IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 18, 2020, accepted July 26, 2020, date of publication July 29, 2020, date of current version August 10, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3012684

VCF: Virtual Code Folding to Enhance Virtualization Obfuscation

JAE HYUK SUK AND DONG HOON LEE, (Member, IEEE)
Graduate School of Information Security, Korea University, Seoul 02841, South Korea

Corresponding author: Dong Hoon Lee (donghlee@korea.ac.kr)

This work was supported in part by the Samsung Research, Samsung Electronics.

ABSTRACT Code virtualization, also called virtualization obfuscation, is a code obfuscation technique that protects software from malicious analysis. Unlike code packing or code encryption techniques, code virtualization does not restore the original code on the memory. However, because basic components of the structure are simple, if a virtualization structure is revealed statically, there is a limitation in that the analysis process is somewhat constant. In this paper, we propose *Virtual Code Folding (VCF)* as a new code virtualization technique. The proposed method reduces the amount of virtual code that is statically revealed by folding the virtual code inside a virtualization structure and enables the virtual code to be decoded by generating multiple diversified dispatchers. The folded virtual code is restored by the random key, and then fetched and decoded by the diversified dispatcher. This process makes it possible for *VCF* to effectively obfuscate correspondence between virtual code and handler code (i.e., code that performs real functionality) without significant performance overhead or strong assumptions.

INDEX TERMS Code obfuscation, code virtualization, program diversity, software protection, virtualization obfuscation.

I. INTRODUCTION

As software becomes increasingly important in modern society, intellectual property rights infringements and attacks on program vulnerabilities are becoming more serious. A malicious analyst (i.e., attacker) infringes on the intellectual property rights of software through static analysis and dynamic analysis, and then bypasses the license routine by modifying it. To solve this problem, much current scholarship on delaying reverse engineering focuses on this point and utilizes the technique code obfuscation [1], which transforms a piece of code to make the internal structure difficult to understand while preserving functionality of the original code.

Code encryption [2], an obfuscation technique, encrypts relevant code using a key to protect important routines of the software. The encrypted code must be decrypted and returned to the original code before it is executed. If the attacker does not apply dynamic analysis, the code is encrypted and cannot be analyzed. However, because there is a time when the encrypted code area is decrypted, it is possible for the attacker to find the decryption point of the code by applying dynamic analysis.

Unlike code encryption, code virtualization is characterized by the fact that the original code is not restored in memory. Virtualization obfuscation translates the target code into other types of machine language (i.e., virtual code) that are unfamiliar to attackers. By replacing the original code with self-built virtual code, software developers can hide the logic of critical code and greatly increase the time required for reverse engineering. In addition, because virtualization structures and virtual code are generated randomly, the obfuscated program appears completely different from the original program.

However, because the basic components of a virtualization structure are somewhat fixed, the attacker can restore the functionality of the original program by analyzing the virtual code and handler code together if the structure is revealed statically [5]. In order to prevent this situation, packing or encryption is generally applied when hiding the virtualization structure. In some cases, additional powerful protection techniques (e.g., strong obfuscation) are applied to the structure itself or to the entrypoint, but this increases performance overhead. Although virtualization obfuscation has the advantage of being resistant to dynamic analysis

The associate editor coordinating the review of this manuscript and approving it for publication was Taha Selim Ustun.

techniques, performance overhead is generally large. Therefore, additional performance overhead should be considered when other protection techniques are applied together.

Several related works have been proposed to solve the above problems, but most are techniques that make it difficult to analyze a single virtual code and handler code set. In this case, when the virtual code and dispatcher are statically exposed, an attacker can generate the correspondence table by analyzing the relationship between the virtual code and handler code.

For example, several studies already attempted to encrypt the virtual code to prevent it from being statically exposed. Since the virtual code itself cannot be interpreted by the real CPU, the virtual CPU (i.e., VCPU) is embedded in the target software. If the virtual code is encrypted, it cannot be interpreted or executed by the VCPU, so decryption must precede this process. If the virtual code is then decrypted, all original virtual code could be obtained by an attacker, meaning that the subsequent analysis method becomes tantamount to the existing virtualization structure analysis.

That is, if all the virtual code can be acquired, the subsequent analysis process is constant. Therefore, in order to delay analysis, it is necessary to make it difficult to acquire all virtual code. Likewise, even if all are acquired by an attacker, an additional analysis should be given to the analysis process that is designed within an acceptable performance overhead range.

The proposed technique makes it difficult to acquire the total virtual code, and at the same time, additional performance overhead is lower than similar previous studies. In particular, even if all virtual code is acquired by an attacker, the attacker must analyze a larger volume of relationships in order to understand functionality.

A. CONTRIBUTION

In this paper, we propose *virtual code folding (VCF)* as a technique that can protect a virtualization structure without significant performance overhead or strong assumptions.

In this paper, we propose *virtual code folding (VCF)* that can protect the virtualization structure without a significant performance overhead or strong assumptions.

This paper makes the following contributions:

- In this paper, we propose a code virtualization method using diversified dispatchers. Compared to other related works, there is a lower performance overhead and a similar level of protection. This is possible because the number of virtualization structures is not simply increased, but a new proposed technique called *Folding* is applied.
- The proposed method uses a new technique for concealing (folding) virtual code, which makes it difficult for an attacker to acquire the entire virtual code and analyze the relationship between the handler code and virtual code. It is also different from previous research, in which even a Man-At-The-End (MATE) attacker who could acquire a virtualization structure needed to analyze at least N dispatchers in order to acquire original functionality. In other studies, if a MATE attacker acquired the virtualization structure, the number of analyses of dispatchers that could be reduced was limited from N times to 1 time.

B. PAPER ORGANIZATION

The remainder of this paper is organized as follows: In Section 2, we describe the overview of a basic virtualization structure and its analysis process. Section 3 describes previous research related to enhancing the protection strength of virtualization structures. We present the challenges and overview of *VCF* in Section 4 and Section 5, respectively. Section 6 and Section 7 describe implementation and evaluation. Finally, we provide a discussion and conclusion in Section 8 and Section 9, respectively.

II. BACKGROUND

A. BASIC VIRTUALIZATION OBFUSCATION TECHNIQUE

The virtualization obfuscation technique converts original code into virtual code that a virtualization structure can interpret. The components of the structure for converting, fetching, decoding, and executing virtual code are as follows: [12]

- *Original Code*: A sequence of instructions to be protected in the target program.
- *Virtual Code*: A sequence of instructions converted from the original codes, such that only the generated virtual machine (i.e., dispatcher) can decode it.
- *Handler Code*: A sequence of target machine instructions that is responsible for the actual functionality of the virtual code.
- *Dispatcher*: A component in the virtual machine that is responsible for decoding the virtual code.
- *Virtual Register*: A register used in a virtualization structure that is stored in a specific memory area or stack area.

