Inspecting Virtual Machine Diversification Inside Virtualization Obfuscation

Naiqian Zhang*, Dongpeng Xu*, Jiang Ming†, Jun Xu‡, Qiaoyan Yu*
*University of New Hampshire
†Tulane University
‡University of Utah

***Abstract*—Virtualization obfuscators are commonly employed to safeguard proprietary code or to impede malware analysis. Despite significant efforts to combat these obfuscators over the past decade, code virtualization continues to be an exceedingly effective obfuscation technique. At the core of modern virtualization obfuscators are the virtual machines (VMs), which employ a variety of diversification techniques to complicate their internal structures. Due to its intricate and diverse nature, reverse engineering one VM is a time-consuming task and is not useful in cracking other VMs. Yet, despite the success of these VMs, there has been no systematic study of their diversification techniques, creating a knowledge gap that needs to be addressed to enhance VM deobfuscation.**

This work aims to bridge the above gap. First, we categorize and unveil the techniques under the hood of VM diversification, from the perspectives of VM interpretation, bytecode organization, and handler permutation/relocation. This systematic knowledge about modern virtualization is a crucial contribution to the field. Second, we develop an automated tool to identify the VM diversification techniques adopted by state-of-the-art virtualization obfuscators. The results demystify how the VM diversification methods are deployed in practice. Third, our research also involves patching current deobfuscation tools using the newly revealed knowledge of VM diversification to overcome their weaknesses. This outcome highlights how the results of our study pave the way for next-generation VM deobfuscation.

1. Introduction

In the realm of computing, virtualization generally refers to the techniques that run virtual machines on versatile platforms [1]. Besides the popular hardware virtualization like VMWare [2], the power of virtualization has been unleashed in software obfuscation for protecting commercial software. Virtualization obfuscation converts the original code (e.g., x86 instructions) into virtual bytecode that can only be emulated by an internal virtual machine (VM). With the advances in virtualization techniques in recent years, a multitude of obfuscation products have become available on the market [3]–[12] for various platforms and programming languages.

It is not surprising that cybercriminals have quickly caught up with technological trends. In fact, virtualization obfuscation has become the leading-edge technique adopted by malware developers in order to frustrate malware analysts [13], [14]. Driven by huge financial gains, a large number of cryptocurrency mining malware and ransomware protect their secrets via code virtualization [15]–[19]. In May 2019, Chinese hackers infected over 50,000 servers around the world with cryptocurrency mining malware, whose kernel-mode rootkit is protected by code virtualization to frustrate reverse engineers and malware researchers [17], [18]. TikTok [20] adopts virtualization obfuscation to hide its data collection behaviors. Overall, virtualization obfuscation is not necessarily a criminal-focused technique. It is usually used for protecting benign and legitimate code, but the abuse of virtualization immediately captured the attention of security professionals after it raised public concern. As a result, many deobfuscation tools have been developed with significant efforts to reverse engineer virtualization obfuscation [21]–[29]. These tools have demonstrated their effectiveness in cracking virtualization obfuscators, especially their early versions released before 2013.

In the ongoing arms race, virtualization obfuscator vendors continually release updated versions of their products to enhance resistance against reverse engineering endeavors. A prevalent strategy among these vendors involves **employing diversification to generate a multitude of heterogeneous VMs within the obfuscators**. For example, in Oreans virtualization obfuscators [3], each VM is defined by a unique configuration file, as shown in Figure 13, specifying the applied diversification methods, such as register relocation and opcode permutation. Consequently, all VMs undergo diversification, significantly escalating the complexity of reverse engineering efforts.

Motivation. The current state of this arms-race is yet to be fully understood. Both sides have proposed numerous prototypes and tools, with their creators claiming superiority over their opponents. In particular, VM diversification has emerged as a game-changing technique that poses a significant challenge to existing deobfuscation techniques. This is because cracking one VM is no longer sufficient to analyze others. However, the study of the modern VM diversification techniques and their impact on deobfuscation is still unclear.

Additionally, recent studies [30]–[33] have raised concerns over the insufficiency of deobfuscation tools' capability. They reveal that the development of deobfuscation is falling behind the pace of virtualization obfuscation. To

2025 IEEE Symposium on Security and Privacy (SP) | 979-8-3315-2236-0/25/$31.00 ©2025 IEEE | DOI: 10.1109/SP61157.2025.00071

DOI 10.1109/SP61157.2025.00071

catch up, it is crucial to gain a systematic understanding of the new diversification techniques adopted by virtualization obfuscators and how these techniques impede deobfuscation tools. This forms the motivation for conducting this work.

Our Work. Our work involves three components.

Taxonomy: The first part of our study is a qualitative analysis to derive a taxonomy of diverse VM techniques. Following a top-down perspective, our analysis models a VM as a three-level process (*interpretation*, *bytecode*, and *handler*). We discovered that modern obfuscators incorporate rich diversification techniques at each level. In total, 27 techniques belonging to 11 categories have been developed. *Our study successfully demystifies the details of each diversification technique, filling a critical knowledge gap.*

Measurement: The second part of our study incorporates semi-automated methods to scrutinize the execution traces extracted from programs obfuscated by existing tools. The methods identify and extract the major techniques of diverse VMs adopted by different obfuscation tools. The results show that the techniques covered in our taxonomy have been widely deployed in practice, significantly escalating the difficulty of reverse engineering.

Enhancement: The VM diversification knowledge revealed in this work offers plenty of opportunities to reinforce existing deobfuscation tools. We have patched existing tools to overcome their limitations. The result shows that *the patched tools reveal a substantial increase in precision, a reduction in errors, and a significant boost in overall performance when compared to their original versions.*

Contributions. Our main contributions are as follows.

- We present an in-depth taxonomy of the existing VM diversification techniques, bridging a knowledge gap in the current understanding of virtualization obfuscation.
- We develop a new tool to detect and extract these VM diversification features from state-of-the-art virtualization obfuscators.
- We applied the VM diversification knowledge to enhancing existing deobfuscation methods and observed a significant improvement in their performance.

2. Background

2.1. Virtualization Obfuscation

In the realm of software security, the idea of virtualization obfuscation has become one of the most sophisticated code obfuscation techniques [34]–[36]. It transforms selected program parts to bytecode in a customized virtual instruction set architecture (ISA). At runtime, the bytecode is emulated by an embedded VM running on the real machine. Virtualization obfuscation can be seamlessly integrated with other obfuscation schemes such as data encoding [37], [38], metamorphism [39], [40], and control flow obfuscation [41], [42], rendering traditional static and dynamic analysis techniques ineffective [43], [44]. Over the past decade, virtualization obfuscation has been developed as a set of commercial software protection products [3], [4], [6], [8]–[11] and research tools [7], [45]–[47].

In recent years, mainstream virtualization tools have been upgraded to diversify their bytecodes and VMs, producing obfuscated programs that significantly differ from each other. ***VM diversification*** refers to the technologies to make the VMs highly disparate to evade the deobfuscation analysis. For example, if one VM diversification can process N features in the VMs and each feature has M possible choices, then the obfuscator can generate M^N different VMs. These plenty of diversified VMs heavily impede deobfuscation efforts.

Code Virtualizer [3], developed by Oreans Technologies, applies virtualization to user-defined code areas in various executable program formats. It includes dozens of VMs whose names combine an animal name and a color (e.g., "Fish-Black" and "Tiger-White"). The animal name refers to a custom VM architecture, and the color is associated with different variants of that architecture. This diversity of VMs lays the foundation of Code Virtualizer's strong obfuscation capability, and as widely recognized [48], Code Virtualizer's diverse animal VMs have been a challenging puzzle for virtualization deobfuscation. Oreans also develops two other software protection products, Themida [4] and Winlicence [5]. The two products include the same virtualization technique as Code Virtualizer, which we omit for simplicity.

VMProtect [6] is another famous commercial virtualizer. It supports various languages like C++, Visual Basic, and Pascal. The main obfuscation mechanisms in VMProtect include virtualization, mutation, and combined protection. Instead of involving the animal VMs in Code Virtualizer, VMProtect uses mutation as a lightweight way to support VM diversity.

Tigress [7] is a virtualization obfuscator for C programs. It was primarily developed by Professor Christian Collberg at the University of Arizona. Tigress virtualization is source-to-source, i.e., the original C program is translated to a new C program with the VM and bytecode embedded. The VM diversity in Tigress is achieved by applying a variety of heterogeneous obfuscations to the virtualized program. It supports abundant VM customization options, such as opaque predicate insertion, data encoding, self-modifying, and JIT compiling. Tigress compiles these diverse options into four recipes [49] as user guidelines.

Code Virtualizer, VMProtect, and Tigress are widely regarded as state-of-the-art virtualization obfuscators. They are actively maintained, and their documents are up-to-date. The commercial versions of these tools (Code Virtualizer 3.1.4, VMProtect 3.6.0, Tigress 3.3.2) are used in this work.

Other Obfuscators. We investigated many other VM-based obfuscators but found that they do not fit our study. Rewolf-x86-virtualizer [45] is a simple VM obfuscator for Windows PE files. It failed to generate valid obfuscation results due to the lack of active maintenance in the past ten years. DynOpVm [47] implements a dynamic control-flow-aware mapping between virtual and original instructions, but it is not publicly available. Furthermore, VirtualMachineObfus-

cationPoC [50] is a prototype software only handling specific instructions. Loki [31] is a recent obfuscator leveraging Mixed-Boolean-Arithmetic, point function, and superoperators. Loki is out of our scope because it is not virtualization-based.

2.2. Deobfuscation: What Do We Have Now?

The research community has designed various automatic deobfuscation methods to analyze the VM obfuscated code. One intuitive method implemented in tools like Virtual Deobfuscator [24] is to scan and match the patterns in virtualized programs. However, this strategy is easy to circumvent as the patterns are easily broken by the VM diversification methods.

The second category of work attempts to reverse-engineer the VM interpreter [21]–[25]. They focus on identifying the "decode-dispatch-based" interpretation, a conventional way to implement an instruction set virtualization [1]. The key feature is a central loop that fetches virtual instructions based on the current value of a virtual program counter. Then it dispatches to the corresponding handlers that perform the actual behavior of that virtual instruction. After recognizing the interpretation structure, they simplify the handlers using approaches such as program synthesis [51], symbolic execution [52], [53], or compiler optimizations [28], [54]–[56].

The third deobfuscation strategy is to strip off the virtualization obfuscation layer from the tedious execution instructions [26], [27], [57], [58]. They adopt dynamic taint analysis or concolic execution to identify the instructions contributing to the actual program behaviors. As the state-of-the-art work in this category, VMHunt [59] identifies the context switches between the virtualized and normal program portions, and then extracts the program logic related to the context switches.

3. A Taxonomy of VM Diversification

In this paper, we define a new way to categorize the VM diversification methods from a top-down perspective as three levels. The top level is *interpretation*, which decides the fundamental design of a VM, e.g., fetching bytecode and calling the corresponding processing code. The middle level is *bytecode*, which defines the bytecode format and run-time environment. The bottom level is *handlers*, which reveals how the virtualized code is actually executed on a real machine. Figure 1 shows the structure of a VM and the three-level abstraction.

Modern VM diversification can be applied to all three levels to produce highly diverse VM implementations. Typically, a diversification method is designed as various options for a VM feature. For example, on the interpretation level, the VM design could be stack-based or register-based, so two options are feasible. In fact, VM diversification actually involves a combination of the various variants of these features. If one VM diversification includes N features and each feature has M possible options, then theoretically the obfuscator can generate M^N different diversified VMs.

