Nightingale: Translating Embedded VM Code in x86 Binary Executables

Xie Haijiang[1,2], Zhang Yuanyuan[1](✉), Li Juanru[1], and Gu Dawu[1]

[1] Shanghai Jiao Tong University, Shanghai, China
yyjess@sjtu.edu.cn
[2] Keen Security Lab of Tencent, Shanghai, China

Abstract. Code protection schemes nowadays adopt language embedding, a technique in which a customized language is built within a general-purpose one, often referred to as the host language, to obfuscate original code through transforming it into a customized form with which the analyst is not familiar. The transformed code is then interpreted by a so-called Embedded VM. This type of transformation does increase the cost of code comprehending and maintaining, and introduces extra runtime overhead.

In this paper, we conduct an in-depth study on embedded VM based code protection and propose a de-obfuscation approach that aims to recover the original code form. Our approach first pinpoints the interpretation procedure and partitions handlers of the embedded VM, and then employs a VM-state based handler translating, which represents the VM-state-updated behaviors of handlers. Finally, the translated operations of each handler is optimized and transformed into host code. After this process, we can obtain a clear and runtime efficient code representation. We build Nightingale, a binary translation tool, to fulfil this de-obfuscation automatically with x86 binary executables. We test our approach on the latest commercial code obfuscators, embedded domain-specific languages and a set of home brewed obfuscation schemes. The results demonstrate that this kind of obfuscated code can be simplified with host language effectively.

Keywords: Code obfuscation · Virtual machine interpreter · Code protection

1 Introduction

Embedded languages are programming languages designed to be used from within another program. Compared with its host language, an embedded language is usually more flexible with clear and simple syntax. For instance, the

This work was partially supported by the Key Program of National Natural Science Foundation of China (Grants No. U1636217), the Major Project of the National Key Research Project (Grants No. 2016YFB0801200), and the Technology Project of Shanghai Science and Technology Commission under Grants No. 15511103002.

Windows operating system provides the *WindowsScriptingHost* API for programs to load and execute scripts written in WSH language. While this hybrid programming style significantly extends the feature of the host language and attains success with many concrete examples (e.g., C and Lua), it may also increase the comprehension complexity and runtime overhead if the embedded language is not familiar to code maintainer and user. For that reason more and more code protection schemes use custom embedded language to impede program analysis and reverse engineering efforts. This type of protection is especially popular with the malware developers, who aim to hide the behavior and character of their program and shield away from the scanning of Anti-Virus software. A prevailing implementation technique for those protection schemes is to design a simple virtual machine. It transforms original code fragment (functions or basic blocks) into bytecode corresponding to this VM, and then simulates it in host language by interpreting the bytecode. Code diversity is also introduced to generate different VMs to frustrate automatic analysis. As a result, it is usually more difficult to analyze and understand such protected code with analysis techniques and tools of host languages.

Difficulties of comprehending embedded obfuscated code mainly comes from comprehending the definition of embedded language and the embedded language VM. In the VM obfuscated executable, instead of analyzing original program code, it is the VM interpreter that requires to analyze. The analysis should first recover the structure of the used VM (e.g., program counter variable, the fetch/decode/execute loop, and instruction buffer) and then understand the obfuscated code. Once the structure is well defined, the syntax and semantics of the target instruction set can be derived with static and dynamic analyses. Previous studies on VM de-obfuscation [3,13,19,20], however, mainly concentrate on comprehending obfuscated code with traditional program analysis and do not consider the characteristic of it. For instance, they are trying to recover high-level syntactic structure (e.g., Control Flow Graph) of the obfuscated code, or employ heavyweight symbolic execution to recover the syntax and semantics of VM bytecode. These analyses usually provide less help when understanding the VM interpreter. As a result, although traditional binary code analysis techniques are well-developed to handle commodity programs, they are sometimes too ideal to comprehend obfuscated code. If the target of the analysis is the embedded language rather than the n host language, a more basic problem is to conduct an embedded language disassembling (or translating) to help understand it.

Methodology. To tackle this challenge, this paper presents a heuristic approach to fulfil embedded language translation. It is profitable to translate the bytecode from the embedded language to the host language. This not only helps comprehend the semantics of the code with simplicity, but also reduces the runtime overhead because the execution in host language is generally more efficient than the interpretive style of the embedded language. Our proposed approach relies on the assumption that *each handler of the embedded language's VM interpreter could be translated into a set of simple operations in host language*, and our target is to automated this inverse procedure and achieve binary code translation.

Main issues of this translation work include: (1) how to pinpoint the interpretation and comprehend handlers, (2) how to translate one handler using the host instructions, (3) how to simplify useless code inserted, and (4) how to replace original obfuscated code. To pinpoint the interpretation procedure, we mainly rely on the feature of how a part of the program is driven by data buffer to identify the VM. Then, a concept of VM-state, which is the core memory operated by the VM, is used to slice code of handlers and build the concise description of each handler. After that, the re-expressed instructions are further optimized to generate a simpler alternative function of the obfuscated code stub. Finally, we use dynamic instrumentation to patch the VM interpreter and replace it with our translated code.

