosion problem of symbolic execution, which is described for example in the paper *Demand-Driven Compositional Symbolic Execution*: “Systematically executing symbolically all feasible program paths does not scale to large programs. Indeed, the number of feasible paths can be exponential in the program size, or even infinite in presence of loops with unbounded number of iterations.”
Getting back to the first virtual machine

Before diving into the virtual instructions of the first VM, let us recap where we currently are. We have just described a way to semiautomate processing of the bytecode belonging to the second VM (blue in Figure 28) that interprets virtual instructions of the first VM (red in Figure 28). Now we move on to inspect instructions of the first VM with this approach.

Figure 28. Virtual instructions in the structure of the virtual machines
The initial virtual instruction

In this section we describe the results of processing of the initial virtual instruction of the first VM in the semiautomatic manner that was described in the previous section.

We performed all the processing on a virtual machine with i7-4770 CPU and 4GB of memory. Statistics in Table 2 have been extracted from the processing of the initial virtual instruction.

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the bytecode block in bytes</td>
<td>1,145</td>
</tr>
<tr>
<td>Total number of processed virtual instructions</td>
<td>109</td>
</tr>
<tr>
<td>Total number of underlying native instructions</td>
<td>17,406</td>
</tr>
<tr>
<td>Total number of resulting IR instructions (including IRDest)</td>
<td>307</td>
</tr>
<tr>
<td>Execution time in seconds</td>
<td>10.6509</td>
</tr>
</tbody>
</table>

Table 2. Statistics of the initial virtual instruction

The resulting control flow graph built out of the semantics extracted from the virtual instructions of the second VM’s bytecode that interprets the initial virtual instruction from the first VM can be seen in Figure 29. We can divide the series into a few parts.

Figure 29. Control flow graph of the initial virtual instruction
Intro

As expected, the graph starts with a series of POP instructions that move values of the native registers saved beforehand in \texttt{vm2\_init()} to the virtual ones (Figure 30). To determine positions of the native registers on the stack, we could symbolically evaluate the first block of \texttt{vm2\_init()} and map the virtual registers to their native versions, which would make the code easier to read, but that is not important now.

Remember that the virtual register at offset \texttt{0x1E} contains the stack pointer, and that a POP instruction moves a value off the top of the stack and usually increases the stack pointer.

![Diagram](image)

Figure 30. Beginning of the intro finishing context switch of the second VM
Outro

To map the virtual registers back to the native ones, the second VM pushes them all onto the stack and then subsequently pops them off one by one to the native ones. Note that we set up an exclusion in our algorithm and disabled optimizations to show assignments to registers in the last virtual instruction (Figure 31).

$R10 = @64[RSP\_init + 0x10]$
$R11 = @64[RSP\_init + 0x18]$
$R12 = @64[RSP\_init + 0x20]$
$R13 = @64[RSP\_init + 0x28]$
$R14 = @64[RSP\_init + 0x30]$
$R15 = @64[RSP\_init + 0x38]$
$zf = @32[RSP\_init + 0x78][6:7]$
$nf = @32[RSP\_init + 0x78][7:8]$
$pf = @32[RSP\_init + 0x78][2:3]$
$cf = @32[RSP\_init + 0x78][11:12]$
$af = @32[RSP\_init + 0x78][4:5]$
$df = @32[RSP\_init + 0x78][10:11]$
$tf = @32[RSP\_init + 0x78][8:9]$
$i\_f = @32[RSP\_init + 0x78][9:10]$
$iopl\_f = @32[RSP\_init + 0x78][12:14]$
$nt = @32[RSP\_init + 0x78][14:15]$
$rf = @32[RSP\_init + 0x78][16:17]$
$vm = @32[RSP\_init + 0x78][17:18]$
$ac = @32[RSP\_init + 0x78][18:19]$
$vif = @32[RSP\_init + 0x78][19:20]$
$vip = @32[RSP\_init + 0x78][20:21]$
$id = @32[RSP\_init + 0x78][21:22]$

$exception\_flags = @32[RSP\_init + 0x78][8:9]$$(0x2,\text{exception\_flags\_init})$

$IRDest = @64[RBP\_init + 0x74]$
$@32[RBP\_init + 0xFF] = 0x0$

**Figure 31.** Virtual registers of the second machine being mapped back to the native ones at the end of the virtual instruction
Analysis of the virtual context

In this section we analyze the behavior of the first VM based on the results of the The first virtual instruction section.