Virtualization obfuscation is generally applied as follows.

1) The virtualization obfuscator maps the original codes to virtual codes.
2) The virtualization obfuscator maps the virtual codes to handler codes.
3) If the virtualization obfuscator provides the appropriate options, it obfuscates or disorders the sequences of the handler codes.

If virtualization obfuscation is applied to the target program, its execution process is as shown in Figure 1, which is similar to the process of fetching, decoding, and executing on a general CPU. One example for Figure 1 is shown in Figure 2. In Intel assembly language, 0xC3 corresponds to *RETN*, but if it is assumed to be virtualized code, it can be regarded as data that the dispatcher can fetch and interpret.

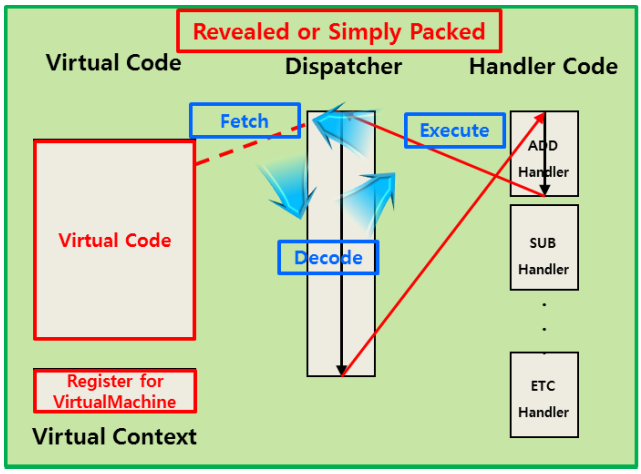


FIGURE 1. Basic virtualization obfuscation and its structure.

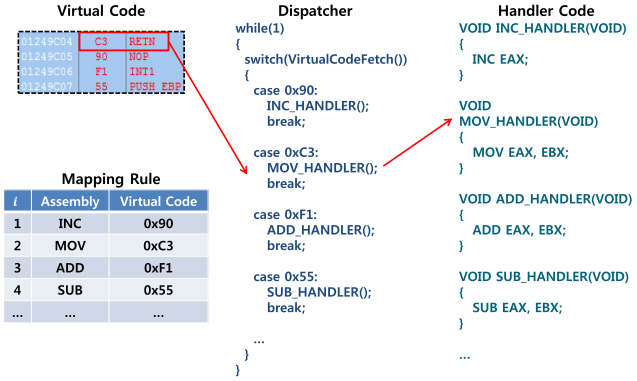


FIGURE 2. Source code level example of basic virtualization obfuscation.

B. ANALYSIS PROCESS OF BASIC VIRTUALIZATION OBFUSCATION TECHNIQUE

Virtualization obfuscation is generally analyzed as follows: [5]

1) Virtual codes are fewer in number and type than original code. Reverse engineering of a virtualization structure should be performed to differentiate between the two.
2) Find the entrypoint to the virtualization structure.
3) Create a disassembler that can analyze the virtual code by analyzing the correspondence relationship between the virtual code and handler code.
4) Convert the virtual code to intermediate representation (IR) using the disassembler created above.
5) Apply compiler optimization techniques to the basic blocks converted to IR.
6) Use the optimized code to create a machine code that can be interpreted by the target processor.

Thus, in order to successfully reverse engineer a program that uses virtualization obfuscation techniques, the relationship between the virtual code and handler code must be analyzed with absolute certainty. In this paper, we propose a technique to prevent the attacker from mapping the two codes by obfuscating the correspondence relationship.

III. RELATED WORKS

In this section, we present related works for the purposed of analyzing virtualization obfuscation and enhancing the protection strength of virtualization obfuscation techniques. Although there are many studies that categorize common code obfuscation techniques [3] or deobfuscation [4] techniques, virtualization obfuscation is more often studied individually because of the specificity of the virtualization structure.

A. ANALYSIS OF VIRTUALIZATION OBFUSCATION

Virtualization obfuscation is one of the strongest protection techniques, and in line with this, an abundance of research on automatic analysis precedes our present paper. Representative works analyzing virtualization obfuscation are as follows.

Reference [5] suggested the common method of restoring virtual codes to original codes in order to reverse engineer the virtualized program. However, no automatic tool was presented in [5], which posited that the virtual code could be restored to the original code by converting the virtual code to IR and applying compiler optimization techniques.

Reference [6] implemented Rotalume, a reverse engineering tool that was able to extract the syntax and semantics of the virtual code from the malware applied virtualization obfuscation along with the control flow graph for behavior analysis. Based on Quick EMUlator (QEMU) [7], this tool executed malware on QEMU to log the instruction executed. Based on the extracted log file, Rotalume analyzed memory access patterns to find the Virtual Program Counter (VPC). Doing this, Rotalume was proven to analyze malware efficiently and effectively.

Reference [8] presented a semantics-based approach to deobfuscation of virtualization structures. To do this, [8] focused on the fact that all types of programs must use standard system resources or interfaces such as an Application Programming Interface (API) offered in the OS. That is, the method proposed in [8] identified the argument value of a system call and analyzed those values. Although malware typically hides by using virtualization obfuscation, it must also use a system call to execute malicious behavior. However, since the origin of the argument values cannot be identified easily due to the difficulty of obfuscated malware analysis, [8] used the above fact with a novel equation reasoning system to analyze the origin of data.

The research described above focused on analysis methodology and prototype implementation of a virtualization structure, whereas research on applying a general program analysis methodology (e.g., taint analysis, symbolic execution, etc.) on virtualization structures has only surfaced recently.

Reference [18] demonstrated the result of reducing 90% of the instruction log of the virtualization structure of the

program that is applied virtualization obfuscation by using static analysis and dynamic analysis. The implementation is available in *github* under the name *VMAttack* and is available as an IDA plugin. Reference [18] has been tested with VMProtect for the evaluation of the implementation results.

Reference [9] proposed a method to analyze the virtualization structure by applying symbolic execution and compiler optimization together. In [9], the extracted symbol and virtual code were converted to C language to remain semantically equivalent to the original program. In this study, PinTool was used as a tool to accurately extract the dispatcher of virtualization structure.

Reference [10] pointed out that virtualization obfuscation applies only to some scopes (e.g., license verification routine, etc.) rather than the entire scope of the program, and as such, suggested a more general and sophisticated virtual code section detection strategy compared to other virtualization structure analysis research. The same work presented *Context Switching* instructions, which move control to the virtualization structure, and suggested an advanced symbolic execution to optimize the execution trace of virtualization obfuscation.