TABLE 1: The three-level model and VM features.

Level	Features
Interpretation	• VM Construction: static, obf-time, run-time • Instruction Set: CISC, RISC • Dispatch flow: loop, threaded • Architecture: stack-based, register-based • Compound VMs: multiple, nested • Context: direct, indirect
Bytecode	• Structure: linear, linked-list • Opcode Permutation • Bytecode Mutation • Stealth Mode
Handler	• Encryption • Handler Mutation • Jump Table

The novelty and contribution of our work is to systematically examine the VM diversification features at each level. Guided by the new categorization, we present an original and comprehensive view of VM diversification methods, which has not been presented by previous work and also can be easily digested by the audience. Table 1 gives an overview of the features to be discussed.

3.1. Top-level: Interpretation

The interpretation level plays a crucial role in defining the overall VM architecture. It addresses fundamental questions regarding VM construction and the internal control flow. The VM diversification on this layer encapsulates a range of essential features as follows.

VM Construction. Based on when the actual VM is built, the VM diversification can be divided into three different styles: *static*, *obfuscation-time*, and *run-time* construction. Figure 2 describes the three types, respectively. In *Static* construction, VMs are hard-coded template codes that can be directly inserted into a program. This method is easy to implement but prone to reverse analysis. Since the VMs are statically hard-coded into the obfuscators, their patterns are fixed and easy to find. A more advanced type is *obfuscation-time* construction, which constructs the VM from a set of configurations during obfuscation time. The difference from static construction is that the VM here is generated rather than just a piece of static code, so the virtualizer has the flexibility to generate different VMs. Virtualizers may also implement *run-time* construction, so the VM does not exist until the obfuscated program starts running. In this case, the VM never appears in the binary code. One common run-time construction technique is called Just-In-Time (JIT) VM, as shown in Figure 2(c). The virtualizer translates the VM and bytecode into an intermediate code bundled with a JIT compiler. When the obfuscated program is running, the JIT compiler will first compile the intermediate code to an executable code (the VM and bytecode) and then run from there. Therefore, the VM only appears in the run-time memory.

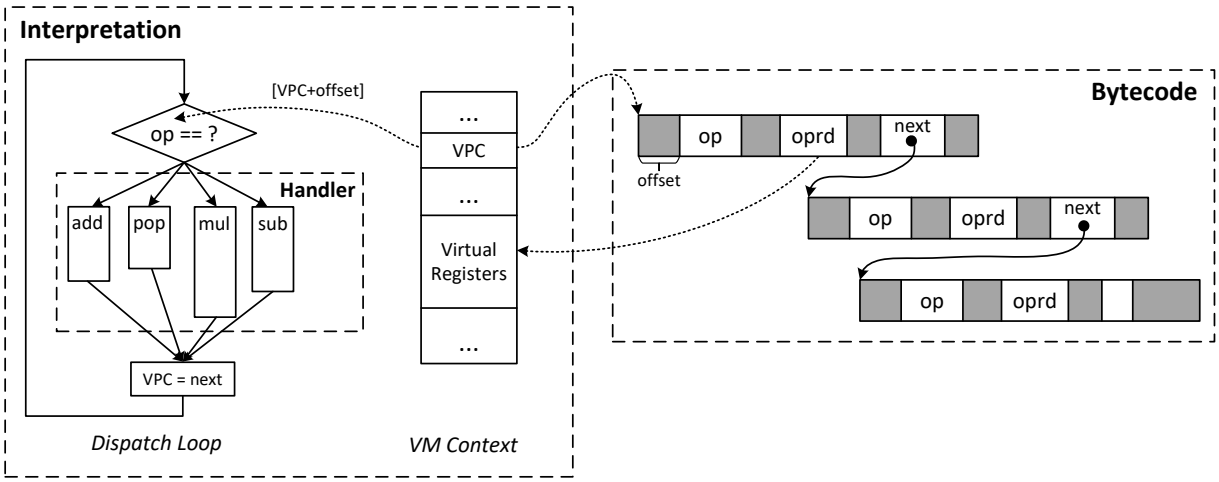


Figure 1: The architecture of a normal VM showing the interpretation, bytecode, and handler levels. The interpretation has a dispatch-loop. The bytecode is a linked-list, where each node is one bytecode instruction. The Virtual Program Counter (VPC) always points to the start of the current bytecode instruction. The dashed arrows represent possible memory access.

Modern VM diversification often uses obfuscation-time and run-time VM construction to make every generated VM different. Using Code Virtualizer as an example, when the "Opcode Permutation" feature is on, it generates a different location for the opcodes in every VM. Traditionally, finding the opcode location is a milestone in cracking the VM. However, it does not apply to the obfuscation-time and run-time VM construction, because every VM has a different opcode location. Finding the opcode location in one VM cannot be directly reused in another VM. Section 3.2 explains more details related to opcode permutation. Consequently, the dynamic features introduced by obfuscation-time and run-time constructions dramatically increase the difficulty of cracking these diversified VMs.

Dispatch Flow. Dispatch flow [60] indicates the control flow structure of the interpretation procedure. VM obfuscators have adopted heterogeneous interpretation structures to vastly diversify the VMs. Two main dispatch flows are summarized as follows.

- *Dispatch-Loop* is the classic way to implement code virtualization, as shown in the left part of Figure 1. The interpretation process is implemented as a branch statement inside a loop. In each loop iteration, the branch will fetch the opcode from the current bytecode instruction and then choose one handler to run based on the opcode. The loop continues until all bytecode instructions are interpreted.
- *Threaded Code* is an alternative VM structure [61], [62] adopted by commercial obfuscators to reduce the dispatch branch overhead in Decode-Dispatch architecture. As shown in Figure 3, the bytecode operations, such as `VPC = next`, are attached to the end of each handler. Indirect jumps are employed to implement the dispatch behavior. Hence the whole dispatch loop is removed, which is why it defies the assumption held by many deobfuscation works. The threaded code interpretation is very popular in animal VMs from Code Virtualizer.

Virtual Instruction Set. A Virtual Instruction Set [63]–[66] defines the instructions that a VM can interpret and execute such as arithmetic computation (*V_add, V_sub, ...*), logic function (*V_gt, V_eq, ...*), and data manipulation (*V_load, V_store, ...*). Traditionally, two primary choices in the design of VM instruction set are Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) [21]. The design choices diversify the number of handlers and the functionality of each handler.

- *RISC* VMs are classical and straightforward VM designs. It has more handlers where each handler only simulates simple behaviors. Usually, one x86 assembly instruction needs multiple handlers to simulate a RISC VM.
- *CISC* VMs have the opposite design. It uses fewer handlers than RISC VMs, but each can simulate complex behaviors. Typically, one CISC handler can simulate one or multiple x86 instructions.
- *Hybrid* VMs contain both CISC and RISC handlers. The VMs in Code Virtualizer have adopted this hybrid style. Hybrid design can produce more robust obfuscated code, but bytecode decoding is also more complex.

The RISC and CISC VMs in Code Virtualizer have different types of instruction sets. RISC VMs have more handlers than CISC VMs. Meanwhile, the function of each handler in RISC VMs are much simpler than those in CISC VMs. For instance, a CISC VM usually has a single and independent handler to implement the arithmetic operation, like *V_add* and *V_sub* handlers. However, a RISC VM needs to combine several handlers or reuse certain handlers to implement the same arithmetic operation. That means a RISC VM has no official addition handler. Instead, it invokes the *V_load*, *V_store*, and *V_mov* handler, which are much simpler than *V_add*, to implement the addition operation. The detailed statistical analysis can be found in Sec. 5.3.

Architecture. Modern VM diversification involves the following two VM architecture types.

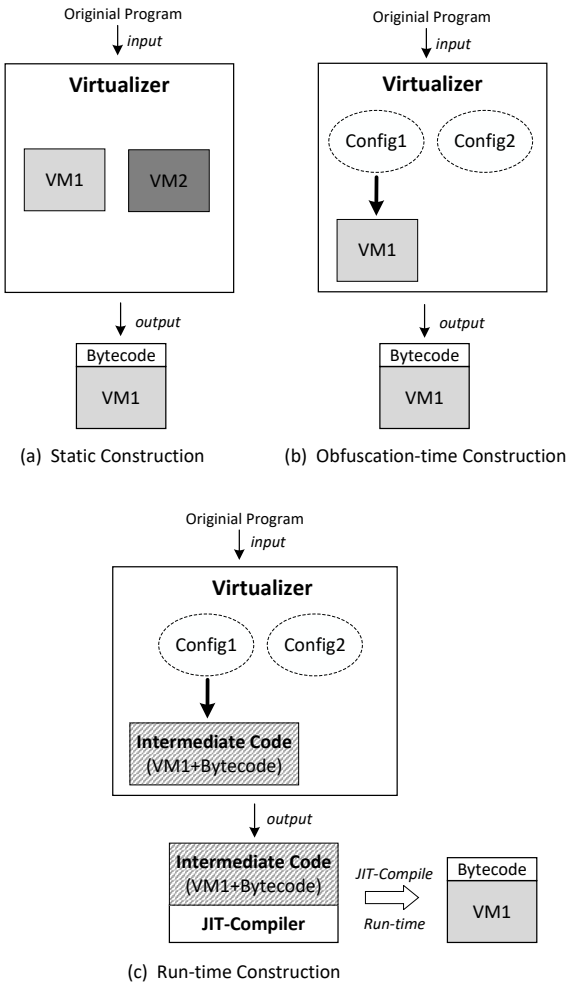


Figure 2: Three types of VM construction. The bold arrow means constructing a VM from a configuration.

- *Stack-based* interpreter is a simple and classical VM design, where the operands are implicitly located in stack frames. VMProtect [6] adopts this design style. Taking the addition operation as an example, as shown in Figure 4 (a), V_add handler in VMProtect needs to pop two values off the virtual stack and the second value is added to the first value. Then the new first value is pushed back to the virtual stack.
- *Register-based* interpreter has to specify the operands explicitly, but it can significantly reduce the number of executed instructions. Due to the difficulty of analysis, register-based VM is more attractive to malware developers. Code Virtualizer [3] has implemented register-based VMs. Figure 4 (b) shows an addition operation in a register-based VM. The VM uses V_mov instruction to move two values to different virtual registers. After that, it uses the V_add handler to add those two values together and then stores the result in another register using V_mov again.

Compound VMs. VM diversification leverages compound VMs, i.e., combining multiple different VMs in a single obfuscated program. It raises a solid challenge to the existing deobfuscation methods, but also at the cost of high overhead. Two patterns are observed in the current VM diversification methods.

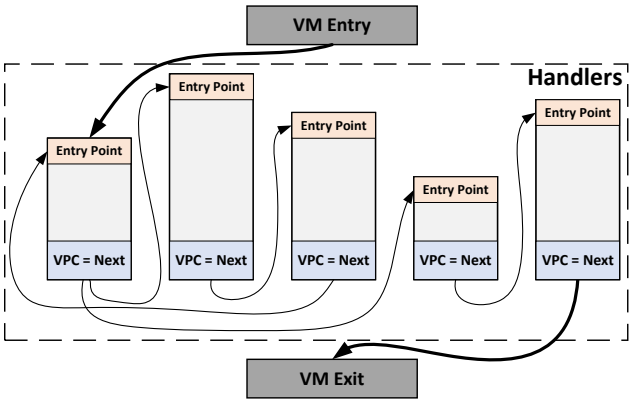


Figure 3: The architecture of threaded code. `VPC = next` at the end of each handler fetch the next virtual instruction and jump to the corresponding handler.