Figure 32 shows:

- virtual registers being pushed onto the stack at the beginning of the outro (red)
- partially the way the next virtual instruction is prepared to be executed (green)
- the virtual program counter being increased (blue)

In particular, the virtual program counter is represented by \@$64[@64[RBP\_init + 0x38] + 0x2C]$\], where the register at \@$64[RBP\_init + 0x38]$\] holds the address of the virtual context. We can see that size of the initial virtual instruction was 8 bytes, since the virtual program counter is increased by 8 in the line highlighted with blue in Figure 32.

As one can see in Figure 31 (IRDST = \@$64[RBP\_init + 0x74]$\)), the virtual register at offset 0x74 determines IRDST – the address of the next instruction. If we follow the virtual register \@$64[RBP\_init + 0x74]$\] in Figure 32, we can see that it appears to be preparing to execute the next virtual instruction similarly to the second VM. Its code slice is the following series of expressions:

\begin{verbatim}
#64[RBP\_init + 0x137] = #64[RBP\_init + 0x38] + 0x26
#32[#64[RBP\_init + 0x38] + 0x26] = #32[RBP\_init + 0x30] | #32[#64[RBP\_init + 0x38] + 0x26]
#64[RBP\_init + 0x50] = (#64[RBP\_init + 0x30] & $00FFFF00$)(&(0x2 0 2, parity(#64[RBP\_init + 0x30] & $00FFFF00$))
#64[RBP\_init + 0x30] = #64[RBP\_init + 0x30] & $00FFFF00$
#64[RBP\_init + 0x50] = (#64[RBP\_init + 0x30] << 0x3)7(#64[RBP\_init + 0x30][61:62] 0 1 0 0)
#64[RBP\_init + 0x30] = #64[RBP\_init + 0x30] << 0x3
#64[RBP\_init + 0xDE] = #64[RBP\_init + 0x30] + #64[RBP\_init + 0xDE]
#64[RBP\_init + 0x50] = (#64[RBP\_init + 0x30] + #64[RBP\_init + 0xDE]) & (#64[RBP\_init + 0x30] + #64[RBP\_init + 0xDE])
#64[RBP\_init + 0x74] = #64[RBP\_init + 0x30]\
#64[RBP\_init + 0x30] = #64[RBP\_init + 0x30] + 0x2
#64[RBP\_init + 0x30] = \{#16[#64[RBP\_init + 0x30]] 0 16, 0x0 16 64\}
#64[RBP\_init + 0x38] = #64[RBP\_init + 0x38] + 0x2C
#64[RBP\_init + 0x50] = #64[RBP\_init + 0x50] + 0x2C
#64[RBP\_init + 0x74] = #64[RBP\_init + 0x74]
#64[RBP\_init + 0x30] = #64[RBP\_init + 0x30] + 0x2C
\end{verbatim}

Figure 32. Last few virtual instructions executed before mapping the virtual registers back to the native ones.
The entire slice of @64[RBP_init + 0x30] is meant just to acquire the offset of the next virtual instruction (opcode): it gets the virtual instruction’s offset from the bytecode whose pointer is stored in the @64[@64[RBP_init + 0x38] + 0xEE] register, and the offset is subsequently increased by 0x8E839329... which could have been omitted and serves solely to obscure the virtual instruction.

The virtual register @64[@64[RBP_init + 0x38] + 0xEE] contains the address of the virtual instruction table. Now it is clear that the first VM is obfuscated using known values from the bytecode too and that the code indeed executes a next virtual instruction as well – it definitely uses Direct Threading.

One can additionally see that @64[RBP_init + 0x50] stores the RFLAGS in Figure 32.

Behavior

The virtual instruction behaves similarly to the virtual instructions from the second VM – offsets of the virtual registers to be used are fetched from the virtual instruction’s operands.

Subsequently a virtual register’s value is moved to a memory address stored in another one: [<virt_reg_1>] = <virt_reg_2>. The target register is then either increased or decreased by 8: <virt_reg_1> = <virt_reg_1> +- 8. This is most likely a PUSH instruction prepared also for environments where the stack grows upwards.

Initially executed virtual instructions

We will have a look at a few other virtual instructions to confirm our findings and the correctness of methods for analysis of the first VM. Specifically, the virtual instructions that are initially executed as we can compare the first VM’s initial behavior to the second VM’s.