Two properties of embedded VM based obfuscation are leveraged to support our translation. First, most of the embedded bytecode is a transformation of existing program code. Thus it is feasible to re-express it with the original instruction set. This often becomes an important prerequisite for effective de-obfuscation. Second, to communicate with host languages, the embedded code generally uses data structures conforming to host language to pass parameters to and from the host program to the interpreter. For instance, an x86 assembly function will still use stack to pass the parameters even if it is obfuscated.

The core insight of our work is to leverage an abstract **VM-state** to represent the heavily obfuscated operations. Abstractly, the VM-state is the set of intermediate buffer of the VM interpreter, which could be defined through a program analysis of the interpretation. Then the behavior of the VM interpreter is defined by how the VM-state is updated. Through this way different behaviors of various VM interpreters can be expressed in a unified way.

We design and implement an embedded language translator, Nightingale, to execute automated obfuscated code extraction and translation. Nightingale mainly makes use of dynamic analysis to employ the obfuscated code extraction. It monitors certain execution that contains a VM interpretation and extracts handlers of the interpreter. When the handler is extracted, an offline analysis is executed to translate and simplify the corresponding embedded code. Finally, the simplified code in host language is dynamically inserted into the program to replace the original obfuscated one.

Evaluation. To evaluate the effectiveness of our approach, we conduct a series of empirical studies on several code obfuscators. To the best of our knowledge, most previous studies on code de-obfuscation only focus on two mainstream obfuscator manufacturers. While those code obfuscators covers a large portion of obfuscated programs, there are still many custom obfuscators used by different software products in the wild. Our evaluation also considers them and conducts an in-depth analysis on some novel obfuscation measures adopted. In detail, we collect five obfuscated samples from online Capture The Flag (CTF) contests as well as our home brewed sample obfuscated by the popular *VMProtect* obfuscator as one of the most famous obfuscators. We then use Nightingale to analyze these samples and translate their embedded code stubs. While other works try to compare the similarity of recovered code structure with the original one, our validation is simple: we only observe if our rewritten code is able to fulfil the same transformation as the obfuscated one for multiple inputs. If this input-output relationship preserves, it is believed that the translation works. Besides, analysts will get a more comprehensible expression of the program.

Contributions. This paper makes the following contributions:

- We propose an obfuscated code translating approach for code comprehension. Our translating approach adopts a embedded language disassembling methodology and simplifies the obfuscated code. It not only helps understanding the obfuscated code but also improves the execution efficiency to some extent.
- We propose a VM-state analysis to deal with different VM implementations and express the behavior of handlers based on this VM-state. The VM-state based behavior expression is helpful when performing binary translating because it is defined using host language, and is able to be integrated into host program as a patch of the VM code.
- We implement Nightingale, a binary translating tool to fulfil the task of code de-obfuscation. Our evaluation shows different VM implementations can be analyzed and translated by Nightingale with a unified analysis style.

2 Preliminaries

2.1 Basic Concept

Figure 1 depicts a concrete example of VM code embedding. The non-obfuscated program, a Windows x86 or x64 executable, is generated with normal compilation process and the layout of the executable follows standard Windows PE file format. After a VM-based code obfuscation (i.e., a code transformation process), part of the original code is wiped and replaced as control flow transitions to lately inserted code section defined in this paper as a *VM stub*. In Fig. 1, original code of *func_A* and *func_B* is replaced as *vm_func_A* and *vm_func_B*. Notice that *vm_func_A* and *vm_func_B* are not typical binary code functions. Instead, they are composed of the header in the original Code section and a series of bytecode placed at the VM section. Then the VM core is responsible for executing the bytecode in the VM section. A typical header (control flow transition) of VM stub can be a simple branch instruction in code section:

```
00401000|push ebp
00401001|mov ebp, esp
00401003|sub esp, 0x8
00401006|push 0x4020f4
0040100b|jmp 0x4a4a97
```

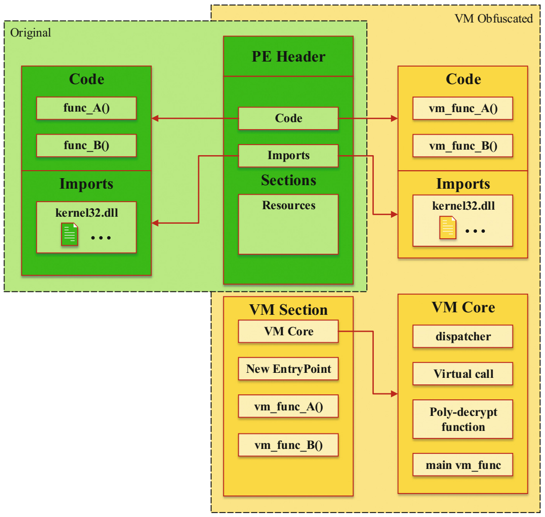

Fig. 1. An instance of VM code embedding

The last *jmp* instruction in this example leads the control flow to the entry point of the VM stub in VM sections, which consists of mainly a VM bytecode buffer and a VM interpreter.

To fulfil the same functionality as the original code, the obfuscator will generate a segment of VM bytecode through analyzing and transforming the original instructions. For instance, if there exists an *add* instruction in original code and the code interpreter also contains an instruction that fulfils addition operation, the obfuscator will then generate a corresponding VM bytecode instruction. The VM bytecode buffer is basically the transformed results of original code with the form of a customized instruction set architecture (ISA). However, not all of the original instructions can be replaced by an alternative VM bytecode. Particular instruction in host language may be complex and the obfuscator may use a set of alternative VM bytecode instructions to replace it. In this manner, the embedded VM code executes within the host language execution environment and always tries to keep the same semantics to prove the reliability.