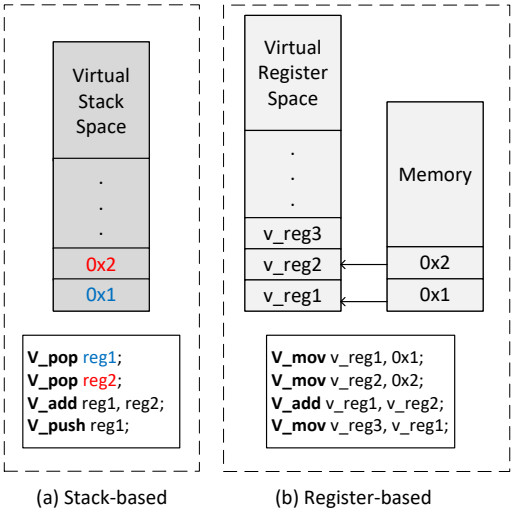


Figure 4: Two types of VM architecture to implement addition operation *add 0x1, 0x2*.

- *Multiple VMs.* The obfuscator provides a list of VMs with various interpretation methods and architectures. Users can apply different VMs to different sensitive areas in the same program, so the execution trace of that program contains multiple virtualized snippets of different VMs. As a result, cracking one VM provides very little information for cracking other VMs.
- *Nested VMs.* The other compound VM scheme is to create nested, multiple-layer virtualization [3]. It applies another virtualization layer to an existing VM (usually a part of the VM, e.g., handler functions), to transform the original program into two nested-VM executions. We observe that the number of instructions from nested-VM obfuscated programs is 4.5X more than one single VM, so nested VMs increase the cracking difficulty.

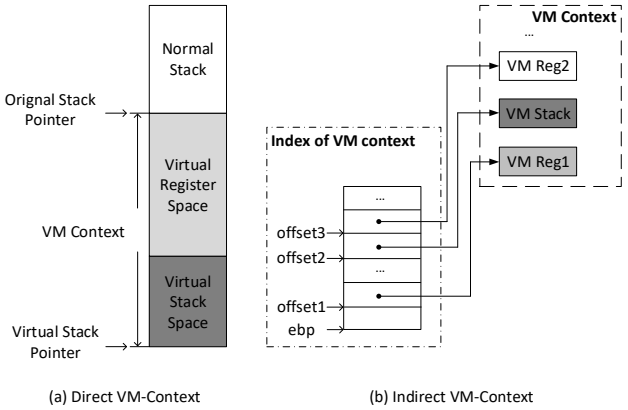


Figure 5: Two types of VM context implementation.

All modern virtualization obfuscators support multiple VMs by different implementations. Code Virtualizer provides complex nested VMs such as Puma, Shark, and Eagle. We have identified the two internal nested VMs, e.g., Puma is constructed by applying Tiger to Fish. More details are reported in the analysis of Code Virtualizer in Section 5.3.

VM Context. A VM uses memory space to store the current execution state, including the Virtual Program Counter (VPC), virtual registers, virtual stack, etc. This memory space is called VM context. It maintains the critical states that guarantee the VM is running correctly. For example, VPC saves the virtual instruction that will be executed next.

We have observed two types of VM contexts generated by VM diversification, as shown in Figure 5. The first type is *direct context*, a fixed length of memory allocated on the stack. The second type is *indirect context*. The VM is accessed by a list of indexes, whose locations are calculated from a base address and an offset at run-time. The base address and offset are set up at the VM initialization stage. In Figure 5(b), the based address is saved in EBP, and the VM context is accessed by dereferencing the addresses such as `ebp+67`. The indirect context is more difficult to track because the base address and offset change in every VM execution.

3.2. Middle-level: Bytecode

Bytecode is the virtual instructions interpreted by a VM. An obfuscator translates the original unobfuscated program into this bytecode, which can be interpreted by the VM during run-time. Traditionally, this intermediate-level code is called "bytecode" because its opcode length is one byte. Although we have observed that some VMs have multiple-byte opcode, we still use the term bytecode to refer to the intermediate representation in VMs.

Bytecode Structure. Bytecode must be organized using a regular data structure to facilitate VM access and interpretation. The basic way is to place the bytecode instructions one by one inside a contiguous memory range like an array. We call this bytecode structure *contiguous*. Tigress and VMProtect adopt this basic structure. Figure 6 shows

```
1  enum ops {
2    Locals = 116, Plus  = 135, Load   = 60,
3    Goto = 231, Const = 3, Store = 122, Return = 72
4  };
5
6  unsigned char bytecode[41]  = {
7    Locals,24,0,0,0,Const,1,0,0,0,Locals,24,0,0,0,
8    Load,Plus,Store,Locals,28,0,0,0,Const,0,0,0,0,
9    Store,Goto,4,0,0,0,Locals,28,0,0,0,Load,Return };
10
11 int main() {
12   while (1) {
13     switch (*pc) {
14     case Const: pc++; (sp+1)->_int = *((int *)pc);
15           sp++; pc+=4; break;
16     case Load: pc++; sp->_int = *((int *)sp->_vs); break;
17     case Goto: pc++; pc+= *((int *)pc); break;
18     ...
19     }}
20 }
```

Figure 6: The bytecode structure in a Tigress VM to obfuscate the code: *main(){int x; x++;}*. The opcode and operand of the bytecode are separately presented in red and blue.

an example of the bytecode in Tigress' VM. It provides source code level obfuscation. The bytecode is located in the array `bytecode` in line 6. The bytecode can be divided into two parts, opcode in red color and operand in blue color. Note that some bytecode does not have operand. Bytecode is the core structure in the VM obfuscated program. In this contiguous structure, if the reverse engineer locates any bytecode instruction, they are very likely to find the rest bytecode around it. As a VM diversification feature, obfuscators like Code Virtualizer implement *linked-list* as the bytecode structure. In Figure 1, Every instruction is one node in the linked list. The crucial parts of a node are the opcode `op` and operands `oprd`. Note that they are placed at a specific offset within the node, so the VM will use `[VPC+offset]` to fetch the `op` from the bytecode instruction. The linked list structure breaks the contiguous connection between two bytecode instructions and thus largely increases the difficulty of locating all bytecode. Moreover, the offset inside every node is randomized via the following method to further impede reverse engineering.

Opcode Permutation. Opcode permutation means that the locations of opcodes (and operands) are changed in every obfuscated program. For example, the size of the gray areas in Figure 1. If attackers know the opcode or operand offset in one VM, they still have to search when reversing another VM. Particularly, when the VMs are generated at obfuscation-time or run-time as described in Section 3.1, the opcodes are permuted differently even if the two VMs are constructed using the same configuration. For example, if two VM instances are constructed from Fish in Code Virtualizer, they still have different opcode offsets. Consequently, opcode permutation diversifies the bytecode structure, forcing attackers to repeat the tedious work and, thus, increasing the difficulty of reversing.

Bytecode Mutation. Because the original program is represented in the form of bytecode, it exposes opportunities to mutate the bytecode program. Modern virtualization implements various mutation strategies. For example, it changes

the order of running bytecode during the VM execution without affecting the execution result. Taking the arithmetic operation $x + y + z$ as an example, this feature changes the calculation order such the actual calculation process might be $x + z + y$ or $y + z + x$.

Another bytecode mutation strategy is to insert *garbage bytecode* instructions. These instructions appear in the same way as any regular bytecode instructions, but the associated handlers do not perform any useful function. Consequently, the garbage instructions interleave with actual bytecode instructions, leading to a very heavy burden for reverse engineers to distinguish them from normal bytecode instructions. The most obvious example is that Tigress generates a huge amount of garbage bytecode by turning on the bogus function option.

Stealth Mode. This VM diversification feature decides where the bytecode and VM are placed in the obfuscated program. One popular approach is to create a new, separate segment in the executable file and then place the VM in this segment. By default, Code Virtualizer creates a segment named "`.vlizer`" and VMProtect creates "`.vmp`" segment to store the VM data. VMProtect lets users customize the section name.

Placing a VM in such a separate segment directly exposes the VM location in front of attackers. Therefore, Code Virtualizer supports a diversification feature named "stealth mode", which lets users specify an area in the source code for placing the VM. This area is compiled into the `.text` segment with other normal functions. A detailed example as shown in Figure 7. The VM is placed inside the function `StealthArea` (line 3–13). The macros `STEALTH_AREA_START` (line 5) and `STEALTH_AREA_END` (line 12) mark the start and end of the stealth area. Between them is a sequence of the macro `STEALTH_AREA_CHUNK` as placeholders (line 7–10). Usually, the size of one chunk is 4KB. In practice, the number of chunks is decided by how much memory the VM uses. For example, Fish White uses 120KB of memory, requiring at least 30 chunks to reserve enough space. Notably, the function `StealthArea` must be used in the program so that compiler optimizations do not remove it during compilation. The recommended way is to create an always-false condition and then call `StealthArea` under this condition, as shown in line 17. Code Virtualizer can hide a VM by mixing it with regular program code inside the `.text` section. Tigress directly inserts VMs into the source code, which is stealthy by default.

3.3. Bottom-level: Handler

At the bottom-level, handlers are the concrete machine code running to implement the actual program behavior. We have observed that modern virtualization obfuscators introduce various methods to diversify handlers against reverse engineering.

Encryption. Because handlers represent the real program logic, some sensitive data may appear during the execution, e.g., a magic number or encryption key. VMs adopt encryption to hide the intermediate data in handlers. All

```
1  #include "VirtualizerSDK.h"
2
3  void StealthArea(void)
4  {
5      STEALTH_AREA_START
6
7      STEALTH_AREA_CHUNK
8      STEALTH_AREA_CHUNK
9      STEALTH_AREA_CHUNK
10     STEALTH_AREA_CHUNK
11
12     STEALTH_AREA_END
13 }
14
15 int main(void)
16 {
17    if ((void*)&MessageBox == (void*)&Sleep)
18       StealthArea();
19 }
```

Figure 7: Using stealth mode in Code Virtualizer.

critical values stored in memory are encrypted, so attackers cannot find them by taking a memory snapshot. When the obfuscated program runs, the value is decrypted just before use and then encrypted again after use. These encryption are light-weighted because they are very frequently called. Two examples are shown in Figure 8. The first example in Figure 8(a) adds and subtracts a constant key. The encryption reads the plaintext from memory `[esi]`, adds an immediate value as the key, and saves the ciphertext to memory `[edi]`. Decryption is the reverse process. The constant key is generated during the obfuscation time and hard-coded into the obfuscated program. This encryption/decryption only involves one instruction and hence is a very light-weighted form. It is leveraged to encrypt frequently used data, such as virtual registers. Figure 8(b) shows a more complex example, which involves two keys stored in the memory. The first key is stored in `[ebp+0xAC]`, which is `xor` with the plaintext in `edx`. Then the result is further subtracted by the second key in `[ebp+0xA4]` to produce the final ciphertext. Unlike the previous type, the two keys here are initialized when the VM starts running, so the keys can be different in every execution. Therefore, this encryption is stronger than the first type. On the other hand, it involves more instructions and memory accesses, leading to heavier overhead, so it is mostly used for less frequently used data like the data in virtual memory.