The first executed virtual instruction

We can see in the highlighted line of Figure 33 that the first executed instruction of the first VM behaves indeed just like the one in the second VM – it just zeroes out an internal register and prepares another virtual instruction to be executed.
Figure 33. Zeroing out an internal register

```assembly
; Zeroing out an internal register

@64[RBP_init + 0x141] = RSP_init + 0x98

@64[RBP_init + 0x88] = @64[RBP_init + 0x38]

@64[RBP_init + 0x88] = @64[RBP_init + 0x38] + 0x47

@64[RBP_init + 0x50] = ((@64[RBP_init + 0x38] ^ (@64[RBP_init + 0x38] + 0x2C)) + 0x47)

@32[(@64[RBP_init + 0x38] + 0x47) = 0x0]

@64[RBP_init + 0x98] = 0x0

@64[RBP_init + 0x30] = @64[RBP_init + 0x38]

@64[RBP_init + 0x50] = ((@64[RBP_init + 0x38] ^ (@64[RBP_init + 0x38] + 0x2C)) + 0x47)

@64[RBP_init + 0x30] = @64[RBP_init + 0x38] + 0x2C

@64[RBP_init + 0x50] = ((@64[RBP_init + 0x38] ^ (@64[RBP_init + 0x38] + 0x2C)) + 0x2C)

@64[RBP_init + 0x98] = @64[RBP_init + 0x30]

@64[RBP_init + 0x98] = @64[RBP_init + 0x98] << 0x3

@64[RBP_init + 0x50] = (@64[RBP_init + 0x98] << 0x3) ? (@64[RBP_init + 0x98] + 0x47)

@64[RBP_init + 0x30] = @64[RBP_init + 0x38]

@64[RBP_init + 0x74] = @64[RBP_init + 0x38]

@64[RBP_init + 0x74] = @64[RBP_init + 0x38] + 0x3F

@64[RBP_init + 0x50] = ((@64[RBP_init + 0x38] ^ (@64[RBP_init + 0x38] + 0x62)) + 0x3F)

@64[RBP_init + 0x98] = @64[RBP_init + 0x38]

@64[RBP_init + 0x98] = @64[RBP_init + 0x38] + 0x3F

@64[RBP_init + 0x98] = @64[RBP_init + 0x38] + 0x92

@64[RBP_init + 0x50] = ((@64[RBP_init + 0x38] ^ (@64[RBP_init + 0x38] + 0x62)) + 0x92)

@64[RBP_init + 0x30] = @64[RBP_init + 0x38]

@64[RBP_init + 0x50] = ((@64[RBP_init + 0x38] ^ (@64[RBP_init + 0x38] + 0x62)) + 0x92)

@64[RBP_init + 0x50] = ((@64[RBP_init + 0x38] ^ (@64[RBP_init + 0x38] + 0x62)) + 0x92)

@64[RBP_init + 0x50] = ((@64[RBP_init + 0x38] ^ (@64[RBP_init + 0x38] + 0x62)) + 0x92)

@64[RBP_init + 0x50] = ((@64[RBP_init + 0x38] ^ (@64[RBP_init + 0x38] + 0x62)) + 0x92)

...
Statistics in Table 3 have been extracted from the processing of the first executed virtual instruction.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the bytecode block in bytes</td>
<td>548</td>
</tr>
<tr>
<td>Total number of processed virtual instructions</td>
<td>62</td>
</tr>
<tr>
<td>Total number of underlying native instructions</td>
<td>9,444</td>
</tr>
<tr>
<td>Total number of resulting IR instructions (including IRDsts)</td>
<td>195</td>
</tr>
<tr>
<td>Execution time in seconds</td>
<td>6.4810</td>
</tr>
</tbody>
</table>

*Table 3. Statistics of the first executed virtual instruction*

**The second executed virtual instruction**

The second virtual instruction just zeroes out several internal registers, which are most likely about to be used for obfuscation, as in the second VM.

Statistics in Table 4 have been extracted from the processing of the second executed virtual instruction.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the bytecode block in bytes</td>
<td>755</td>
</tr>
<tr>
<td>Total number of processed virtual instructions</td>
<td>83</td>
</tr>
<tr>
<td>Total number of underlying native instructions</td>
<td>13,740</td>
</tr>
<tr>
<td>Total number of resulting IR instructions (including IRDsts)</td>
<td>259</td>
</tr>
<tr>
<td>Execution time in seconds</td>
<td>7.7718</td>
</tr>
</tbody>
</table>

*Table 4. Statistics of the second executed virtual instruction*

**The third executed virtual instruction**

The third virtual instruction behaves just like the third one of the second VM too – it stores the stack pointer (highlighted in Figure 34). The addition of `0x98` is present due to applied optimizations which took into account the previously executed `POP` instructions in the *Intro* section.
Statistics in Table 5 have been extracted from the processing of the third executed virtual instruction.

<table>