In the scenario of VM based code obfuscation, the VM interpreter is generally the implementation of a lightweight code interpreter written in host language. Different VMs adopt different designs of ISA and corresponding bytecode handlers. Some VMs are stack machines while some are register machines. However, both implementations follow the common design principle of code interpreter and each consists of basic components such as a *bytecode decoder*, an *execution scheduler*, and numerous *bytecode handlers*, which are core components that determine the ISA of the VM and fulfil the main functionality.

2.2 Assumptions

One assumption in this paper is that the VM used for code obfuscation is a simple interpreter compared with those heavyweight interpreters (e.g., interpreters of Ruby, Lua, and Python). Moreover, we assume that the protected code are simple data transformations that mainly contain plain instructions. This is reasonable because most obfuscators, according to our observation, only deal with those plain instructions. Our assumption is base on the observation of common commercial obfuscators such as *VMProtect* and *ExeCryptor*. The obfuscation is often employed through using SDKs of those obfuscators to transform only part of their code. Otherwise, the obfuscation process may fail or the generated executable may not able to work properly. This indicates that these automated VM obfuscators only deal with relatively simple instructions to prove the stability.

Another important feature is that most obfuscators would not recursively obfuscate invoked functions in the range of protected code. That is, if the protected code contains a function invoking, obfuscators generally do not obfuscate this invoked function. Instead, they just replace the invoking instruction (call or jmp) with a vague stub that does not obviously expose the target function's address.

For commercial VM obfuscators, although we do not know their accurate work mechanisms, we can send a home brewed sample to them and obtain the obfuscated version (these obfuscators provides trial versions). This also helps understand the used bytecode instructions and handlers.

3 VM Code Translating

3.1 Overview

In this paper we aim at translating the embedded VM code, which is mainly generated by automated code obfuscator, into the form of host language of the program. As the embedded code can be seen as an alternative transformation P' that replaces the original transformation P. The target is to recover the original transformation P as much as possible. However, state-of-the-art obfuscators can add various layers of transformations and heavily complicate the process of reverse engineering the semantics of binary code. In most cases it is unpractical to obtain a complete understanding of the underlying logic of a program. Thus we do not pursuit a perfect recovery because this can be seen as a form of decompilation and it is not expected to have a perfect solution to the problem. Our solution is instead to present a generic and practical translation scheme that reveals the state transition of VM code. Concentrating on VM code restricts the scope of the analysis, and helps analyst focus on collect high-level information and identify interesting parts of the obfuscated code. Particularly, in this paper we do not consider the unpacking and anti-analysis code issues. We mainly focus on how to comprehend the structure of embedded VM and how to translate embedded VM bytecode into host language expression.

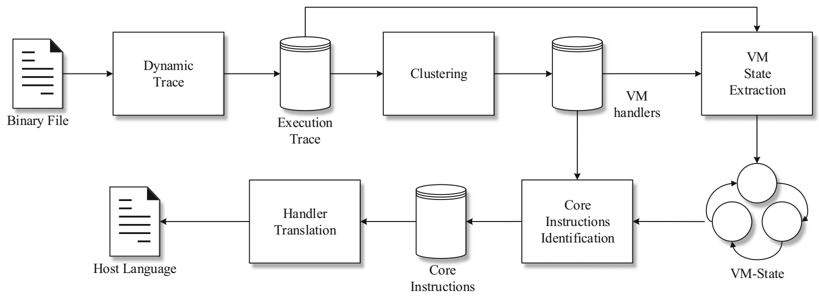

Fig. 2. VM code translating process

Figure 2 depict the entire translating process, which consists of five phases. At the very beginning, the binary code executable is analyzed to first collect execution trace and pinpoint the interpretation procedure. Then, the interpretation procedure is partitioned into different smaller procedures corresponding to VM bytecode handlers. The third phase then extracts and composes a VM-state through synthesizing each handler's behavior. After acquiring the definition of the VM-state, the operation of each handler can be expressed in a new form of host language instructions, and this new representation could be further simplified using traditional program optimization techniques. Finally, to complete the translation, the VM code is replaced by those simplified code through a dynamic binary code instrumentation. In the following, we introduce the details of each phase.

3.2 Interpretation Pinpointing

We propose a handler partition approach, which relies on the analysis of indirected branch semantics. Embedded VM code in host program often executes with a relatively lightweight interpreter, and pinpointing its interpretation process is crucial for the translating. Some studies assume that the VM code and interpreter are placed into a separated section of the executable. Although this corresponds to most commercial VM obfuscators such as VMProtect and Themida, it is not always true for those customized VM obfuscators. Some VM interpreters are embedded into the program during the development stage, hence are located within the same code section as the host code. In this situation, a more generic pinpointing approach is required.

We propose a pinpointing approach based on the feature that the execution of the interpreter is driven by the VM code placed beforehand. A VM interpreter often contains a code dispatching mechanism that responds for choosing the next executing instruction after the interpretation of current bytecode instruction is finished. This code dispatching mechanism can be implemented with a *decode-and-dispatch* style or with a *threaded interpretation* style [14]. For the decode-and-dispatch interpreter, there exists one particular indirect branch instructions (e.g., call eax) that transits the control flow to different handlers.