Tigress hides the sensitive data via a variety of encoding methods: encode literals, encode data, and encode arithmetic. They replace integer or string literals with complex but equivalent expressions. For example, it utilizes linear algebra and modular arithmetic to encode integer multiplication `a*b`.

```
a*b=-757949677*(((3537017619*((1789355803*(1789355803*b
    +1391591831))-3670706997*(1789355803*a+1391591831))
    -3670706997*(1789355803*b+1391591831))+3171898074)
    -3670706997;
```

Handler Mutation. This feature substitutes the code inside a handler with an equivalent form. For instance, indirect jumps are replaced by return instructions, like `jmp eax`

```
// Encryption
mov     ecx, [esi]
add     ecx, 0x25195DC
mov     [edi], ecx

// Decryption
mov     eax, [edi]
sub     eax, 0x25195DC
mov     [esi], eax
```

(a)

```
// Encrypt edx
mov     ecx, ebp
add     ecx, 0xAC
xor     edx, [ecx]
mov     ecx, ebp
add     ecx, 0xA4
sub     edx, [ecx]

// Decrypt ecx
mov     ebx, ebp
add     ebx, 0xA4
add     ecx, [ebx]
mov     ebx, ebp
add     ebx, 0xAC
xor     ecx, [ebx]
```

(b)

Figure 8: The two commonly used encryption schemes in Code Virtualizer.

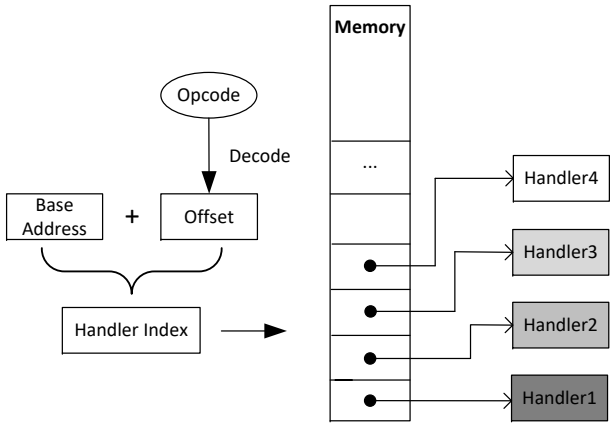


Figure 9: The handler jump table in Code Virtualizer.

can be replaced with `push eax; ret`. The instructions involving registers are substituted by stack operations, like `mov edx [edx]` are equivalent with `push [edx]; mov edx [esp]`.

Another mutation method is junk code insertion. Obfuscators create junk code from simple templates and apply equivalence substitution and code disorder. For example, Code Virtualizer commonly breaks one basic block into multiple blocks connected by junk direct jumps. Calculating values that are never used afterward is another typical junk code pattern. For example, the calculation result is stored in a register, but the register is never used before a new value is written into the register. Tigress inserts opaque predicates and places junk code in the infeasible branch.

Jump Table. A common way to organize handlers is to index them by a jump table. For example, the addresses of all handlers are stored in a contiguous memory range, as shown in Figure 9. Normally, the VM first decodes the offset from the bytecode, adds the offset to the base address of the memory range, fetches the handler's address, and then transfers control flow to the starting address of the handler.

VM diversification shuffles the jump table and places handlers at different addresses. Note that not only the handlers in the jump table can be shuffled, but also the starting address of the jump table can be relocated anywhere in the memory. Therefore, every VM instance has a different jump table. This diverse feature disables the effort of reusing the jump table when reversing multiple VMs.

4. Methodology

This section explains the techniques behind our study. Overall, we first run the obfuscated program with appropriate inputs and record an execution trace. Next, We apply dynamic analysis to recognize the VM parts and their components. Then fine-grained analysis is performed to further identify the specific VM diversification features.

4.1. Recover Key VM Components

We have observed that VMs share some common behaviors, such as high-frequency called snippets and linear access to a memory range. These behaviors are closely related to the major VM components. For example, high-frequency called snippets are likely to be handlers and linear access to a memory range could be reading bytecode. Therefore, tracking these behaviors is the anchor for detecting the key VM components. More specifically, our dynamic analysis focuses on the following behaviors in virtualized programs.
Frequently Executed Program Snippet. Frequently executed snippets are a representative attribute of virtualization obfuscation. They are more intensively executed than other program parts. Usually, frequently executed snippets are strongly related to handlers or dispatch components, especially when they end with an indirect jump instruction. These snippets are regarded as a starting point for analyzing virtualized code.
Linearly Access to Memory Range. In virtualized programs, one essential behavior is linearly accessing a memory range, namely accessing a memory range using a base address and offset, where the base stays the same and the offset changes. This behavior is typically connected to the memory area where the VM bytecode resides. In Figure 1, an example is accessing the opcodes and operands in bytecode instructions.

Another linear access behavior is reading a linked list, i.e., reading data using `[base+offset]` and using the data as the new base address. This behavior is crucial for identifying bytecode interpretation. These linear access patterns are naturally related to the bytecode structure. When the VM is initialized, a list of bytecode is placed in a particular memory area. Then the VM sequentially accesses the bytecode area during the VM execution, reads the bytecode, and interprets their behaviors.
Bytecode Branch Jumping. When interpreting the bytecode, branch jumping is the core component among all VMs. It jumps to different handlers based on the bytecode opcode, as shown in the left part of Figure 1. In practice, most

bytecode jumps are implemented as a jump table, whose index is calculated from the opcode, and the content is the address of the corresponding handler.

VM Context. VM context refers to the run-time environment for a VM, which usually includes virtual registers and memory, as shown in the middle part of Figure 1. Notably, it maintains the virtual program counter (VPC), pointing to the current bytecode instruction. Analyzing the VM context highly benefits understanding the underlying mechanism.

One observable behavior is the "context switch" when the program switches from one VM to another or between native program execution and VM execution. If the switch happens between VMs, the data from one VM context are copied to another one. If the switch happens between the native environment and a VM, all data in the actual registers, like RAX are transmitted to the virtual context in VMs. These behaviors provide a valuable hint to recover the details of the VM context, so we track them to recover the specific inner implementation of the VM context.

Dependency Analysis. Finding all the instructions that depend on another instruction is crucial in our study. We use two types of dependency analyses: data dependency and control dependency. Data dependency discovers the "write-read" relation between instructions, i.e., the previous instruction writes a value that a later instruction reads. Control dependency discovers the path constraints of an instruction. All these dependency analyses can be calculated as forward or backward. In practice, these analyses are often used in a hybrid way. A common scenario is to identify the bytecode location from a handler code. We first use backward control dependency to find the dispatch condition. Next, we follow the data dependency to get the bytecodes that set up the condition.

Encryption Detection. For the simple encryption functions used in VMs, we extract their patterns and remove them. Note that the encryption keys can be recovered from the trace because the trace records all run-time data, such as the registers and memory contents. For more complex encoding schemes like Mixed-Boolean-Arithmetic transformation [67], a technique that converts normal arithmetic calculation to a mixed form of Boolean logic (e.g. $\wedge$, $\vee$, $\oplus$) and arithmetic operations (e.g. +, -, $\times$). We adopted the techniques used in previous reversing works [68], [69] to analyze them.

4.2. Implementation

We implement a set of analysis tools to extract the VM components mentioned above based on dynamic trace recording and static disassembly output. Our tool, named VM-Doctor, contains 2,000 lines of Java, 300 lines of Python, and 200 lines of C/C++ code. We use IDA Pro 7.7 [70] with its Python API, IDAPython [71], to obtain the static disassembly code, run-time trace, and perform dynamic analysis. The recorded trace includes abundant run-time information, such as the instruction and its memory address, register contents, accessed memory address, and memory content.

Given the trace, VM-Doctor first extracts VM's exit and entry points, jump table, and handlers based on fixed-form instruction sets. Then it finds the frequently executed snippets and sorts them for checking linear access of a memory area to search bytecode. Moreover, VM-Doctor can identify junk code through pattern matching, which simplifies the VM's structure and eliminates possible interference in the analysis process. VM-Doctor can vastly reduce the tedious burden and boost efficiency for manual analysis, saving security analysts from reading thousands of lines of assembly code.

Open Source. The source code and analysis materials are publically available at https://github.com/softsec-unh/VM-Doctor.git.

5. Analysis

We pinpoint the VM diversification techniques used by mainstream virtualization obfuscators in Figure 10. This section thoroughly explains the details under the hood of each obfuscator. We totally extract 27 VM diversification features from cutting-edge VM-based obfuscators. Figure 11 shows the diversification features implemented by the different VM variants. The greater number of features present in a VM variant, the higher associated complexity it is. Code Virtualizer's VM variants have the most number (16) of diversification features on average.

5.1. VMProtect

VMProtect is relatively simple compared to the other two obfuscation tools. Its VM architecture is RISC and stack-based. VMProtect uses static VM construction because all obfuscated programs use the same VM template. We can identify a clear dispatch-loop interpretation. It supports multiple-VMs by allowing users to insert various macros into different locations.

VMProtect uses a contiguous bytecode arrangement. In particular, VMProtect keeps a crypto key in the EBX register for decrypting bytecode opcodes and operands. The key is updated for every handler during run-time. We also figured out the unique usage of other registers. EBP stores the top of the virtual stack. ESI contains the virtual program counter (VPC). The base address of VM context is saved in EDI.

VMProtect supports two diversified VMs: normal and ultra. The former applies simple virtualization, and the latter includes mutations. We compare two VMs and discover that the entire VM structures are similar. The mutations mainly happen on the handler level, involving dead store code, opaque branching, jump obfuscation, and code duplication.

5.2. Tigress

Tigress provides virtualization as the main obfuscation method and a variety of other options that can be used together with virtualization. Tigress has recommended several VM configurations as recipes (R2 - R4). For a clear