In [11], the result of compiling a program in deobfuscated form by generating and optimizing the Low-Level Virtual Machine (LLVM) IR for an extracted path through a symbolic execution technique is presented. The result of this paper was verified by a case study called Tigress Challenge, but the study did not consider the scenario in which additional protection is applied internally within the virtualization architecture (e.g., virtual code encryption).

The above research studies extracted and analyzed the instruction log to pinpoint the location of the virtualization structure, and then presented the results of correspondence analysis between the virtual code and handler code. There are also published results on *github*.

However, the above works consider finding the virtualization structure important, and the situation in which all the virtual code is not revealed was not a case for analysis. For this reason, it is not appropriate to apply the above methods and techniques in the situation presented in this paper, as the virtual code is only partially exposed within a virtualization structure.

B. ENHANCING THE PROTECTION STRENGTH OF VIRTUALIZATION OBFUSCATION

When a virtualization structure is revealed statically and no additional protection is applied, the relationship between the virtual code and handler code can be analyzed to restore functionality of the original program before obfuscation.

Reference [12] proposed a method to periodically modify the location of virtual registers and delay the analysis of the virtualization structure. To this end, a virtual code called register rotation instruction (RRI) is defined, and the result of modifying the position of the virtual register is presented by periodically executing RRI. However, this has a limitation in that it cannot delay the analysis of the correspondence relationship between the virtual code and handler code.

Reference [13] suggested a method to apply encryption or signatures to delay the analysis of a virtualization structure. The method included seven versions of algorithms, which the authors argued that, as the version rises, a virtualization structure is protected by strong cryptographic primitives. However, the cryptographic primitive used in this technique used a static key and assumed that the corresponding static key could not be accessed by an attacker, which is as a very large assumption in software-protection techniques.

Reference [14] brought diversity to virtualization structures by delaying the analysis and suggesting a random use of the virtualization structure at execution time of the protected program. Because the virtualization structure and executing handler code are different every time, this method is resistant against cumulative attacks. However, because the technique simply generates multiple virtualization structures, performance overhead is significant, and because the protection algorithm is not applied to the algorithm to select the virtualization structure, it becomes the same as the analysis of the basic virtualization obfuscation if the algorithm can be bypassed by a MATE attacker.

In contrast to a delayed study of virtualization obfuscation analysis, [15] is a commercial obfuscation tool. The virtualization obfuscation option provided by [15] is very powerful because it largely applies obfuscation to delay analysis in a virtualization structure and its entrypoint. However, as described above, performance overhead due to the virtualization obfuscation technique itself is large, so there is a limitation in that the maximum amount of performance overhead is additionally applied due to obfuscation.

Reference [16] suggested that the mapping rule can be inferred by the frequency analysis technique because the mapping rule is composed somewhat static, and the result of implementing *DynOpVm* as a technique to solve the above problem is presented. The implemented tool is designed to use the different and dynamic mapping rule for each basic block. It had a different mapping rule according to the control transfer, so it has a resistance to frequency analysis and has the higher entropy. However, since the mapping rule of each basic block was reconstructed at run-time, there is a limitation that if the *DynOpVm* is applied as a strong option, the performance overhead amount can be very large, and the authors said it can be improved.

IV. CONCEPT OF VIRTUAL CODE Folding(*VCF*)

In this paper, we propose a protection scheme called *VCF* that reduces the amount of virtual code statically revealed by dividing the virtual code section into N virtual code blocks (*VCB*, i.e., $VC_{Total} = VCB_1||VCB_2||\ldots||VCB_N$) and partially concealing (i.e., folding) it. In this section, we present the concept of *VCF* and describing the differences between traditional virtual code encryption or packing.

Before describing the traditional virtual code encryption, we first explain how virtual code is generated (mapped) in virtualization obfuscation. The original code ([Original Assembly] in Figure 3. In this example, the operands of assembly are

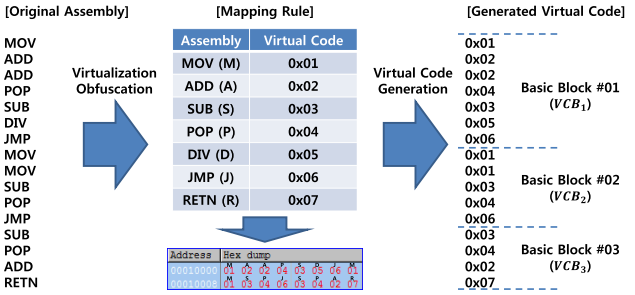


FIGURE 3. Example of virtualization obfuscation & the mapping rule.

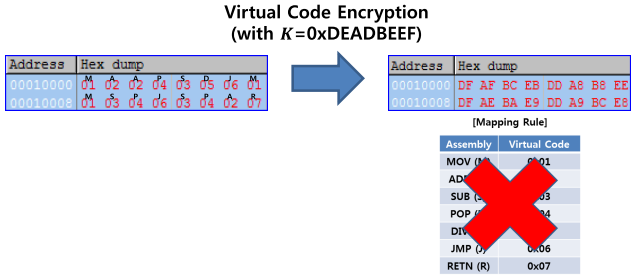


FIGURE 4. Example of virtual code encryption.

omitted.) to be protected is composed of assembly language. When virtualization obfuscation is applied to the original code, it is converted into bytecode by the corresponding table ([Mapping Rule] in Figure 3). The converted bytecode is the virtual code, and an example is shown in Figure 3.

If the virtual code encryption is applied to the above example with a 4-byte key (e.g., K = 0xDEADBEEF), the virtual code is encrypted as shown in Figure 4 with XOR encryption.

As shown in Figure 4, the encrypted virtual code deviates from the [Mapping Rule], and as such, the dispatcher can neither decode nor execute the encrypted virtual code until the virtual code is decrypted. Therefore, virtual code encryption has a limitation in that there is a time when all original virtual code is revealed.

On the other hand, the virtual code folding technique proposed in this paper is a protection scheme performed in basic block units. A basic block is a unit of executable code that ends with instructions that change the control flow (e.g., jump, return, jcc, etc.). As shown in Figure 3, [Generated Virtual Code] consisted of three basic blocks, and in this paper, we call it VCB_i, or i-th Virtual Code Block.