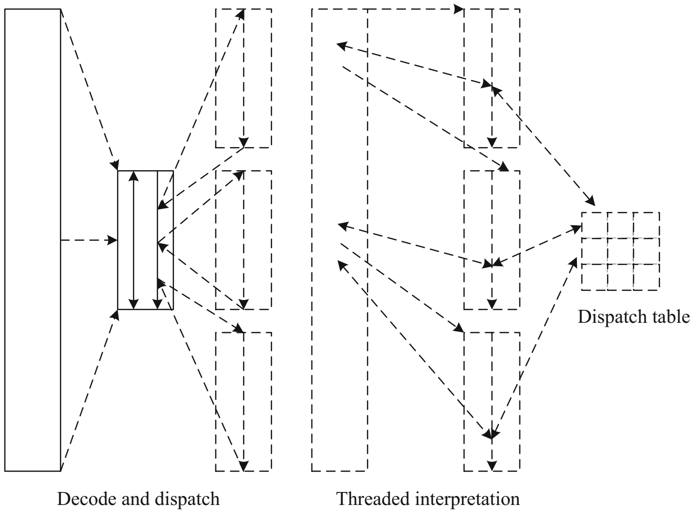

Fig. 3. Two types of interpretation

For the threaded interpretation, the indirect branch instructions may be contained in different handlers (see Fig. 3). However, both kinds of indirect branch instructions, as we called dispatching instructions, are driven by the VM code. Hence for both implementations, we first collect all indirect branch instructions in the execution trace. Then how those concrete control flow transitions are influenced by the input data (from external input or be directly coded in the program) are extracted through a data dependency analysis. The data dependency analysis mainly calculates which part of the input data determines the final indirect branching with a basic data flow analysis against the execution trace. The input data that influences the branching is labeled as the data source. After the analysis, these indirect branch instructions are clustered according to the data source that influence them. The clustering is based on the metric of data source's distance. A basic K-means clustering is adopted here, intending to group those instructions that are influenced by data source with closed distance. According to our observation, the VM code is generally placed in a continuous buffer in data section, or hard coded in code section. If instructions are driven by similar data that is from a small region in memory, it is very possible that the data represents the VM code and the clustered instructions indicate the existence of the interpretation. Another observation is that the embedded VM code has generally been placed during the program generation stage. Thus the buffer of VM bytecode should be placed before the execution of the program. We leverage this property to classify VM bytecode interpreter and the state machine of network protocol, which possesses similar data-driven behavior but the data source is often determined during the execution (i.e., received from the network).

After pinpointing the code dispatching part of the interpretation, the next step is to partition the entire execution trace into individual operation of bytecode instruction handler. We directly use the code dispatching part as the splitter to partition the execution trace, and consider each partitioned segment as a handler. Notice that a handler is not necessarily implemented as a function. Thus a partitioning with the granularity of assembly function is not feasible for this application.

3.3 VM-State Analysis

The key insight of our approach is to recover the format of VM-state, which contains the virtual context of the VM during the interpretation. In general, a VM-state is a set of memory buffer and registers that represents the context of the current VM execution and is maintained by the VM. However, because our analyzed VM is embedded into a host program and the VM itself is implemented using the host language, its VM-state is also expressed using the host memory and registers and is not easily distinguished from the host program's context. Moreover, we expect that the VM-state can still be defined using host language so that in the later translating we can utilize this expression to rewriting the interpretation. To this end, our VM-state analysis is a reverse engineering effort to recover basic format of the VM-state. Since we do not know the virtual ISA beforehand, it is infeasible to define a fixed abstraction of this state beforehand. For instance, if the VM is a stack machine, it often uses a memory buffer to simulate its own virtual stack and manages its own stack *push* and *pop* operations. However, if the VM is a register machine, the abstraction may vary significantly. Hence, our analysis only define a VM as the program that manipulates a memory buffer with relative pointers. Take a virtual *push* operation as an example, our analysis gives the result of a memory write operation only. In this way, we aim to express different VMs in a unified style.

The VM-state reverse engineering starts from analyzing memory and registers updating of each handler in a trace. Now that the aforementioned handler partitioning has already defined the range of each handler, in this phase we concern about how each handler update memory and registers and among the updated content, which part is the used by the following operations. This can be done by a simple data citation analysis: the memory and registers updating of one handler is first recorded and then the following handlers' operations are checked to see which part of those memory buffers and registers is cited in at least one following handler's operation. If the particular memory buffer or register is cited, it is labeled as a *critical* context, otherwise it is labeled as a *forgiving* context. Then we analyze every handler to acquire each one's critical context, and merge them to generate the VM-state. In addition, how each handler manipulates the element in the VM-state is also recorded so that we can define data member of the VM-state with a finer granularity. After this phase the VM-state is extracted from the host program context and the handlers are expected to be translated into host language.