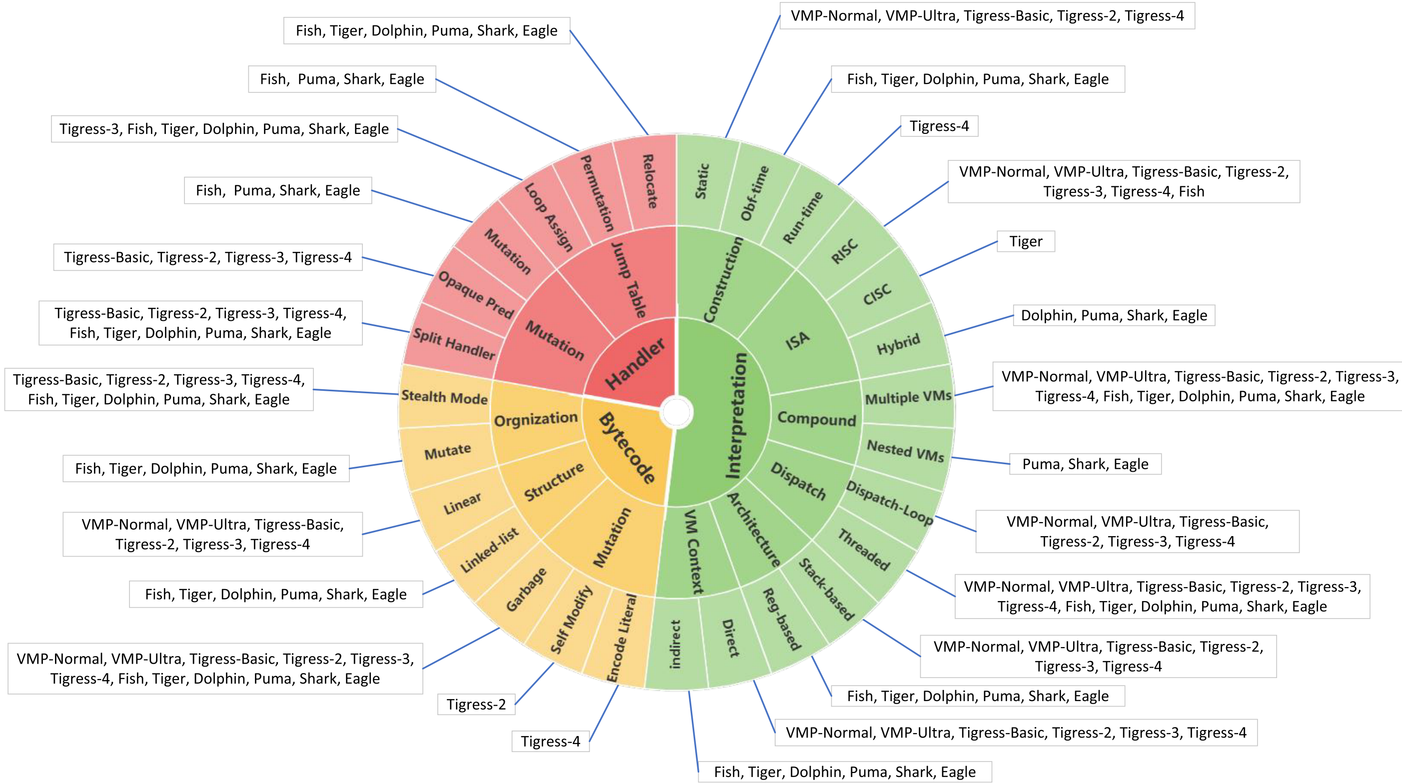


Figure 10: The diversification techniques used by VMProtect (Normal, Ultra), Tigress (Basic, Recipe 2-4), and Code Virtualizer (Fish, Tiger, Dolphin, Puma, Shark, Eagle).

comparison, we include a basic recipe that only enables basic VM obfuscation.

Basic Recipe. This basic configuration implements a static VM template with a clear VM structure. Unlike Code Virtualizer and VMProtect, which mainly use one dispatch method, Tigress supports various dispatch flows. For example, the traditional dispatch-loop includes different implementations via branch and call instructions. The threaded code also provides various implementations, such as direct, indirect, and linear/binary search. The basic recipe also incorporates bytecode and handler-level diversity features such as instruction handlers obfuscation, VPC obfuscation, bogus functions, and implicit flow. Conditional jumps with opaque predicates are commonly found inside handlers.

Recipe 2. R2 employs self-modification on the handler level so that a handler can change itself at run-time. Specifically, it translates the indirect jump instructions to direct jumps using a fixed x86 assembly pattern. This technique helps Tigress circumvent those deobfuscation techniques relying on detecting the indirect jumps, but the drawback is that the number of patterns is very limited.

Recipe 3. This recipe leverages the JIT compiling diversify VMs dynamically. The MyJit library [72] is employed for generating and executing machine code at run-time, which means the VM is dynamically constructed in run-time memory. Our analysis uses techniques similar to tracking packed software. We identify the OEP (Original Entry Point) and recover the whole VM execution afterward.

TABLE 2: The handlers and jump table statistics from Fish/Tiger/Dolphin VMs. "N" means the number of handlers. "Avg Length" is the average length of every handler measured by the number of instructions.

VM	N	Avg Length (Inst. #)	Jump Table Size (Byte)
Fish	150	1,562	600
Tiger	882	206	3,528
Dolphin	334	214	1,336

Recipe 4. R4 adds handler-level diversification methods: function merging and literal encoding. Function merging puts two or more functions into a new function. Literal encoding encrypts integers or char strings to opaque functions that build them at run-time.

5.3. Code Virtualizer

The VMs in Code Virtualizer are named after an animal plus a color. In our observation, the VMs with different colors only include trivial changes mostly within one handler, e.g., the handler in black has more junk code than that in white. VMs with different colors do not vary on the whole architecture or bytecode, so we ignore the colors and only focus on the VMs with different animal names.

All VMs in Code Virtualizer adopt obfuscation-time VM construction. Instead of being hard-coded, every VM is defined by an internal configuration that designates the

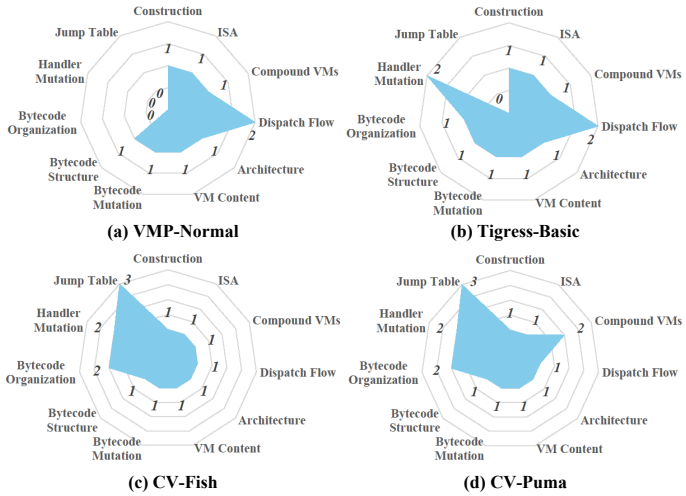


Figure 11: The comparison of VM diversification techniques used by four different VM variants.

features to be enabled or disabled. For example, the Fish VM enables features like "Opcode Permutation" and "Relocate Handlers." When obfuscating a program, Code Virtualizer reads the configuration file and constructs the VM as needed.

Code Virtualizer implements multiple VMs by adding preprocessing macros to different locations inside the C source code. Notably, it allows users to specify VM names in macro to obfuscate different code sections with different VMs.

Fish. Among all the VMs provided by Code Virtualizer, Fish has the lowest complexity and the highest speed. It is a 32-bit VM using 120KB of memory. Code Virtualizer uses a number between 0∼100 to describe the VM's running speed, where 100 is the highest, and 0 is the slowest. Fish White's speed is 90, meaning it is very speedy. Similarly, another number between 0∼100 describes a VM's complexity. The Fish's complexity number is 10, indicating a considerably low complexity.

That said, Fish still keeps regular VM components with several diverse features. First, Fish uses threaded code interpretation architecture. Table 2 calculates the handler statistics showing that Fish has 150 handlers, less than other VMs. Figure 12 compares the handler execution times between various VMs when obfuscating one additional operation. Fish uses around 20 handlers running multiple times. A smaller handler set and jump table with higher execution frequency indicate that Fish is at the CISC end of the VM design spectrum.

The average handler length is 1,562 instructions. We successfully observed permutate handlers, opcode permutation, junk code, and relocate VM context and handlers. Our analysis can successfully identify the handlers and opcodes inside Fish. Also, we found encryption operations at every handler's start and end to protect bytecode operands.

Tiger is a 32-bit VM using 550KB memory with a complexity value of 15 and a speed value of 96. Tiger also leverages a threaded interpretation structure. Tiger is a RISC VM, reflected by its number of handlers and the jump table size. Table 2 shows Tiger has 882 handlers, nearly 6× than the Fish's handler number. Thus, the Tiger's jump table size is significantly larger than the Fish's. The RISC indicates that Tiger has more specialized handlers rather than reusing existing handlers. Figure 12 shows Tiger has lower handler execution times when performing the same operation.

One interesting observation is that Tiger's handlers are shorter than Fish's. We found that Tiger does not apply encryption inside the handlers. It directly loads operands to the VM context as plaintext, performs the calculation, and then saves the result as plaintext as well. This is a trade-off between handler number and performance. Given the larger number of handlers in Tiger, if every handler had built-in encryption, the VM size would expand and lead to high-performance costs.

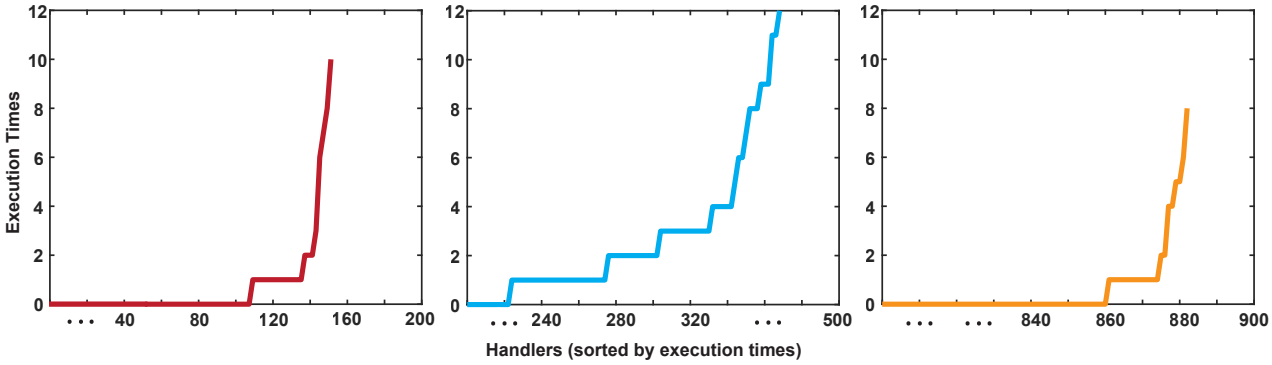
Dolphin takes 207KB of memory with a complexity value of 26 and a speed value of 86. Its interpretation structure is still in threaded code. Table 2 shows that Dolphin has 334 handlers, each of which, on average, contains 214 instructions. These numbers are between Fish and Tiger so that the Dolphin can be viewed as a hybrid VM between RISC and CISC. Figure 12 shows that Dolphin uses more handlers and runs them more than Fish and Tiger for the same operation.

One interesting finding about Dolphin is that it implements a relocatable VM context, e.g., adding a different offset to the original memory location whenever the VM initializes. It hides VM context to impede reverse analysis.

Puma, Shark, Eagle. These three VMs are compound VMs created by nesting two VMs discussed above, i.e., a second VM is used to obfuscate part of the first VM, like handlers. These nested VMs are larger and more complex than the single VMs.

We further identify which exactly VMs are included inside the nested VMs. First, we search for the context switches occurring at the entry and exit of VM execution. Figure 15 in the Appendix shows an example of such operations. Next, we detect the handler jump table and then compare the handlers inside the table. Consequently, we find that Puma is constructed using Fish as the first VM and Tiger as the second VM. Shark is the opposite, i.e., using Tiger as the first VM and Fish as the second. Eagle is constructed from Dolphin as the first and Fish as the second. In most cases, the second VM is used to protect the first VM's handlers.

After detailing the diversification features of each VM variant, we conducted a comparison of four VM variants: VMP-Normal, Tigress-Basic, and Fish, Puma. VMP-Normal and Tigress-Basic recipe both possess their most fundamental virtualization techniques. Fish is a classic VM variant of Code Virtualizer, featuring most of its features, while Puma is a variant based on Fish with nested VM capabilities. We focused on 11 feature categories to highlight the differences in how these VM variants handle diversification. Figure 11 illustrates the varying features of four VM variants. We can see that the more diversification features a VM variant has, the higher its design complexity. Fish and Puma, with 15



(a) Fish (b) Dolphin (c) Tiger

Figure 12: Compare the handler execution times between Fish, Tiger, and Dolphin VMs.

and 16 features respectively, are clearly the more complex VM variants, and they look very similar, as they have applied features in every category without any zero counts. However, Puma includes an additional feature, nested VMs, in the compound VMs category compared to Fish, which significantly increases its complexity. Tigress-Basic is more complex than VMP-Normal because VMP-Normal does not utilize any features in the Jump Table, Handler Mutation, and Bytecode Organization categories. In contrast, Tigress-Basic only lacks features in the Jump Table category.