Basically, code or virtual code encryption is widely used to conceal the original code of a program. This is done by encrypting the selected target code section using a random key (i.e., $E_{K_i}(VCB_i)$), and then decrypting the encrypted code section with that key (i.e., $D_{K_i}(E_{K_i}(VCB_i)) = VCB_i$) at execution time. Virtual code folding uses a random key K_i to modify a selected target code section that has not been encrypted but has been executed before (i.e., $M_{K_{i-1}}(VCB_{i-1}) = VCB_i$). (In this paper, $VCB_{i-1} \oplus K_{i-1} = VCB_i$ because XOR is chosen as the unfolding algorithm to minimize performance overhead.) Because the next VCB (VCB_i) to be executed is generated by computing with a random key K_{i-1} and the previous VCB (VCB_{i-1}), *VCF* effectively diversifies the virtual code set. Furthermore, because the next VCB to be executed is overwritten on the previous VCB, the amount of statically exposed virtual code is greatly reduced (i.e., about $1/N$). An example is shown in Figure 5 with K_1 = 0xDEAD, K_2 = 0xBEEF. In this paper, we call the generated VCB as *NVCB*, or newly generated virtual code block (i.e., $NVCB_{i-1} \oplus K_{i-1} = NVCB_i$, $VCB_1 = NVCB_1$).

As shown in Figure 5, the 16-byte virtual code section has been reduced to 7 bytes, which is the longest of the byte lengths of the basic block *VCB* in the entire virtual code section. In *VCF*, the total N dispatchers are pre-generated at protect time, with D_1 being the original dispatcher and $D_i (i \neq 1)$ the diversified dispatchers, and each dispatcher has a mapping rule corresponding with $NVCB_i$.

V. TECHNICAL DETAILS OF VIRTUAL CODE FOLDING

Our *VCF* protection scheme reduces the amount of virtual code statically revealed by dividing the virtual code section into N virtual code blocks (VCBs) (i.e., $VC_{total} = VCB_1||VCB_2||\ldots||VCB_N$) and partially concealing the code. *VCF* can utilize N diversified dispatchers and the *UnFold* handler code defined in this paper to unfold and interpret hidden VCBs (i.e., *NVCB*) as well as and maintain program functionality.

However, *VCF* is not a method of encrypting or decrypting the virtual code section, but rather a method of generating virtual code blocks to be executed next with a random key. Therefore, there are some challenges to designing *VCF*. In this section, we present the challenges and technical details for *VCF* design and implementation.

A. OPERAND DESTRUCTION OF VIRTUAL CODE VC

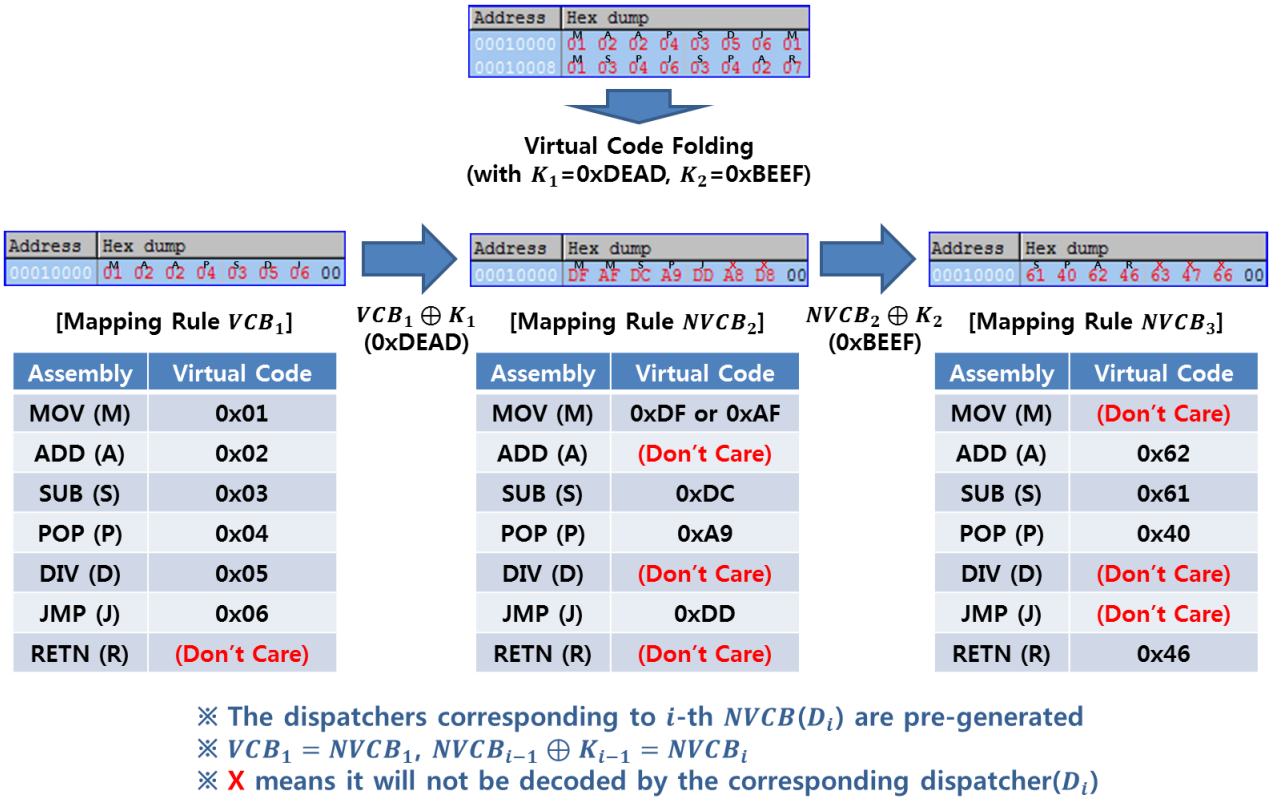
VCF provides diversification to the virtual code section. The i-th single virtual code VC_i is normally configured in the form of concatenation between the i-th virtual instruction VI_i and the i-th virtual operand VO_i (i.e., $VC_i = VI_i||VO_i$) as shown in Figure 6.

Although Figure 6 shows the Intel x86 code, the virtual code is also a set of bytes like this, so we can use this code as an example. We explain the use of the code in Figure 6 as follows. In the Table 1, VIL_i and VOL_i represent the instruction length and operand length of the i-th virtual code.[1]

As shown in Figure 5, when VCB_1 (0×01020204030506), VCB_2 (0×0101030406), and VCB_3 (0×03040207) are given, *VCF* calculates $NVCB_2$ (0xDFAFDCA9DD) and $NVCB_3$ (0×61406246) with random keys K_1 (0xDEAD) and K_2 (0xBEEF).[2] At this point, the functionality of $NVCB_i$ and VCB_i must match. Otherwise, the functionality of the whole protected program is broken. That is, even if $NVC_i \neq VC_i$,

[1] In this paper, the VIL_is generated by *VCF* are all fixed as 1.