3.4 Handler Translating

Handler translating is the core phase of the entire VM code translating process. It translates variously implemented handlers into a unified form based on the definition of VM-state. That is, one handler's operation is translated as an expression consisted of basic calculation and VM-state elements. For instance, if a handler originally fulfils an add operation on two abstract registers, then the translation results may be:

```
VM-state.buffer[0:4] =
    VM-state.buffer[0:4] + VM-state.buffer[4:8]
```

As the operation of one handler is represented as the operation on the VM-state, it provides a clear description of the handler's behavior with the help of the VM-state. Moreover, it tackles the issue of implementation diversity issue. Even the VM obfuscator adopts code diversity technique to change same handler in different implementations, our analysis is still able to recover the semantics with the VM-state representation.

The detailed handler translating starts from a value-based backward code slicing [3] that resects irrelevant instructions in the handler. It keeps those instructions related to VM-state updating in the handler, which can be employed by a standard slicing approach. Then the remained instructions are transformed into a expression. This expression is generated according to the input and the output of the handler, and illustrates the semantics of the input and the output. Because we can define the input and the output using VM-state, the expression is obviously consists of the relevant VM-state elements.

3.5 Code Simplification

The VM-state based expression of handler may still be complex even if the code slicing removes irrelevant instructions. The reasons for this complexity include the VM obfuscator's implementation is not efficient, or the VM obfuscator intentionally uses a combination of operations to fulfil a simple operation. For instance, some VM obfuscators would use NOR and NAND operations only to emulate every arithmetic operations. To improve the execution efficiency of our translated code, a further code simplification is required.

Our code simplification relies on state-of-the-art code compilation tools to perform code optimization. We first translate every handler in the concrete execution trace to output a VM-state operation sequence. This VM-state operation sequence represents the specific transformation executed by the VM interpretation. Then we rewrite this sequence as a single function using commodity program language so that it can be compiled by state-of-the-art code compilation tools. In our work we use C programming language to rewrite this sequence and use LLVM as the optimization tool. We can compile this single function as a static or a dynamic lib and it could be linked latterly.

3.6 Dynamic Patching

The final step of our translating is to replace the embedded code with a more clear and efficient form. As the embedded code has already been translated and encapsulated into a static or dynamic lib. We can link this lib and use the alternative function to replace the VM stub.

Our dynamic patching is implemented through dynamic code instrumentation. We use popular code instrumentation tools such as Intel's PIN to rewrite the binary code. For a VM stub, we instrument an alternative stub before its entry point to replace its functionality. The control flow is then directed to the new translated function implemented in our lib. And after the execution of this function as a replacement, the alternative stub directly leads the control flow to the invoker of VM stub.

Notice that our translated function is generated by a dynamic analysis phase, which means it may suffer from code coverage problem. The translated function may only able to perform a partial transformation of the original one. However, our observation indicates that most VM stubs are simple transformations with few or no branches. This guarantees our patching works most of the time.

4 Empirical Evaluation

We implement Nightingale, a binary translation tool, to fulfil this deobfuscation automatically with x86 and x64 binary executables. Nightingale consists of an execution trace recording module, an offline program analysis module, and a code patching module. The execution trace recording module and the code patching module are based on Intel's PIN instrumentation framework (900+ LOC) [8], and the offline program analysis module is written in Python (2900+ LOC). In this section we report our empirical study using Nightingale on five different obfuscators including the state-of-the-art VM obfuscator–*VMProtect 3.0*, and four VM obfuscators from different CTF contests that introduce special code obfuscation techniques (all of the samples from CTF contests can be found online).

4.1 Analysis Results

The chosen samples cover mainstream implementation styles of VM obfuscation and the diversity of each sample is significant for analysis. **Foodie-VM** is a simple VM from *0CTF 2015* CTF contest. It is implemented in C and adopts a standard decode-and-dispatch model. **BCTF-VM** is a C++ implemented VM adopting standard decode-and-dispatch interpretation model. It contains basic arithmetic operations (*add*, *sub*, *mul*, and *div*), logic operations (*xor*, *and*), and virtual stack operations (*push*, *pop*). **Paris-VM** is an obfuscation sample from the *PlaidCTF 2014* CTF contest, which utilizes exception-driven and data-driven implicit control flow manipulating to hide the execution path. **DonnBeach-VM** is an obfuscation sample from the *Hack.lu 2012* CTF contest, which utilizes Intel's MMX instruction set to fulfil a simple AES encryption (2 rounds). The overall experiment results are listed in Table 1.

Table 1. Features of different VMs and the analysis results

VMs	Type	Host language	Handlers	VM-state
VMProtect	Threaded interpretation	C++	138	53 units, 156 bytes
BCTF-VM	Decode-and-dispatch	C++	19	59 units, 448 bytes
Foodie-VM	Decode-and-dispatch	C	6	104 units, 260 bytes
Paris-VM	Data-control	C++	20	7 units, 440 bytes
DonnBeach-VM	Decode-and-dispatch	C	16	8 units, 64 bytes

VMProtect. VMProtect adopts a threaded interpretation style rather than the classic decode-and-dispatch style used in previous versions. Each handler of its interpreter contains a decode stub at the end of its procedure and calculates next handler in situ, which increases the difficulty of handler partitioning. However, using our indirect branch instruction clustering, Nightingale still successfully extracts the handler related decoding and dispatching instructions and partitions the handlers from the entire execution trace.