6. Deobfuscation Enhancement

This section reports our experiences of using the VM diversification knowledge to enhance existing deobfuscation tools. We prefer newer tools with detailed documents and well-organized source code because improving these tools requires a significant amount of work, including reading the source code, analyzing the weaknesses, designing/implementing our patch, and testing the patched tool. Virtual Deobfuscator [24], VMHunt [59], and Syntia [73] represent three different types of VM deobfuscation methodologies (pattern matching, symbolic execution, and program synthesis). Virtual Deobfuscator performs analysis on an execution trace and identifies the instruction patterns for extracting the interpreter's logic and removing the redundant instructions. VMHunt identifies the virtualized parts and extracts the kernel part of the VM based on a program trace recorded by Intel Pin. Syntia splits a program trace into trace windows and feeds random inputs to every window to produce input-output pairs that describe the semantics of each window, then synthesizes a program with the same semantics. Our success in improving them demonstrates that the VM diversification knowledge summarized in this work is general and helpful in enhancing heterogeneous deobfuscation tools.

6.1. Benchmark

A good benchmark is essential to the success of our study, which we envision should carry the following properties.

1) *Objectiveness.* The benchmark programs should be provided by a disinterested third party that also works on obfuscation/deobfuscation.
2) *Diversity.* The benchmark programs should cover a variety of functionalities and code structures.
3) *Ground Truth.* The benchmark programs are easy to understand such that we can verify the deobfuscation result.
4) *Applicability.* The diverse VMs in Code Virtualizer, VMProtect, and Tigress are applicable to these programs since these features are crucial barriers to today's deobfuscation.

According to the above criteria, we identify an obfuscation benchmark from Banescu et al [74]. It is developed by researchers independent of any of the obfuscation providers. It includes a variety of C programs covering basic I/O functions, basic algorithms, and crypto hash functions. Note that Tigress is a source-to-source obfuscator for C programs so using C programs is necessary. The benchmark includes explanations of the programs so the ground truth is easy to verify.

6.2. Virtual Deobfuscator

We enhance Virtual Deobfuscator [24] to handle the Fish, Tiger, and Dolphin VMs. We found that Virtual Deobfuscator cannot give helpful information when processing the programs obfuscated by Fish. Mechanically, it takes an execution trace as the input and clusters the repeat instructions as the VM components. The remaining instructions are regarded as the original program logic. Taking a more careful look at the result, we find the clustering result includes false positives (the instructions belong to normal program logic but are clustered) and false negatives (the instructions belong to VM structure but are not clustered).
Weakness Analysis. The VM diversification knowledge learned from this work helps us quickly determine the reasons for these errors. The false negatives are due to the VM context initialization, which should be clustered as the VM components but falsely recognized as the program logic. Virtual Deobfuscator only regards repeat instructions as VM components, whereas most VM initialization code

only executes once. Take the Fish as an example. Virtual Deobfuscator only correctly clusters 1,752 out of 6,433 instructions for the VM context initialization, missing the rest of 4,681 instructions.

On the other hand, the false positives are because of the threaded interpretation structure adopted by VM diversification. The handlers between the operations for updating VPC and handler jumps are falsely recognized as VM components. Fish VM falsely clusters 50,889 instructions as handler jumps and VPC operations, but these operations only use 7,620 instructions. The false-negative and false-positive rates are shown in Table 3.

Patching. First, we pre-processed the trace using VM-Doctor and marked the VM context initialization and the threaded transfer part. Second, we modify the Virtual Deobfuscator to consider these marks when searching for repeat instructions. For example, it gives a higher priority when searching from these marked instructions.

Our patched Virtual Deobfuscator largely reduces the false-negative and false-positive rates. It correctly recognizes all VM components, including the VM context initialization, VPC operations, handler jumps, and threaded code parts of the dispatch structure, thus lowering the false-negative rate to zero. While the result includes some junk handlers, the false-positive rate drops significantly compared with the original version. Hence this enhancement improves deobfuscation precision and also helps the later analysis steps on Fish, Tiger, and Dolphin. The comparison result is shown in Table 3.

We observe no direct relation between FNR and the VM complexity (Fish has the lowest complexity and Dolphin has the highest). We calculated the FPR/FNR based on the number of instructions from the run trace before or after deobfuscation. Although Fish has the lowest complexity (assigned by the official Oreans configuration file), the program obfuscated by Fish has more instructions on its run trace than those obfuscated by Dolphin.

Overall, False Positives (FPs) occur because the Virtual Deobfuscator mistakenly identifies certain instructions (VPC update instructions, handler jump instructions, and even some garbage handlers) as VM components. The more frequently such instructions are misclassified as VM components, the more FPs there are. Particularly, Dolphin's FP remains relatively high because: (1) Dolphin's VPC implementation is more complex than that of Fish and Tiger, complicating the removal of VPC-related instructions during the patching process; (2) Dolphin inserts fewer garbage handlers than Fish and Tiger in its implementation, which results in fewer instructions being removed than that in Fish and Tiger.

6.3. VMHunt

We enhance VMHunt [59] to handle the nested VMs, i.e., Puma, Shark, and Eagle VMs, because we found that VMHunt has difficulties in processing them. Technically, VMHunt takes an execution trace as input and separates the virtualized snippets based on context saving and restoring instructions. Then, it extracts the virtualized kernel of each snippet and runs multiple granularity symbolic execution to obtain the semantics of the virtualized code. Checking Puma, Shark, and Eagle results, we find that around 32% of the virtualized snippets are irrelevant to the original program logic. Removing redundant instructions will help VMHunt more precisely extract each snippet's virtualized kernel.

Weakness Analysis. Junk handlers are the main factor impeding VMHunt's analysis, especially the virtual kernel extraction. From the VM diversification knowledge, we learn that junk handlers are irrelevant to the original program logic, but VMHunt will treat them as virtualized snippets according to the context switch pattern. Taking Puma as an example, VMhunt extracts 51 virtualized snippets from a trace, but 17 of them are junk handlers. That means VMHunt wastes 1/3 of the time analyzing the junk handlers.

Patching. First, we add more heuristics for pairing context switch instructions. If context saving or storing instructions appear between another pair of context saving and storing instructions, it will be identified as a nested VM type. Second, we encode the junk handler pattern into VMHunt to recognize and remove them. The patched VMHunt can generate a much simpler extraction result than the original version. It correctly removes all the junk handlers from the inner VM. As shown in Table 3, the original version of VMHunt extracts almost 50 virtualized snippets on nested VMs, and 33% of them are detected as junk handlers. Our enhancement can help VMHunt remove over 11 million (1,150,100) instructions on average, and significantly accelerate VMHunt's performance.

6.4. Syntia

We found that the program synthesis in Syntia is very time-consuming, especially for the complex VMs in Code Virtualizer. We use the VM diversification knowledge to analyze and improve Syntia's performance bottleneck. Syntia contains three steps to obtain deobfuscation results. First, the instruction trace is dissected into multiple windows. Second, it generates random input-output pairs to describe the semantics for each trace window. Last, it executes program synthesis to find an expression that maps all inputs to their relevant outputs in each trace window.

Weakness Analysis. The synthesis time in Syntia is heavily affected by the number of trace windows, which are split at indirect branches. From our VM diversification knowledge, we learn that indirect jumps appear a lot inside VMs, wasting Syntia's time on too many trace windows that are actually irrelevant to the real program logic.

Patching. First, we use context switch patterns to separate the virtualized code. Second, we remove the handler jump table and virtual environment initialization content from the virtualized code trace. For the rest of the trace, we dissected it by indirect jumps to get trace windows and then ran the sampling and synthesis. From our experiment, the number of reduced trace windows generated by simplified trace is 90 less on average for Tiger, Dolphin, and Puma, as shown in Table 3. It saves 1,317 seconds as the total synthesis time

TABLE 3: Comparison of three deobfuscation tools before and after our enhancement. For Virtual-Deobfuscator, we compare the false-negative (FN) and false-positive (FP) rate. For VMHunt, we show the number of virtualized snippets and instructions on Puma/Shark/Eagle. For Syntia, we present the number of trace windows and total synthesis time for Tiger/Dolphin/Puma.

Virtual-Dobfuscator		Fish	Tiger	Dolphin
	FNR (Before/After)	72.70% / 0.00%	75.86% / 0.00%	63.04% / 0.00%
	FPR (Before/After)	85.02% / 40.04%	91.67% / 39.02%	93.54% / 76.31%
VMHunt		Puma	Shark	Eagle
	# of VM Snippet (Before/After)	51 / 34	49 / 43	46 / 22
	# of Instruction (Before/After)	3,764,280 / 2,705,520	1,464,806 / 1,285,442	4,240,004 / 2,027,828
Syntia		Tiger	Dolphin	Puma
	# of Trace Window (Before/After)	340 / 290	1,385 / 1,321	2,427 / 2,271
	Synthesis Time(s) (Before/After)	3,808 / 3,190	15,927 / 14,795	27,182 / 24,981

for Syntia. In this case, Syntia's deobfuscation results stay the same, but its performance has been greatly improved.

7. Discussion

Related Work. Code virtualization has attracted many researchers working on deobfuscation. Due to the widely used dispatch-loop-based VM, many previous deobfuscation works focus on this particular type [21]–[25]. For instance, Sharif [22] adopts dynamic binary analysis to detect the dispatch loop and find the mappings between bytecode and related handler functions. However, our result shows that many modern VMs have adopted a variety of interpretation structures, so the dispatch loop cannot be found in most of the VMs, and thus, these lines of approaches do not work on modern VMs.

The other line of research works does not assume the underlying VM structure. They directly process the obfuscated program to analyze the control/data dependencies [26], [27], [57], [58]. For example, Coogan et al. [26] track the instructions that affect system call arguments to approximate the original program. Yadegari et al. [27] rely on forwarding taint analysis to find all instructions influenced by input values. However, our work reveals that modern VMs involve diverse features, intensely confusing the data analysis and thus disabling these lines of work. It is almost impossible to deobfuscate the modern VMs without understanding the underlying VM.

Some other recent studies also started to explore the knowledge of virtualization. Li et al. [75] leverage a special group of Intel instructions as "anchors" to locate and extract the knowledge-related instructions from obfuscated programs. The SoK work [30] simply tests and summarizes the existing deobfuscation methods and shows the limitations and difficulty of automatic deobfuscation. Notably, this SoK work listed 15 deobfuscation tools but actually only tested 4 in their experiments without any enhancements. Nevertheless, these works did not further discuss the diverse VMs and the under-the-hood techniques, which is our work's main contribution.

Non-Virtualization Obfuscations. Since Collberg's pioneering work [76], various obfuscation strategies have emerged. In general, these obfuscation techniques can be categorized from three different aspects: control flow, data, and program layout.

Control flow obfuscation techniques [77] complicates a program's control flow. One of the most common techniques is bogus code insertion [78], which introduces dead or dummy code that is never executed. Opaque predicates [79] is often used for complicating the control flow by inserting unpredictable conditions. Loop transformations (e.g., unrolling, insertion, and modification [80]–[82]) and instruction-level reordering and hiding [83], [84] transform a program's control flow to a more complex form. Inlining [85] and cloning [86] techniques obscure function calls by replacing them with function bodies or creating multiple versions. Self-modifying code [87] adds further complexity at runtime, where polymorphism [88], branching functions [89], and jump table transformations [90] are commonly used.