[2] Since it is actually an example without operands, it is more appropriate to describe it as a Virtual Instruction Block, or *VIB*, than a *VCB*, but for convenience, it is referred to as *VCB*.



Assembly	Virtual Code
MOV (M)	0x01
ADD (A)	0x02
SUB (S)	0x03
POP (P)	0x04
DIV (D)	0x05
JMP (J)	0x06
RETN (R)	(Don't Care)

Assembly	Virtual Code
MOV (M)	0xDF or 0xAF
ADD (A)	(Don't Care)
SUB (S)	0xDC
POP (P)	0xA9
DIV (D)	(Don't Care)
JMP (J)	0xDD
RETN (R)	(Don't Care)

Assembly	Virtual Code
MOV (M)	(Don't Care)
ADD (A)	0x62
SUB (S)	0x61
POP (P)	0x40
DIV (D)	(Don't Care)
JMP (J)	(Don't Care)
RETN (R)	0x46

FIGURE 5. Concept of Virtual Code Folding (VCF).

TABLE 1. Intel x86 code description.

i	VC_i	VI_i	VO_i	VIL_i	VOL_i
1	83E00F	83E0	0F	2	1
2	8D748108	8D7481	08	3	1
3	8B16	8B16	Null	2	0

```
83E0 0F          AND EAX,0F
8D7481 08        LEA ESI,DWORD PTR DS:[ECX+EAX*4+8]
8B16             MOV EDX,DWORD PTR DS:[ESI]
```

FIGURE 6. Intel x86 code example.

both virtual codes must execute the same handler code. For example, in Figure 5, 0 × 01, 0xDF, and 0xAF all execute the MOV handler code. This illuminates as to why *VCF* must create additional diversified dispatchers. However, because NVC_i is a random value generated by a random key K_{i-1}, the probability that the operand of a NVC_i to be generated will be equal to the operand of a VC_i is very low if the VC_i has an operand. This implies the destruction of the operand, which in turn, leads to the destruction of program functionality.

Therefore, in this paper, we propose a method of folding the i-th virtual instruction block VIB_i by only separating VCB_i into VIB_i and the i-th virtual operand block VOB_i. To this end, the virtual extended operand pointer (VEOP) is defined as a concept against the virtual extended instruction pointer (VEIP) in the virtual register, and the *VI* and *VO* sections are separated. An example of this is shown in Figure 7. Using Figure 5 as an example, $VIB_1 = 0 \times 01020204030506$, $VIB_2 = 0 \times 0101030406$, $VIB_3 = 0 \times 03040207$, $NVIB_2 = VIB_1 \oplus K_1 = 0$xDFAFDCA9DD, and $NVIB_3 = NVIB_2 \oplus K_2 = 0 \times 61406246$. In this manner, *VI* separating and folding can be utilized to avoid the destruction of operands.

B. AMBIGUITY OF NEWLY GENERATED VIB NVIB

When the program protected with *VCF* is executed, K_i is selected after VIB_i executes, and the i-th newly generated virtual instruction block $NVIB_i$ is calculated. At this time, an ambiguous virtual instruction *NVI* can be generated in *NVIB*. The ambiguity mentioned in this paper is a case in which one virtual instruction *NVI* points to two or more types of handler code. For example, in Figure 5, if K_1 was created with 0xDEDE instead of 0xDEAD, both MOV and SUB of $NVIB_2$ would have been set to the same virtual code.

As shown in Figure 8, when an *NVI* of 0xDC appears twice in $NVIB_2$ generated by K_1, the first fetched 0xDC should be

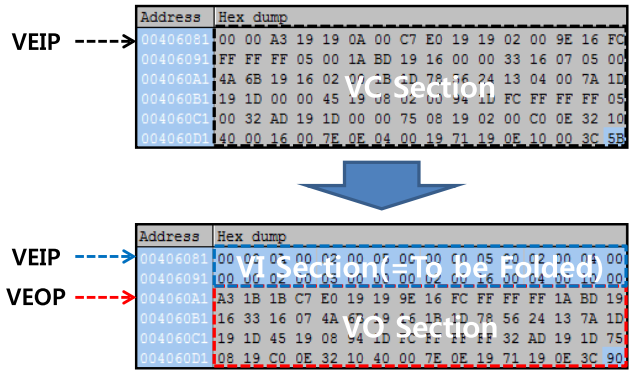


FIGURE 7. Example of *VI* and *VO* separation.

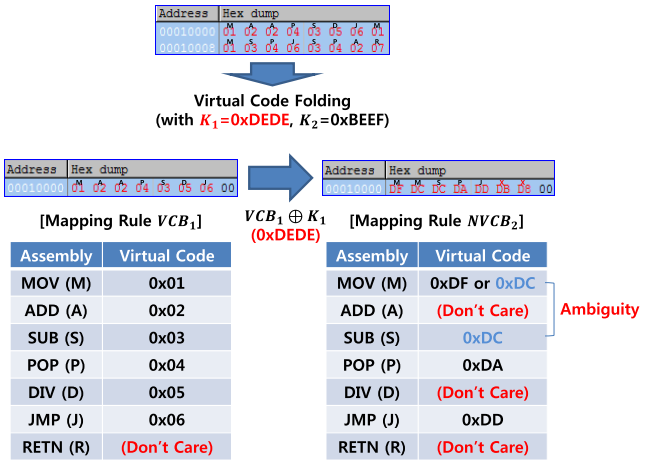


FIGURE 8. Example of ambiguity virtual instruction.

decoded to execute the MOV handler, and the second fetched 0xDC should be decoded to execute the SUB handler. If *VCF* does not take into account the use of the same *NVI* to point to different handler codes, *VCF* can run the wrong handler code and destruct the functionality of the program.

In this paper, we propose a structure that can generate the index table and determine the decoding rules of the *NVI* that has a degree of ambiguity. When the ambiguous *NVI* is fetched, the index table is checked. For example, if the byte fetched from the index table is 0×00, the MOV handler code is executed, and if the byte fetched is 0×01, the SUB handler code is executed. In this case, it can be considered that the value of *NVIL* generated by *VCF* increases by one (i.e., 1byte[0xDC] $\rightarrow$ 2byte[0xDC00 or 0xDC01])

The index table can be generated in advance by analyzing the correspondence between $NVIB_{i+1}$ and VIB_{i+1} before VIB_{i+1} is concealed (folded).