<thead>
<tr>
<th>Size of the bytecode block in bytes</th>
<th>586</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of processed virtual instructions</td>
<td>66</td>
</tr>
<tr>
<td>Total number of underlying native instructions</td>
<td>10,263</td>
</tr>
<tr>
<td>Total number of resulting IR instructions (including IRDest)</td>
<td>207</td>
</tr>
<tr>
<td>Execution time in seconds</td>
<td>6.8428</td>
</tr>
</tbody>
</table>

Table 5. Statistics of the third executed virtual instruction
The fourth executed virtual instruction

We naturally expect this instruction to be a `POP` as in the second VM; however, it is hard to confirm statically as the already described obfuscation techniques make it too hard to understand. One can see part of the virtual instruction in Figure 35.

Statistics in Table 6 have been extracted from the processing of the fourth executed virtual instruction.

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the bytecode block in bytes</td>
<td>4,883</td>
</tr>
<tr>
<td>Total number of processed virtual instructions</td>
<td>425</td>
</tr>
<tr>
<td>Total number of underlying native instructions</td>
<td>71,192</td>
</tr>
<tr>
<td>Total number of resulting IR instructions (including IRDests)</td>
<td>1,038</td>
</tr>
<tr>
<td>Execution time in seconds</td>
<td>28.1638</td>
</tr>
</tbody>
</table>

**Table 6. Statistics of the fourth executed virtual instruction**

When we look closely at certain parts of Figure 35, it appears to be able to behave as a `POP` instruction. The part of the virtual instruction in Figure 36 clearly behaves just like the fourth one of the second VM – it moves a value off the top of the stack, and if the target register is different from the stack pointer, the stack pointer is increased.
**Instruction merging**

However, the instruction also seems to be capable of performing a **PUSH** and other operations as well, based on the operands (Figure 37), which means that it consists of several other instructions merged into one, which is a kind of **obfuscation technique**. It most likely merged several instructions with different rolling keys into one.

![Figure 37](image.png)

**AUTOMATING ANALYSIS OF THE FIRST VIRTUAL MACHINE**

Now that we know what the internal structure of the first VM is like, we can process the VM as the second one since analyzing all the virtual instructions would be complicated due to the additional obfuscation techniques – we can again effectively eliminate them with symbolic execution.

We definitely need to concretize the virtual instruction table and internal registers dedicated for obfuscation as in the previous one, which is not complicated. The question is: What do we do with the second VM?

There is a pretty simple solution – instead of preserving the entire context of the second VM and working with both at once, we can simply concretize the entire second VM as we know what memory ranges belong to the VMs.

We will also ignore all memory assignments to the second VM’s context and not preserve any information about its structure. This will enable us to focus only on the first one and build the same graph as before.

We could also preserve the obfuscated IR of all the virtual instructions of the first VM and use them instead – it would save a significant amount of time during the processing since we would not repeatedly disassemble, translate and deobfuscate the second VM for each opcode in the bytecode blocks of the first VM. However, we want to show that it is possible to process both layers at once.
Processing the initial bytecode block

We processed the very first bytecode block as was described in the previous section. The resulting code still appears to be too complex since we expected a series of \texttt{POP}s, the deobfuscated code and then a series of \texttt{PUSH}es and finally mapping back to the native registers. However, there are additional, multiple branches. One can see part of the code in Figure 38.