Data obfuscation targets the program's data structures [91] using techniques like array obfuscation [92], which splits, merges, folds, or flattens arrays [93], [93]–[95]. Variable obfuscation [96], [97] encodes variables, substitute them with functions, or splitting/merging them [98]–[100]. Class transformation [101] confuses data by splitting/merging classes [102], [103] and flattening class hierarchies [103]. Other data obfuscation methods include code substitution [104] and data encryption [105], [106].

Layout obfuscation alters the program's structural presentation [107], mainly by renaming identifiers [108], removing comments, and stripping debugging information [103]. This makes reverse engineering more difficult since key information has been stripped. Techniques like Instruction Set Randomization (ISR) [109], Address Randomization [110], and Address Space Layout Randomization (ASLR) [111] also fits into this category.

Limitations. One weakness of VM-Doctor is that some frequently executed snippets are mistakenly recognized as VM handlers. Currently, this problem is mitigated by adding more heuristics constraints and performing manual analysis. The exact false positive and false negative rates are difficult to calculate because the ground truth of these diverse VM features is unavailable. Another weakness comes from the anti-debugging functions in the obfuscators, which sometimes frustrate the trace module of VM-Doctor. Although

this is out of the scope of this work, incorporating anti-anti-debugging technology will benefit VM-Doctor in handling future VMs.

8. Conclusion

After studying for over a decade, virtualization obfuscation is still recognized as one of the most advanced software obfuscation techniques. This paper conducts the first thorough investigation to reveal the diversification methods behind the VMs inside popular obfuscators. we first conduct an in-depth study to systematize the VM diversification techniques from the perspective of interpretation, bytecode-encoding, and handlers. Our insight is that heterogeneous VMs with various features can break the assumptions in existing deobfuscation works. second, we develop an automated analysis tool to capture and analyze these VM diversification features. Enhanced by our newly revealed knowledge, multiple existing deobfuscation tools obtain significant improvements. In conclusion, our work unveils the crucial VM diversification techniques in the state-of-the-art obfuscators, and paves the way toward effective deobfuscation methods.

Acknowledgments

We would like to thank our shepherd and other reviewers for their insightful feedback. This research was supported by NSF grants 2211905 and 2022279. Jiang Ming was supported by NSF grants 2312185 & 2417055 and Google Research Scholar Award. Jun Xu was supported by NSF grants 2340198, 2319880, and 2213727.

References

[1] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[2] VMware, "VMware Workstation," https://www.vmware.com/.

[3] Oreans Technologies, "Code Virtualizer: Total Obfuscation against Reverse Engineering," http://oreans.com/codevirtualizer.php.

[4] ——, "Themida: Advanced Windows Software Protection System," https://www.oreans.com/themida.php.

[5] Oreans, "WinLicense – Professional software protection and Licensing Management," https://www.oreans.com/WinLicense.php.

[6] VMProtect Software, "VMProtect software protection," http://vmpsoft.com.

[7] C. Collberg, "The Tigress C Obfuscator," https://tigress.wtf.

[8] The Enigma Protector, "Enigma Protector: A professional system for executable files licensing and protection," http://enigmaprotector.com/.

[9] StrongBit Technology, "EXECryptor: Bulletproof software protection," http://www.strongbit.com/execryptor.asp.

[10] DexGuard, "DexGuard introduces code virtualization for Android apps," https://www.guardsquare.com/en/blog/dexguard-introduces-code-virtualization-android, January 2019.

[11] Kiwisec, "KiwiVM: Globally Original VM Protection Solution For Mobile Application," https://en.kiwisec.com/product/vm-android.html.

[12] Jscrambler, "Jscrambler: Security Starts at the Screen," https://jscrambler.com/.

[13] P. OKane, S. Sezer, and K. McLaughlin, "Obfuscation: The Hidden Malware," *IEEE Security and Privacy*, vol. 9, no. 5, 2011.

[14] T. Roccia, "Malware Packers Use Tricks to Avoid Analysis, Detection," https://www.mcafee.com/blogs/enterprise/malware-packers-use-tricks-avoid-analysis-detection/, May 2017.

[15] G. Sparrow, "VMProtect Miner Trojan," https://www.enigmasoftware.com/vmprotectminertrojan-removal/, 2017.

[16] Malwarebytes Labs, "Locky Bart Ransomware and Backend Server Analysis," https://blog.malwarebytes.com/threat-analysis/2017/01/locky-bart-ransomware-and-backend-server-analysis/, February 2017.

[17] L. O'Donnell, "50k Servers Infected with Cryptomining Malware in Nansh0u Campaign," https://threatpost.com/50k-servers-infected-with-cryptomining-malware-in-nansh0u-campaign/145140/, May 2019.

[18] E. Targett, "Chinese Hackers Dropped Rootkit in 50,000 Servers: Then Left Theirs Wide Open," https://www.cbronline.com/news/guardicore-chinese-hackers-servers, May 2019.

[19] P. Paganini, "A new trojan Lampion targets Portugal," https://securityaffairs.co/wordpress/95731/malware/lampion-malware-targets-portugal.html, December 2019.

[20] Veritas, "Reverse Engineering TikTok's VM Obfuscation," https://www.nullpt.rs/reverse-engineering-tiktok-vm-1.

[21] R. Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT'09)*, 2009.

[22] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic Reverse Engineering of Malware Emulators," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P'09)*, 2009.

[23] Y. Guillot and A. Gazet, "Automatic Binary Deobfuscation," *Journal in Computer Virology*, vol. 6, no. 3, 2010.

[24] J. Raber, "Virtual Deobfuscator: Removing virtualization obfuscations from malware," Black Hat USA, 2013.

[25] A. Kalysch, J. Götzfried, and T. Müller, "VMAttack: Deobfuscating Virtualization-Based Packed Binaries," in *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES'17)*, 2017.

[26] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, 2011.

[27] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A Generic Approach to Automatic Deobfuscation of Executable Code," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[28] J. Salwan and Sébastien Bardin and Marie-Laure Potet, "Deobfuscation of VM based software protection," in *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC'17)*, 2017.

[29] J. Salwan, S. Bardin, and M.-L. Potet, "Symbolic deobfuscation: From virtualized code back to the original," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2018, pp. 372–392.

[30] P. Kochberger, S. Schrittwieser, S. Schweighofer, P. Kieseberg, and E. Weippl, "Sok: Automatic deobfuscation of virtualization-protected applications," in *The 16th International Conference on Availability, Reliability and Security (ARES)*, 2021.

[31] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi, "Loki: Hardening code obfuscation against automated attacks," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[32] M. Ollivier, S. Bardin, R. Bonichon, and J.-Y. Marion, "How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections)," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 177–189.

[33] ——, "Obfuscation: where are we in anti-dse protections?(a first attempt)," in *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*, 2019, pp. 1–8.

[34] M. Polychronakis, *Reverse Engineering of Malware Emulators*. Springer US, 2011, ch. Encyclopedia of Cryptography and Security.

[35] R. Manikyam, J. T. McDonald, W. R. Mahoney, T. R. Andel, and S. H. Russ, "Comparing the Effectiveness of Commercial Obfuscators Against MATE Attacks," in *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW'16)*, 2016.

[36] S. Banescu, C. Lucaci, B. Krämer, and A. Pretschner, "VOT4CS: A Virtualization Obfuscation Tool for C#," in *Proceedings of the 2016 ACM Workshop on Software PROtection (SPRO'16)*, 2016.

[37] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009, ch. 4.4, pp. 258–276.

[38] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms," in *Proceedings of the 8th International Workshop on Information Security Applications (WISA'07)*, 2007.

[39] Q. Zhang and D. S. Reeves, "MetaAware: Identifying Metamorphic Malware," in *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*, 2007.

[40] S. Alam, I. Traore, and I. Sogukpinar, "Current Trends and the Future of Metamorphic Malware Detection," in *Proceedings of the 7th International Conference on Security of Information and Networks (SIN'14)*, 2014.

[41] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Proceedings of International Conference on Dependable Systems and Networks (DSN'01)*, 2001.

[42] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL'98)*, 1998.

[43] J. Kinder, "Towards Static Analysis of Virtualization-Obfuscated Binaries," in *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*, 2012.

[44] J. Cazalas, J. T. McDonald, T. R. Andel, and N. Stakhanova, "Probing the Limits of Virtualized Software Protection," in *Proceedings of the 4th Program Protection and Reverse Engineering Workshop (PPREW'14)*, 2014.

[45] ReWolf, "x86 Virtualizer," http://www.openrce.org/blog/view/847/x86_Virtualizer_-_source_code.

[46] C. Taylor and C. Collberg, "A Tool for Teaching Reverse Engineering," in *Proceedings of the 2016 USENIX Workshop on Advances in Security Education*, 2016.

[47] X. Cheng, Y. Lin, D. Gao, and C. Jia, "Dynopvm: Vm-based software obfuscation with dynamic opcode mapping," in *Applied Cryptography and Network Security: 17th International Conference, ACNS 2019, Bogota, Colombia, June 5–7, 2019, Proceedings 17*. Springer, 2019, pp. 155–174.

[48] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–37, 2016.

[49] Tigress, "Tigress Recipes," https://tigress.wtf/recipes.html.

[50] eaglx, "VirtualMachineObfuscationPoC," https://github.com/eaglx/VirtualMachineObfuscationPoC.

[51] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the Semantics of Obfuscated Code," in *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*, 2017.

[52] I. Blumenfeld, R. Faux, and P. Li, "SMT Solvers for Malware Unpacking," in *Proceedings of the 11th InternationalWorkshop on Satisfiability Modulo Theories (SMT'13)*, 2013.

[53] M. Liang, Z. Li, Q. Zeng, and Z. Fang, "Deobfuscation of Virtualization-obfuscated Code through Symbolic Execution and Compilation Optimization," in *Proceedings of the 19th International Conference on Information and Communications Security (ICICS'17)*, 2017.

[54] H. Xie, Y. Zhang, J. Li, and D. Gu, "Nightingale: Translating Embedded VM Code in x86 Binary Executables," in *Proceedings of the 20th Information Security Conference(ISC'17)*, 2017.

[55] F. Desclaux and C. Mougey, "Miasm: Reverse Engineering Framework," RECON, 2017.

[56] J. Salwan, , S. Bardin, and M.-L. Potet, "Symbolic Deobfuscation: From Virtualized Code Back to the Original," in *Proceedings of the 15th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'18)*, 2018.

[57] Z. Tang, L. Wang, K. Kuang, C. Xue, X. Gong, X. Chen, D. Fang, and Z. Wang, "SEEAD: A Semantic-based Approach for Automatic Binary Code De-obfuscation," in *Proceedings of 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'17)*, 2017.

[58] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking," in *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*, 2017.

[59] D. Xu, J. Ming, Y. Fu, and D. Wu, "VMHunt: A Verifiable Approach to Partial-Virtualized Binary Code Simplification," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS'18)*, 2018.

[60] P. Klint, "Interpretation techniques," *Software: Practice and Experience*, vol. 11, no. 9, pp. 963–973, 1981.

[61] J. R. Bell, "Threaded Code," *Communications of the ACM*, vol. 16, no. 6, 1973.

[62] R. B. K. Dewar, "Indirect Threaded Code," *Communications of the ACM*, vol. 18, no. 6, 1975.