C. BRANCHING INTO CONCEALED NVIB

In *VCF*, there is one $NVIB_i$ that is statically revealed, and all remaining $NVIB_k$s ($k \neq i$) are concealed. *NVIB* is basically divided into basic blocks. The basic block is an instruction sequence with one entrypoint and one exitpoint, and the exitpoint consists of a branch statement, such as jmp, jcc, or call, that moves control to the entrypoint of another basic block.[3] That is, because the last *VI* of $NVIB_i$ ($=LastVI(NVIB_i)$) consists of branch statements, the following cases may occur.

- Case 1) Control (VEIP) moves from the end address of $NVIB_i$ to the start address of $NVIB_j (j \neq i, j \neq i+1)$ inside the *VI* section in the case of $LastVI(NVIB_i) == JMP$ or $LastVI(NVIB_i) == JCC \&\& JumpCondition == TRUE$.
- Case 2) Control moves from the end address of $NVIB_i$ to the start address of $NVIB_i$ inside the *VI* section in the case of a loop statement, it returns to the entrypoint of the block itself.
- Case 3) Control moves from the end address of $NVIB_i$ to the start address of $NVIB_{i+1}$ inside the *VI* section in the case of $LastVI(NVIB_i) == JCC \&\& JumpCondition == FALSE$.
- Case 4) Control moves from the end address of $NVIB_i$ to outside the *VI* section in the case of $LastVI(NVIB_i) == CALL$.

In Case 4, when control (VEIP) returns to the inside of *VI* section with the ret instruction, it can be handled the same way as Case 3. In Case 2, the unfolding process is not necessary because it branches to $NVIB_i$, and the block itself is already statically revealed. The following process is required to move control to $NVIB_j (j \neq i)$ when it is not statically revealed, as in Case 1.

1) **Extract Target Address**
   Extract the branch target relative address ($Rel.Addr$, i.e., Operand of $LastVI(NVIB_i)$).
2) **Calculate Target *NVIB* Index 1**
   Calculate the branch target $NVIB_j$. To do this, *VCF* needs to know the byte length of each *NVIB* ($NVIBL_i$).
3) **Calculate Target *NVIB* Index 2**
   $(End.AddrOf)NVIB_i + Rel.Addr \rightarrow NVIB_j$. The index j can be calculated because *VCF* knows all *NVIBL* values.
4) **Generate Target *NVIB* with Key 1**
   Iterate Step 5 and Step 6 below $j - i$ times.
5) **Generate Target *NVIB* with Key 2**
   Generate K_{i+n} and calculate $NVIB_{i+n+1}$ with $NVIB_{i+n} \oplus K_{i+n} = NVIB_{i+n+1}$. The initial value of n is zero.
6) **Generate Target *NVIB* with Key 3**
   Increase the value of n by one.
7) **Generate Target *NVIB* with Key 4**
   When the iteration of Step 5 and Step 6 is completed, $NVIB_j$ is revealed statically.
8) **Setting VEIP**
   Change the value of VEIP to the start address of the *VI* section ($VISectionAddr$, $\because$ Overwritten).
9) **Setting VEOP**

[3]It can also return to its own entrypoint (e.g., loop statement).

Algorithm 1 *UnFold* Handler Code Algorithm

Input: Index j

procedure UnFoldHandler
  if $LastVI(NVIB_{j-1})==JMP \parallel (LastVI\ (NVIB_{j-1})==JCC\ \&\&\ Condition==TRUE)$ **then** ▷ *Case 1)*
    $NewIndex=CalculateIndex(Rel.Addr)$ ▷ *Rel.Addr=Operand of JMP/JCC*
    for $i = j-1 \rightarrow NewIndex$ **do**
      $K_i=KeyGeration(i)$
      $VirtualInstructionBlockUnFold(VISectionAddr, K_i)$
      if $i \neq j-1$ **then**
        $VEOP+=(VCBL_i - VCBI_i)$ ▷ $VCBL_i$=*Byte Length of* VCB_i & $VCBI_i$=*Instruction Number of* VCB_i
      end if
    end for
    $VEIP=VISectionAddr$
    $DAddr_{NewIndex}=SelectDispatcher(NewIndex)$
    $JMP(DAddr_{NewIndex})$
  else if $LastVI(NVIB_{j-1})==CALL \parallel (LastVI(NVIB_{j-1})==JCC\ \&\&\ Condition==FALSE)$ **then** ▷ *Case 3) & Case 4)*
    $K_{j-1}=KeyGeration(j-1)$
    $VirtualInstructionBlockUnFold(VISectionAddr, K_{j-1})$
    $VEIP=VISectionAddr$
    $DAddr_j=SelectDispatcher(j)$
    $JMP(DAddr_j)$
  end if
end procedure

Increase the value of VEOP by $VOBL_{i+1} + \ldots + VOBL_{j-1} (j \neq i, j \neq i+1$, $VOBL_i$ is the byte length of VOB_i. In the case of $j = i$ or $j = i+1$, skip this step.

Case 2 corresponds to the situation where $j = i$ in the above process, and Case 3 corresponds to the case where $j = i+1$ in the above process. Algorithm 1 is the algorithm of the *UnFold* handler code using the above process.

VI. DESIGN OF VIRTUAL CODE FOLDING

Although the *VCF* presented in this paper assumes Intel x86 programs to be protected, the same methodology can also be applied to other instruction sets (e.g., MIPS, ARM). *VCF* processes the target program as shown in Figure 9.

Because processes before the ''virtual code diversifier'' and after the ''unfold handler generator'' are the same as those for basic virtualization obfuscation, only the virtual code diversifier process and subsequent processes are described in this section.

A. VIRTUAL CODE DIVERSIFIER

The virtualization structure generated at the time of entering this module is similar to the basic virtualization structure shown in Figure 1. This module performs the following processes.

1) This module separates VC_{Total} into VI_{Total} and VO_{Total}.
2) Next, the module defines and creates a VEOP that can point to the *VO* section and saves the start address of the *VO* section. (*VOSectionAddr*).
3) The module divides the VI_{Total} into basic block units. The number of *VIB*s to be divided is called $Num(VIB) = N$.
4) The module calculates $NVIB_2, \ldots, NVIB_N$ using $N-1$ keys[4] $(K_1, \ldots, K_{N-1})$ and $VIB_1 (NVIB_2 = VIB_1 \oplus K_1, NVIB_i = NVIB_{i-1} \oplus K_{i-1}, i \geq 3)$.
5) Next, the module analyzes VIB_i along with its mapping rule table and generates the mapping rule of $NVIB_i$ with the analysis result. For example, if VI_j (0×01) at VIB_i is decoded as executing the MOV handler code originally, and if NVI_j at the same position (index j) in $NVIB_i$ is converted from 0×01 to 0xDC, then it is possible to reconstruct and compensate the dispatcher by utilizing this step, such that the NVI_j (0xDC) executes the MOV handler code as well.
6) The module generates $N-1$ mapping rules by performing Step 5 from $i = 2 \rightarrow N$
7) When ambiguity occurs in the above mapping-rule-generation process, the module creates an index in the index table.
8) The module leaves only VIB_1 and conceals the remaining $NVIB_2, \ldots, NVIB_N$.