BCTF-VM. For BCTF-VM, because of the C++ implementation style, static program analysis does not recognize the caller and callee relationship of dispatching procedure. Our approach solves this issue through dynamic analysis and successfully recognizes all handlers in the execution trace. The recovered VM-state contains 59 memory units and because this VM does not insert any interfering instructions, the backward slicing only resect a few instructions. We can pinpoint handler with method proposed in Sect. 3.2.

Foodie-VM. Handlers of Foodie-VM generally include core functionality and a decode procedure to determine next handler. The extracted VM-state include 104 memory units, and with value-based backward slicing and handler translating, the result is partially showed in Fig. 6. We then compare this recovered result with the original source code of the VM and find it corresponds to original design well.

DonnBeach-VM. The analysis of DonnBeach-VM finds the dispatcher–an obvious indirect branch instruction at `0x40522F` driven by `buffer_0x405000`, and handlers are easily partitioned due to its decode-and-dispatch interpretation style. However, the VM-state of this obfuscator is hard to be analyzed due to the MMX instructions such as `palignr mmx7, mmx7, 0x7`. To handle this situation we add an extra MMX instruction analysis to Nightingale so that it could parse these handlers. As the handlers are parsed, the VM-state of this obfuscator is finally defined as an 8×8 byte array, which reflects the eight MMX registers (each register is 128-bit). Also notice that in the host language (x86 Assembly) there is no corresponding instruction for those SIMD operations, e.g., an 128-bit xor operation, we manually add some template functions to fulfil such operations.

Paris-VM. Paris-VM is the most special sample in our analysis. It uses three continuous memory buffers plus four independent bytes to store the VM-state. Instead of using either threaded interpretation or decode-and-dispatch interpretation, this VM executes every handler in each iteration. Only one handler is effective in each iteration and this is determined by the current VM bytecode. Each handler first executes its own functionality and then performs a calculation according to the VM bytecode. Only if the result corresponds to particular handler, the updating of VM-state could be preserved. Otherwise, state updating of those ineffective handlers is restored from a mirror VM-state maintained by the VM.

4.2 Case Studies

VMProtect 3.0. In our experiment we use VMProtect 3.0, the latest version of VMProtect software (until 2015.08), to protect a sample program. VMProtect inserts many interfering instructions in the handler to obscure the semantics from being comprehended. Using our VM-state analysis proposed in Sect. 3.3, we obtain a VM-state containing 53 units and according to relevant operations of those 53 units, crucial instructions in this handler can be determined. After the backward slicing with the information collected we obtain optimized handlers and the simplification effect is shown in Fig. 4

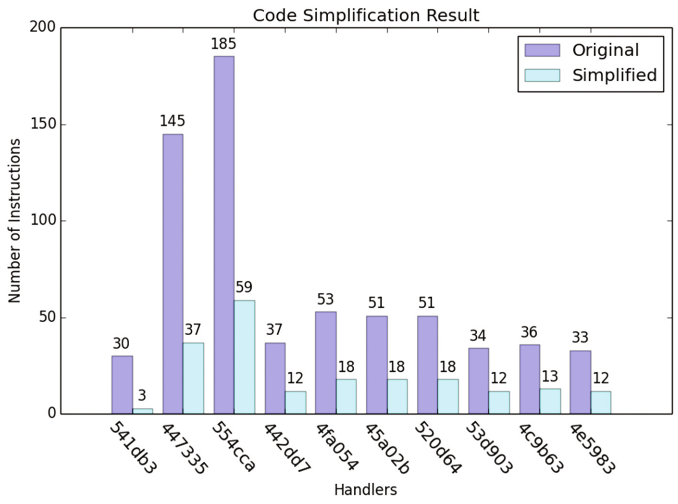

Fig. 4. Handler simplification of VMP handlers

We use one of the handlers to illustrate our analysis. The original handler fulfils the functionality of poping two data elements from the virtual stack (VMProtect uses *ebp* to store the virtual stack's header pointer). Then those two elements are stored into *eax* and *ecx* register respectively. Finally a calculation

`((!eax) & (!ecx))`, i.e., a *NOR* logic computation is executed and the results of calculation and flag register modification are pushed into the virtual VM stack. In addition, the decode procedure, which fetches a 4-byte VM code and uses *ret* instruction to transit to next handler, is attached at the end.

Then we execute the handler translating on this result to obtain the translated code in Fig. 5. It shows the top 10 handlers with the most simplification degree. The translated code is expressed in C and is able to be compiled (the decode part of VM-state is omitted). We then integrated the entire translated code of the execution trace to replace the original VM stub. The execution displays that our code updates the status of the program with the same semantics.

```
...

void handler_NOR()
{
    /* Pop 2 data from VM Stack */
    // 0x44ae3c: mov eax, dword ptr [ebp];
    (eax.r32[0]) = vm_state[22];
    // 0x44ae47: mov ecx, dword ptr [ebp+0x4]
    (ecx.r32[0]) = vm_state[24];

    /* NOR */
        // 0x44ae51: not eax
    (eax.r32[0]) = (~(eax.r32[0])) & 0xffffffff;
    // 0x44ae55: not ecx
    (ecx.r32[0]) = (~(ecx.r32[0])) & 0xffffffff;
    // 0x44ae5d: and eax, ecx
    (eax.r32[0]) = (eax.r32[0]) & (ecx.r32[0]);

    /* Push Result to VM Stack */
    //44ae5f: mov dword ptr [ebp+0x4], eax
    vm_state[24] = (eax.r32[0]);

    // Push Flag to VM Stack
    // 0x44ae6b: pushfd
    (esp.r32[0]) = (esp.r32[0]) - 0x4;
    // 0x44ae76: pop dword ptr [ebp]
    *(unsigned int *)(esp.r32[0]) = eflags.r32[0];
    vm_state[22] = *(unsigned int *)((esp.r32[0]));
    (esp.r32[0]) = (esp.r32[0]) + 0x4;

    /* Fetch next handler offset */
    // 0x44ae8e: mov eax, dword ptr [esi]
    (eax.r32[0]) = (*(unsigned int *)((esi.r32[0])));

        /* Offset Decryption
                Calculating next Handler address */
    ...
}
```