[63] B. E. Clark and M. J. Corrigan, "Application system/400 performance characteristics," *IBM Systems Journal*, vol. 28, no. 3, pp. 407–423, 1989.

[64] K. Ebcioğlu and E. R. Altman, "Daisy: Dynamic compilation for 100% architectural compatibility," in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997, pp. 26–37.

[65] A. Klaiber *et al.*, "The technology behind crusoe processors," *Transmeta Technical Brief*, pp. 77–84, 2000.

[66] J. E. Smith, S. Sastry, T. Heil, and T. M. Bezenek, "Achieving high performance via co-designed virtual machines," in *Innovative Architecture for Future Generation High-Performance Processors and Systems*. IEEE, 1998, pp. 77–84.

[67] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *International Workshop on Information Security Applications*. Springer, 2007, pp. 61–75.

[68] N. Eyrolles, L. Goubin, and M. Videau, "Defeating MBA-based Obfuscation," in *Proceedings of the 2016 ACM Workshop on Software PROtection (SPRO'16)*, 2016.

[69] B. Liu, J. Shen, J. Ming, Q. Zheng, J. Li, and D. Xu, "MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security'21)*, 2021.

[70] Hex-Rays, "IDA-Pro disassembler and debugger," https://hex-rays.com/ida-pro/, 2022.

[71] E. Bachaalany, "IDAPython IDA Pro plugin," https://github.com/idapython/src.

[72] P. Krajca, "MyJit Library." http://myjit.sourceforge.net/.

[73] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the semantics of obfuscated code," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 643–659.

[74] S. Banescu, C. Collberg, and A. Pretschner, "Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning," in *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*, 2017.

[75] S. Li, C. Jia, P. Qiu, Q. Chen, J. Ming, and D. Gao, "Chosen-instruction attack against commercial code virtualization obfuscators," in *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.

[76] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," The University of Auckland, Tech. Rep., 1997.

[77] M. Protsenko and T. Müller, "Protecting android apps against reverse engineering by the use of the native code," in *Trust, Privacy and Security in Digital Business: 12th International Conference, TrustBus 2015, Valencia, Spain, September 1-2, 2015, Proceedings 12*. Springer, 2015, pp. 99–110.

[78] J. Macbride, C. Mascioli, S. Marks, Y. Tang, L. M. Head, and R. Ramachandran, "A comparative study of java obfuscators," in *Software Engineering and Applications: Proceedings of the Ninth IASTED International Conference*, 2005.

[79] Z. Wang, C. Jia, M. Liu, and X. Yu, "Branch obfuscation using code mobility and signal," in *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. IEEE, 2012, pp. 553–558.

[80] M. Hataba, R. Elkhouly, and A. El-Mahdy, "Diversified remote code execution using dynamic obfuscation of conditional branches," in *2015 IEEE 35th International Conference on Distributed Computing Systems Workshops*. IEEE, 2015, pp. 120–127.

[81] A. Zambon, "Aucsmith-like obfuscation of java bytecode," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 114–119.

[82] J. Petke, "Genetic improvement for code obfuscation," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, 2016, pp. 1135–1136.

[83] V. Balachandran, N. W. Keong, and S. Emmanuel, "Function level control flow obfuscation for software security," in *2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE, 2014, pp. 133–140.

[84] K. Lu, S. Xiong, and D. Gao, "Ropsteg: program steganography with return oriented programming," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, 2014, pp. 265–272.

[85] C. Collberg, G. Myles, and A. Huntwork, "Sandmark-a tool for software protection research," *IEEE security & privacy*, vol. 1, no. 4, pp. 40–49, 2003.

[86] A. Kulkarni and R. Metta, "A code obfuscation framework using code clones," in *Proceedings of the 22Nd International Conference on Program Comprehension*, 2014, pp. 295–299.

[87] L. Shan and S. Emmanuel, "Mobile agent protection with self-modifying code," *Journal of Signal Processing Systems*, vol. 65, pp. 105–116, 2011.

[88] Y. Sakabe, M. Soshi, and A. Miyaji, "Java obfuscation with a theoretical basis for building secure mobile agents," in *Communications and Multimedia Security. Advanced Techniques for Network and Data Protection: 7th IFIP-TC6 TC11 International Conference, CMS 2003, Torino, Italy, October 2-3, 2003. Proceedings 7*. Springer, 2003, pp. 89–103.

[89] J. Cappaert and B. Preneel, "A general model for hiding control flow," in *Proceedings of the tenth annual ACM workshop on Digital rights management*, 2010, pp. 35–42.

[90] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 290–299.

[91] D. M. Stanley, D. Xu, and E. H. Spafford, "Improved kernel security through memory layout randomization," in *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2013, pp. 1–10.

[92] S. Drape, O. de Moor, and G. Sittampalam, "Transforming the. net intermediate language using path logic programming," in *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, 2002, pp. 133–144.

[93] P. Sivadasan, P. SojanLal, and N. Sivadasan, "Jdatatrans for array obfuscation in java source codes to defeat reverse engineering from decompiled codes," in *Proceedings of the 2nd Bangalore Annual Compute Conference*, 2009, pp. 1–4.

[94] P. Sivadasan and P. S. Lal, "Array based java source code obfuscation using classes with restructured arrays," *arXiv preprint arXiv:0807.4309*, 2008.

[95] W. Zhu, C. Thomborson, and F.-Y. Wang, "Obfuscate arrays by homomorphic functions," in *2006 IEEE International Conference on Granular Computing*. IEEE, 2006, pp. 770–773.

[96] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 54–65.

[97] A. R. Nurmukhametov, S. F. Kurmangaleev, V. Kaushan, and S. S. Gaissaryan, "Application of compiler transformations against software vulnerabilities exploitation," *Programming and Computer Software*, vol. 41, pp. 231–236, 2015.

[98] S. Drape and A. Majumdar, "Design and evaluation of slicing obfuscations," 2007.

[99] F. B. Cohen, "Operating system protection through program evolution." *Comput. Secur.*, vol. 12, no. 6, pp. 565–584, 1993.

[100] S. Cho, H. Chang, and Y. Cho, "Implementation of an obfuscation tool for c/c++ source code protection on the xscale architecture," in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer, 2008, pp. 406–416.

[101] S. Drape, "An obfuscation for binary trees," in *TENCON 2006-2006 IEEE Region 10 Conference*. IEEE, 2006, pp. 1–4.

[102] L. Yang and H.-j. He, "Research on java bytecode parse and obfuscate tool," in *2012 International Conference on Computer Science and Service System*. IEEE, 2012, pp. 50–53.

[103] C. Foket, B. De Sutter, and K. De Bosschere, "Pushing java type obfuscation to the limit," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 6, pp. 553–567, 2014.

[104] S. M. Darwish, S. K. Guirguis, and M. S. Zalat, "Stealthy code obfuscation technique for software security," in *The 2010 International Conference on Computer Engineering & Systems*. IEEE, 2010, pp. 93–99.

[105] A. Kovacheva, "Efficient code obfuscation for android," in *Advances in Information Technology: 6th International Conference, IAIT 2013, Bangkok, Thailand, December 12-13, 2013. Proceedings 6*. Springer, 2013, pp. 104–119.

[106] L. Arockiam and S. Monikandan, "Efficient cloud storage confidentiality to ensure data security," in *2014 International Conference on Computer Communication and Informatics*. IEEE, 2014, pp. 1–5.

[107] C. Collberg, "A taxonomy of obfuscating transformations," Technical Report 148, Tech. Rep., 1997.

[108] M. Batchelder and L. Hendren, "Obfuscating java: The most pain for the least gain," in *International Conference on Compiler Construction*. Springer, 2007, pp. 96–110.

[109] S. W. Boyd and A. D. Keromytis, "Sqlrand: Preventing sql injection attacks," in *Applied Cryptography and Network Security: Second International Conference, ACNS 2004, Yellow Mountain, China, June 8-11, 2004. Proceedings 2*. Springer, 2004, pp. 292–302.

[110] D. C. DuVarney, V. Venkatakrishnan, and S. Bhatkar, "Self: a transparent security extension for elf binaries," in *Proceedings of the 2003 workshop on New security paradigms*, 2003, pp. 29–38.

[111] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming." in *NDSS*, 2015.

Appendix A.

Figure 14 shows an example of using Code Virtualizer to protect a simple C/C++ program. The normal procedure of using Code Virtualizer obfuscation involves three steps.

1) In the source code, marks (e.g., macros in C/C++ code) are inserted around the sensitive areas that need virtualization. As shown in Figure 14, `VIRTUALIZER_START` and `VIRTUALIZER_END` are macros marking the start and end of the obfuscated code. In this case, only `c = a+b` at line 8 will be obfuscated.
2) The source files are compiled using a standard compiler like GCC or Microsoft Visual Studio. Then the object code is linked to a library provided by Code Virtualizer to produce an executable file. Until this step, the marked area has not been obfuscated.
3) Code Virtualizer translates the marked area to byte code and appends the VM to the final obfuscated binary code.

```
[Main Machine Processor]

InternalConversor = 0x31A3BC19
InternalConversorLevel = 1
InternalConversorInProlog = Yes
HideInternalRegs = Yes
UsingMicroRegs = Yes
RelocateRegs = Yes
RelocateStages = Yes
OpcodePermutation = Yes
RelocateHandlers = Yes
JoinUndefinedOpcodes = No
AllowAvidFields = Yes
ExpandedInstructionSet = Yes
MergeStages = Yes
EnableRevirtualization = Yes
EnableJoinHandlers = Yes
EnableStageGarbage = Yes
EnableMicroInstructions = Yes
SmartInstructionsRelocation = Yes
EnableHandlerTimes = Yes
EnableBreakPoints = No
EnableDebugMode = No
EnableInterruptTrace = No
EnableFakeJumps = No
EnableFakeConditionalJumps = No
PermutateHandlers = No
MutateHandlers = No
```

Figure 13: The VM configuration in Code Virtualizer.

```
1  #include "VirtualizerSDK.h"
2
3  int main()
4  {
5    int a = 0x3333, b = 0x4444;
6    int c = 0;
7
8      VIRTUALIZER_START
9      c = a + b;
10     VIRTUALIZER_END
11 }
```

Figure 14: A program obfuscated by Code Virtualizer.

```
1  // VM Entry code
2  jmp  loc_70960E
3  pushf
4  push   esi
5  push   ebp
6  push   edi
7  push   ebx
8  push   edx
9  push   ecx
10 push   eax
11 jz     loc_6A901A
12
13 // VM exit code
14 pop     edi
15 pop     esi
16 pop     ebp
17 pop     ebx
18 pop     edx
19 pop     ecx
20 pop     eax
21 popf
22 retn    0
```

Figure 15: Example of VM entry and exit code.

Appendix B.
Meta-Review

B.1. Summary

Virtualization obfuscation is one commonly used software obfuscation technique. This paper focuses on Virtual Machine diversification techniques deployed inside popular obfuscators. Specifically, the authors analyze VMProtect, Code Virtualizer, and Tigress, extracting 27 obfuscation techniques across 11 categories. Then, they use insights from their analysis to improve multiple existing deobfuscation tools.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The authors promise to make publicly available the source code and analysis material. This is an important step towards further improvements of obfuscation techniques which are used in practice and affect millions of users.
2) The fact that obfuscation is such an adversarial field, with few practitioners publishing papers makes this work really valuable, since its insights and taxonomies can be used as a starting point to evaluate and improve future research in the area.