B. DISPATCHER DIVERSIFIER & UnFold HANDLER GENERATOR

The virtualization structure generated at the time of entering this module is similar to that shown in Figure 10. This module performs the following processes.

[4]In this paper, the key-generation algorithm is not presented separately, but it is assumed that a random sequence is used.

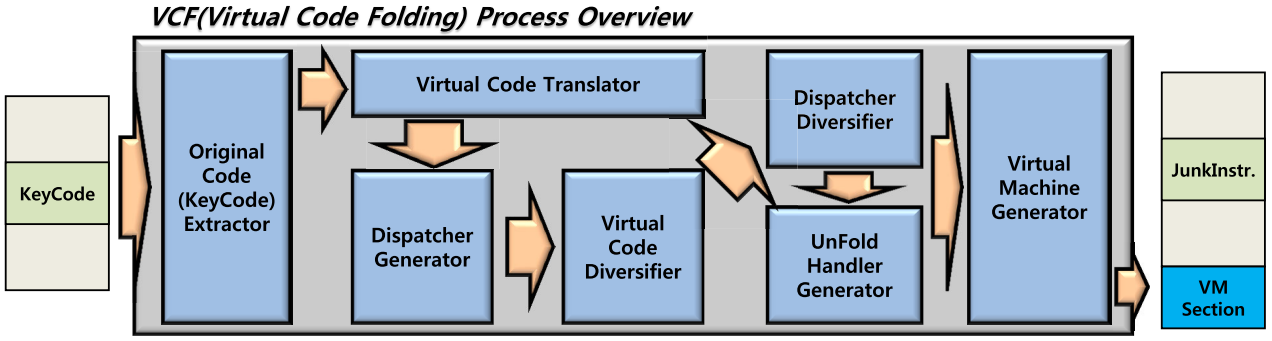


FIGURE 9. *VCF* process overview.

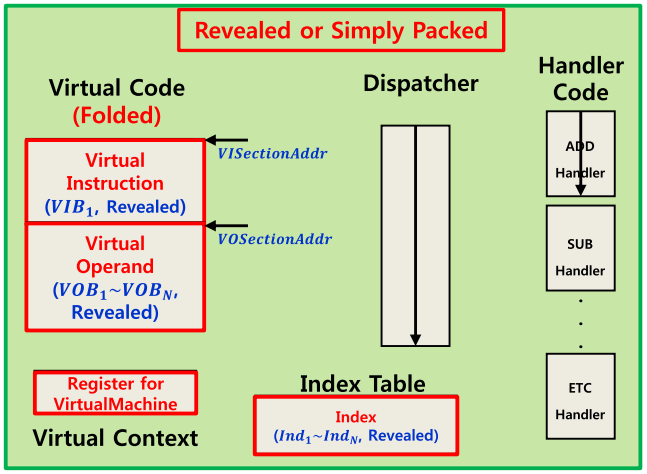


FIGURE 10. Virtualization structure after the virtual code diversifier process.

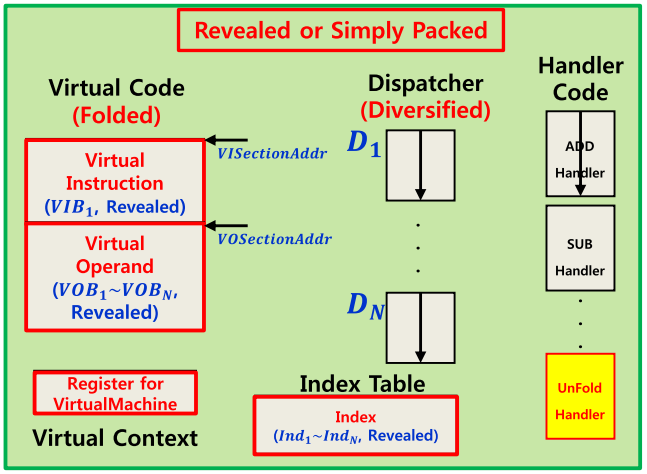


FIGURE 11. Virtualization structure after dispatcher diversifier & UnFold handler generator process.

1) This module generates $N - 1$ dispatchers using the mapping rules generated by the virtual code diversifier module, where D_1 is the initially generated dispatcher and $D_2, \ldots, D_N$ are the newly generated dispatchers.
2) Next, the module defines and generates the *UnFold* handler code that is called immediately after the execution of the branch statement, which calculates the index i of $NVIB_i$ and generates the key K that is necessary to produce $NVIB_i$. After $NVIB_i$ is generated, it returns to the corresponding dispatcher D_i (see Algorithm 1).

The virtualization structure generated as this module finishes is similar to that shown in Figure 11.

VII. IMPLEMENTATION

In this section, we explain the implementation of *VCF* proposed in this paper. *VCF* take as input a 32-bit PE binary and outputs a modified PE binary with VM embedded. *VCF* is based on *x86obf* [17], an open-source and stack-based virtualization obfuscator. It is a tool that can apply basic virtualization obfuscation technique to Windows 32-bit PE files. *x86obf* can obfuscate EXE and DLL files, but users need first specify the range of virtualization by inserting *markers* at the source code level.

In this paper, we modify the source code of the *x86obf* and add the following modules.

- **Virtual Code Separation Module**
  The module that divides VC_{Total} into VI_{Total} and VO_{Total}
- **VEOP Register Generation Module**
  The module that generates VEOP and saves *VOSection-Addr*
- **Virtual Instruction Divider & Dispatcher Diversifier**
  The module that divides VI_{Total} into basic block unit ($VIB_1, \ldots, VIB_N$) and saves information about each *VIB* (such as length *VIBL*) in the target program. Based on that information, it generates diversified dispatcher.
- **Virtual Instruction Block Folding Module** (= *NVIB* **Generation Module)**
  The module that generates the key ($K_1, \ldots, K_{N-1}$) and calculates $NVIB_2, \ldots, NVIB_N$.
- ***UnFold* Handler Code Generation Module**
  The module that generates *UnFold* handler code in the target program.