Figure 38. The first processed bytecode block
Opaque predicates

Looking at the code more closely, we notice two types of expressions that can be further simplified. The first is the value of `RBP_init`, which is the address of the virtual context and it is known (Figure 39).

<table>
<thead>
<tr>
<th>loc_key_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>@32[RBP_init + 0x97] = 0x0</td>
</tr>
<tr>
<td>@64[RBP_init + 0x1E] = RSP_init</td>
</tr>
<tr>
<td>IRDest = (RBP_init == 0x0)?(loc_key_164,loc_key_163)</td>
</tr>
</tbody>
</table>

Figure 39. Expressions that can be further simplified

Both paths that follow the initial block in Figure 39 contain the same code, hence this is not the same case as with the `POP` virtual instruction, where it was important to know what the target register was because it determined the subsequent behavior of the virtual instruction. These checks are, on the other hand, unimportant and we can just get rid of them – they can be considered as a sort of opaque predicate.

Note that the branch of the `POP` virtual instruction was now optimized out automatically since offsets of the registers were present in the bytecode and directly known.

Finally, these were the last obfuscation techniques, and we can look at the simplified code.

Overview

We are finally greeted with a familiar, even pleasant, view in Figure 40 – as expected the code begins with a series of `POPs` (red) and ends with a series of `PUSHes` (green) that represent parts of the context switches.

Another interesting detail is that the VM uses a special internal register to store the destination address – the final jump is not visible, but the code jumps to `@64[RBP_init + 0x133]`. As was mentioned earlier, the VM also stores the base address of its code section; this is stored in virtual register `@64[RBP_init + 0x80]` in our case.

One can see that the code in Figure 40 also accesses certain data using the base address, specifically at offset `0x0E3808` (blue). After looking up the address, we found that it belongs to a `ServiceStatus` structure (Figure 41).
Figure 40. Code of the processed bytecode

Figure 41. Data accessed by the code – ServiceStatus
It additionally sets a register before recovering the native state to a data address at offset 0x2FB0 (yellow). The address contains a non-obfuscated function shown in Figure 42.

Let us now focus on the destination address (gray) – it is set to <base address> + 0x8C038. Looking up that address in the sample, we see it belongs to the Windows API `RegisterServiceCtrlHandlerW`, which makes sense as the application is a service (Figure 43).

The question is now, what is the return address of the API call? When we look at the end of the code, we see that it sets the return address – the highlighted assignment in Figure 44 appears to be 0x88 bytes above the stack pointer, but we need to keep in mind that we started below the stack pointer because we did not perform the initial context pushing from `vm_init()` and in reality, it is the return address.

The return address is set to another `vm_pre_init()`.

The last part of the code that needs to be analyzed is the body of the loop (Figure 45). It is pretty simple – it zeroes out a memory range. If we look back at Figure 40 and look up what is in `@64[RBP_init + 0x74]`, we see that it is set to the address of the `ServiceStatus` structure (blue) – this piece of code zeroes out the structure. Meanwhile, `@64[RBP_init + 0x4F]` (pink in Figure 40) initially contains the constant 0x1C – size of the structure – and `@64[RBP_init + 0xCC]`, the CPU flags.
Now we look at the discovered non-obfuscated sample and compare it against our findings. We can confirm that we deobfuscated the first bytecode block successfully (Figure 46).