Fig. 5. A translated handler of VMProtect obfuscated code

Foodie-VM. Foodie-VM is a VM that simulates an online shellcode battle between two players. The authors have released the source code so we can verify the de-obfuscation result, especially the recovered VM-state with the original

```
// MOVri source code
int32_t vm(Ins *code, uint32_t code_size, char *input)
{
    ...
    for (i = 0; i < code_size && executing == VM_EXECUTING; ++i)
    {
        Ins ins = read_mem(ctx->memory, ctx->pc);
        Opcode op = get_opcode(ins);
        ctx->pc++;
        switch(op)
        {
            ...
            case MOVri:
                reg0 = get_reg_idx(ins, 0);
                if (reg0 == ERR_REG_IDX)
                    executing = VM_STOP;
                else
                    ctx->reg[reg0] = (Reg)get_imm(ins);
                break;
            ...
        }
    }
    ...
}
```

(a) Source code of Foodie-VM

```
// Result of Handler Translating
void MOVri()
{
    ...
    // Fetch Immediate from VM bytecode
    eax.r32[0] = (*(unsigned short *)((ebp.r32[0]) + 0x8));
    eax.r32[0] = (eax.r32[0]) & 0x3ff;

    // Get VM Context address
    ecx.r32[0] = vm_state[11];

    // Update VM Register with Immediate
    vm_state[18] = (eax.r16[0]);
    ...
    // Update VM PC
    edx.r16[0] = vm_state[17];
    edx.r16[0] = (edx.r16[0]) + 0x1;
    vm_state[17] = (edx.r16[0]);
    ...
}
```

(b) Translated handler of MOVri operation

Fig. 6. Comparison between original code and translated handler of Foodie-VM

structure. We got 104 memory units from the VM-State Analysis. After value-based backward slicing and handler Translating, all of the vm bytecode handlers were successfully translated. Figure 6 lists one bytecode named *MOVri*, which fulfils the function of moving one immediate into VM register that specified in the operand component of the bytecode (we only reserve the key part of the source code and translating result).

In the result of handler translating, new code fetches 4 bytes whose memory address is specified in *ebp.r32[0]* (line 6 of Fig. 6b) and stores the fetched data to *vm_state[18]* (line 13 of Fig. 6b). The corresponding operations in source code are listed at line 19 of Fig. 6a, which indicate the assignment from immediate operand of VM bytecode to the VM register *reg0*. Finally, *vm_state[17]* increases by one (line 16–18 in Fig. 6b), which corresponds to `ctx->pc++` in source code. From the result analysts could infer that *vm_state[17]* is the VM's virtual PC after observing all of the handlers since most of handlers have to update the VM's virtual PC during execution. Thus, our translated results will be helpful to accelerate the process of reverse engineering.

5 Related Work

Code obfuscation is an active and practical field of code protection. Although the theoretic proof of impossibility of perfect obfuscation has been provided by Barak *et al.* [1] in 2012. There are still numerous code obfuscation schemes and most of them are ad hoc implemented. These schemes can be classified into two categories. Schemes in the first category mainly work with source code only, and cover many programming languages include C, C++, Java and C#. Among them, the Obfuscator-LLVM [7] (OLLVM) project is a recently emerged obfuscation scheme that takes advantage of the feature of LLVM-IR to help obfuscate. It is initiated in June 2010 by the information security group of the University of Applied Sciences and Arts Western Switzerland of Yverdon-les-Bains (HEIG-VD). As it works at the Intermediate Representation (IR) level, Obfuscator-LLVM compatible with all programming languages and target platforms currently supported by LLVM. Thus it is widely deployed by many applications on different ISAs.

The second category of code obfuscation schemes could manipulate binary code and are frequently used by commercial software and malware. Two famous obfuscation software providers, VMProtect Software [17] and Oreans Technologies [9], release a vast majority of publicly known obfuscators such as *VMProtect*, *Themida*, *WinLicense*, and *Code Virtualizer*). Other binary code obfuscators such EXEcryptor [16] and SafeEngine [12] may even be more complex, but are not so popular and less used mainly due to their compatibility issues.