```
064[RBP_init + 0xCC] = (032[RBP_init + 0xCC][0:1] 0 1, 0x1 l 2, parity(032[RBP_init
```

Figure 45. Body of the code’s loop

```
    public ServiceMain
    ServiceMain proc near
    arg_0 = qword ptr 8
    arg_8 = qword ptr 18h

    4B 89 5C 24 18
    57
    4B 83 EC 20
    14 8B 1A
    8B 1C 00 00 00
    4D 9D 07 0E 00
    1F 1F 00 00 00

    loc_1800030660:
    dec rax
    mov byte ptr [rax+rdi], 0
    jnz short loc_1800030660

    3D 15 10 FF FF
    8B C0
    15 BF 8F 00 00
    89 05 80 07 0E 00
    05 C0
    70
```

Figure 46. The same part of code in the non-obfuscated binary

Statistics in Table 7 have been extracted from the processing of the first bytecode block.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of the bytecode block in bytes</td>
<td>695</td>
</tr>
<tr>
<td>Total number of processed virtual instructions</td>
<td>62</td>
</tr>
<tr>
<td>Total number of underlying native instructions</td>
<td>3,536,427</td>
</tr>
<tr>
<td>Total number of resulting IR instructions (including IRDest)</td>
<td>192</td>
</tr>
<tr>
<td>Execution time in seconds</td>
<td>382.5678</td>
</tr>
</tbody>
</table>

Table 7. Statistics of the first processed bytecode block
DESCRIPTION OF OUR FINAL VM ANALYZER CODE

Our final analyzer code consists of several classes that interact together, as described in the following sections. The full code listing is available in [our GitHub repository](https://github.com). The classes follow the high-level descriptions from the previous Automating analysis sections.

Class Wslink

Wslink is a mediator that handles interaction of the remaining classes, its constructor processes the supplied memory dump, and its method `process()` accepts the value of the virtual program counter – pointer to the bytecode – with the opcode of the initial instruction. The bytecode is subsequently processed using classes VirtualContext, SymbolicCFG and MySymbolicExecutionEngine; the resulting control flow graph is written into a **DOT** file `vma.dot`.

Parts of the VM, such as address of the instruction table or offsets of the virtual registers for obfuscation, should be overwritten to provide specific values for individual VMs.

Class VirtualContext

This class represents the virtual context – it contains most notably the initial values of the virtual registers for obfuscation, virtual program counter, and the address of the instruction table.

It also provides several methods for working with the context described in the following sections.

Method `get_next_instr()`

The method `get_next_instr()` applies the address of the instruction table to the destination address to simplify the corresponding expression and attempts to unambiguously determine the address of the next virtual instruction to be executed.

Method `get_irb_symbs()`

This method simply acquires the expressions that should be preserved in the nodes of the resulting control flow graph.

Method `get_updated_internal_context()`

The method `get_updated_internal_context()` updates values of the internal registers that need to be preserved between virtual instructions, such as the virtual program counter or the obfuscation registers.

Method `get_state_hash()`

This method calculates a hash for virtual instructions – the hash is used to specify the actual position in the bytecode to reconstruct the control flow graph without duplicate nodes or paths and to avoid infinite loops in cycles. It is calculated just from the virtual program counter.

Class MySymbolicExecutionEngine

This class overrides the method `mem_read()` of Miasm's class `SymbolicExecutionEngine` primarily to transform memory accesses relative to the virtual program counter and the virtual instruction table into concrete values. It is additionally meant to make the second VM completely concrete when we are processing the first one.
**Class SymbolicCFG**

This class is meant to construct the resulting control flow graph. It uses class `Node` to process individual virtual instructions, to acquire the expressions that need to be preserved, and to determine addresses of the next virtual instructions.

Each `Node` is tied to a hash generated by `get_state_hash()` (as described above) and the address, `StateID`, of the block of code that is being processed. This means that virtual instructions containing unbounded loops cannot currently be processed correctly because when we connect a state to an already processed one, it will not take into account the changes introduced in the body of the loop.

**Class Node**

This class simply represents a node in the resulting control flow graph – it most notably contains the values of the obfuscation registers and virtual program counter that are together called `init_symbols`. These are the values required to determine the addresses of the next virtual instructions.

It provides a method `process_addr()` that can get the following `Node` classes that have not yet been processed and return them along with the expressions that should be preserved in a data-class `ContextResult`.

The new `Node` classes are created based on the supplied addresses using method `_get_next()`, which accepts several arguments. The arguments can instruct the function to slightly modify the resulting `Node` – make a copy of the actual symbolic state when there is a branch, or update `init_symbols` for a new virtual instruction.

**FUTURE WORK**

Once we discovered a non-obfuscated sample, we were not motivated to completely deobfuscate the rest of the code.

Our next steps would consist of:

1. Getting rid of the intro and outro and mapping the virtual registers directly to the native ones.
2. Automatically processing the subsequent bytecode blocks and extending the graph with resulting code listings to get an overview of the whole function.
3. Optionally addressing individual instructions with unbounded loops that cannot be fully processed using symbolic execution (e.g., instructions like `DEC_RC4` mentioned in Miasm’s blog) and manually creating their IR to be added to the graph. We could also experiment with some enhancements of symbolic execution that attempt to mitigate the issue.
4. Optionally matching resulting IR expressions against known IR expressions of assembly instructions to recover assembly code.
CONCLUSION

We have described internals of an advanced multilayered virtual machine featured in Wslink and successfully designed and implemented a semiautomatic solution capable of significantly facilitating analysis of the program’s code. This virtual machine introduced several other obfuscation techniques such as junk code, encoding of virtual operands, duplication of virtual opcodes, opaque predicates, merging of virtual instructions and a nested virtual machine to further obstruct reverse engineering of the code that it protects, yet we successfully overcame them all.

To deal with the obfuscation we modified a known technique that extracts the semantics of the virtual opcodes using symbolic execution with simplifying rules. Additionally, we made concrete the internal virtual registers for obfuscation along with memory accesses relative to the virtual program counter to automatically apply known values and deobfuscate semantics of the virtual instructions – this additionally broke down boundaries between individual virtual instructions. Boundaries are necessary to prevent path explosion of the symbolic execution; we would lose track of the virtual program counter – our position in the interpreted code – without them.

We defined new boundaries by symbolizing the address of the virtual instruction table, since it is required to get the next instruction, and concretized it only when we needed to move to the following virtual instructions. We subsequently constructed a control flow graph of the original code in an intermediate representation from one of the bytecode blocks based on the virtual program counter, and extracted deobfuscated semantics of individual virtual instructions. We finally extended the approach to process both virtual machines at once by entirely concretizing the nested one.