To the best of our knowledeg, the work of Sharif *et al.* [13] proposed the first generic de-obfuscation approach against VM based code obfuscation. They mainly relies on abstract variable analysis and binding to recognize VPC (virtual pc of the emulator) and re-construct the CFG. Their work provides a clear definition of the VM analyzed. However, their analysis relies on the assumption of certain VM structure and only focuses on recovering structure (CFG) of the VM bytecode. This is less meaningful for VM based code obfuscation because a VM stub is generally transformed from a relatively simple function or basic block. It is the bytecode's definition rather than the structure that gives the information of the obfuscation code. Yadegari *et al.* [20] also propose a generic de-obfuscation approach. The advantage of their proposed approach is that it does not make any assumptions about the nature of the obfuscation scheme, but instead using semantics-preserving program transformations to simplify away obfuscation code. Although the proposed code simplification technique is effective, the main target of their approach is still the CFG and the approach does not provide any concrete bytecode definition.

Coogan *et al.* [3] proposed a semantics-based approach to de-obfuscate common commercial obfuscators. However they make a strong assumption that requires the involving of system calls to help analyzing. This assumption is not valid for many VM stubs and thus their approach is not universal. Rolf Rolles gives a well-defined de-obfuscation procedure on unpacking virtualization obfuscators in [10] and proposes a semantics-based methods in [11]. However these work lacks details on handling many obfuscator variants and do not scale.

Specific de-obfuscation tools corresponding to particular version of obfuscators are frequently developed. VMSweeper is a plugin of popular Ollydbg debugger that helps decompile VM code of Code Virtualizer (Oreans Technology) and VMProtect (VMProtect Software). Oreans UnVirtualizer is also an Ollydbg plugin that focus on analyzing Code Virtualizer. In response to LLVM-IR based obfuscation, de-obfuscation technique [5] against OLLVM is also proposed. This technique utilize Miasm [2], a Python open source reverse engineering framework, to deal with specific cases of Control Flow Flattening, Bogus Control Flow, and Instructions Substitution. Besides, there are works concentrating on particular aspects of de-obfuscation. Using symbolic execution to help de-obfuscate VM stub is a promising strategy and many studies have been proposed [6,15,19]. Other de-obfuscation techniques include using probable-plaintext attacks to de-obfuscate malware [18] and simplifying obfuscated machine Code [4].

For famous code obfuscator, corresponding analysis tools are able to deal with fixed pattern and recover the obfuscated code with necessary manual effort. However, as the obfuscators change or evolve, these tools are immediately not available. This becomes an endless arms race and the designers of VM obfuscator have the advantage of adopting "security by obscurity" strategy. Moreover, for those obfuscators in the wild, there is no known effective de-obfuscation tool to analyze them. As a result, our automated and universal analysis is more profitable.

6 Conclusion

In this paper we study the VM based obfuscation and propose a binary translation approach to simplify the embedded VM stub in a host program. Our approach differs from most recent de-obfuscation schemes for its VM-state analysis, which is a universal analysis against various VM implementations. Based on the VM-state a clear expression of VM handler is generated and translated into host language. This translated code can replace the VM stub and fulfil same functionality, and is easily to understand and more efficient. Experiments on five different VMs illustrate the feasibility of our approach.

References

1. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im)possibility of obfuscating programs. J. ACM **59**(2), 1–6 (2012)
2. CEA IT Security. Miasm: Reverse engineering framework in Python. https://github.com/cea-sec/miasm
3. Coogan, K., Lu, G., Debray, S.: Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS) (2011)
4. COSEINC. COSEINC OptiCode: Deobfuscate Machine Code. http://opticode.coseinc.com/
5. Gabriel, F.: Deobfuscation: recovering an OLLVM-protected program. http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html
6. Guillot, Y., Gazet, A.: Automatic binary deobfuscation. J. Comput. Virol. **6**(3), 261–276 (2010)
7. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM - software protection for the masses. In: Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO) (2015)
8. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation (2005)
9. Oreans Inc. Oreans Technology: Software Security Defined. http://www.oreans.com/
10. Rolles, R.: Unpacking virtualization obfuscators. In: Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT) (2009)
11. Rolles, R.: The case for semantics-based methods in reverse engineering. In: RECON (2012)
12. Safengine.com. Safengine Protector. http://safengine.com/
13. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. In: Proceedings of the 30th IEEE Symposium on Security and Privacy (SP). IEEE (2009)
14. Smith, J., Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Elsevier, Amsterdam (2005)
15. Souchet, A.: Obfuscation, breaking kryptonite's: a static analysis approach relying on symbolic execution. http://doar-e.github.io/blog/2013/09/16/breaking-kryptonites-obfuscation-with-symbolic-execution/
16. StrongBit Technology. EXECryptor - bulletproof software protection. http://www.strongbit.com/execryptor.asp
17. VMProtect Inc. VMProtect Software Protection. http://vmpsoft.com/
18. Wressnegger, C., Boldewin, F., Rieck, K.: Deobfuscating embedded malware using probable-plaintext attacks. In: Stolfo, S.J., Stavrou, A., Wright, C.V. (eds.) RAID 2013. LNCS, vol. 8145, pp. 164–183. Springer, Heidelberg (2013). doi:10.1007/978-3-642-41284-4_9
19. Yadegari, B., Debray, S.: Symbolic execution of obfuscated code. In: Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS) (2015)
20. Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S.: A generic approach to automatic deobfuscation of executable code. In: Proceedings of the 36th IEEE Symposium on Security and Privacy (SP) (2015)