VCF divides VC_{Total} into VI_{Total} and VO_{Total}, splits VI_{Total} into basic blocks, and then generates diversified dispatchers. Then, *VCF* creates $N - 1$ keys to generate $NVIB_2, \ldots, NVIB_N$. Finally, *VCF* compares the *VIB* set and

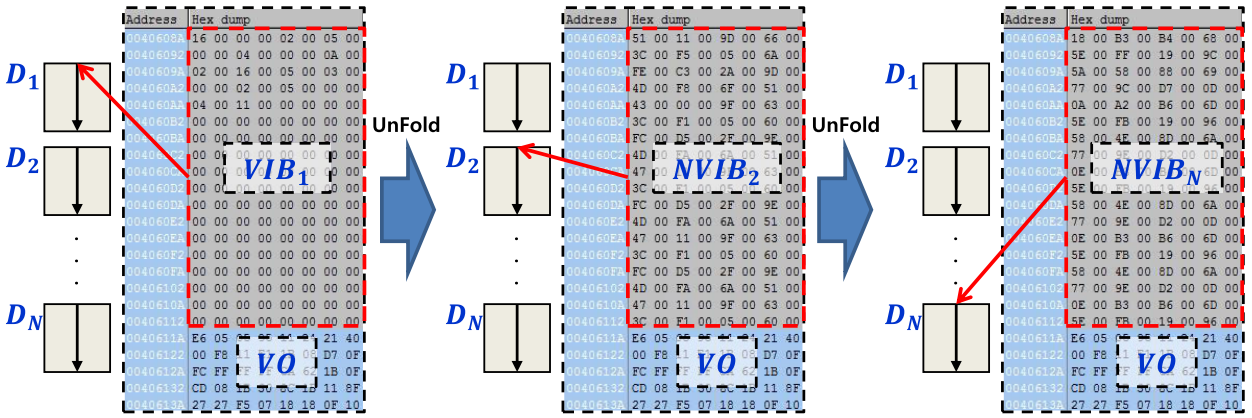


FIGURE 12. UnFold result of *VCF* at runtime.

TABLE 2. Performance evaluation.

Orig: Original, *BV*: BasicVirtualization(*x86obf*), *VMP*: VMProtect, *TMIN*: Themida Min, *TMAX*: Themida Max, *VCF*: **VirtualCodeFolding(Ours)**

	Orig(*ms* & *KB*)	*BV*(*ms* & *KB*)	*VMP*(*ms* & *KB*)	*TMIN*(*ms* & *KB*)	*TMAX*(*ms* & *KB*)	*VCF*(*ms* & *KB*)
HelloWorld	0.1800 (*ms*)	0.2885 (60.28% ↑)	0.6394 (255.22% ↑)	0.2667 (48.17% ↑)	57.08 (31,611% ↑)	0.2936 (63.11% ↑)
(*N*: 2)	7 (*KB*)	17 (142.86% ↑)	745 (10,543% ↑)	2,318 (330,14% ↑)	2,232 (317,86% ↑)	18 (157.14% ↑)
AES	0.2099 (*ms*)	0.4906 (133.73% ↑)	1.0107 (381.52% ↑)	0.6587 (213.82% ↑)	300.1 (142,873% ↑)	0.6856 (226.63% ↑)
(*N*: 22)	19 (*KB*)	31(63.16% ↑)	747 (3,832% ↑)	2,403 (12,547% ↑)	2,342 (12,226% ↑)	45 (136.84% ↑)
Base64	0.2556 (*ms*)	0.5833 (128.21% ↑)	1.3368 (423.00% ↑)	0.9016 (252.66% ↑)	388.2 (151,778% ↑)	0.7705 (201.45% ↑)
(*N*: 26)	9 (*KB*)	21(133.33% ↑)	724 (7,944% ↑)	2,263 (25,044% ↑)	2,117 (23,422% ↑)	37 (311.11% ↑)
StringProcess	0.1992 (*ms*)	0.3539 (77.66% ↑)	0.6861 (244.43% ↑)	0.7138 (258.33% ↑)	140.5 (70,432% ↑)	0.7336 (268.27% ↑)
(*N*: 33)	7 (*KB*)	20(185.71% ↑)	725 (10,257% ↑)	2,363 (33,657% ↑)	2,208 (31,443% ↑)	40 (471.43% ↑)

NVIB set, generates index tables, and removes the *VIB* and *NVIB* sets except for VIB_1. During runtime, *VCF* creates a key with a predefined order and generates (i.e., overwrites) *NVIB* as shown in Figure 12.

VIII. EVALUATION

In this section, we evaluate the performance and security of the implemented *VCF* and describe its result.

A. PERFORMANCE ANALYSIS

Adding diversity to the target program is very effective in lengthening the analysis time for the attacker or any other automated analysis tool. However, generally, as the number of items or, in this case, virtualization structures wanting diversity increases, performance overhead also increases.

Generated test programs are written in C++, and virtualization is applied to functions that are the main of functionality (e.g., cryptographic functions, string modification parts, etc.). *StringProcess* in the evaluation table is a program that adds, cuts, and copies words to a given sentence.

The *VCF* algorithm in this paper has the effect of providing diversity to the *VI* and *D* while minimizing performance overhead. As shown in Table 2, even though an additional 21 dispatchers ($D_2, \ldots, D_{22}$) and 21 virtual instruction blocks ($NVIB_2, \ldots, NVIB_{22}$) are added with diversification, the additional memory usage is less than 15 kB. This is, of course, when compared to a basic virtualization obfuscated program with *x86obf*, not the original program.

This is an improvement in performance when compared with the virtualization structure diversification technique proposed in [14] that provides comparable protection strength. In short, though previous works that protect virtualization structure using *virtualization structure diversification* are rare, [14] is the most recent to apply diversity to virtual code. In this paper, that study is used as a comparison baseline.

In order to provide diversity to a virtualization structure, [14] deformed and diversified not only the dispatcher but also the virtual code and handler code. The protection technique proposed in [14] additionally executed a routine for randomly returns to the dispatcher every time the handler code was executed. This highlights the fact that time-side performance overhead cannot be ignored.

The authors of [14] stated that memory usage increased by at least several dozens of kB when the number of virtualization structures was increased to 10. In addition, performance overhead evaluation results showed that an average runtime overhead of 28 times (i.e., more than 2,700%) that of the original program. On the other hand, the average execution time overhead of our proposed scheme is no more than 3 times (i.e., less than 200%) than the original program execution time. Detailed performance overhead evaluation results of *VCF* are shown in Table 2.

Underlying factors for performance improvement are as follows. In the case of memory usage, our proposed scheme improves on [14], which requires duplicating all elements including virtual code, handler code, and dispatcher, because

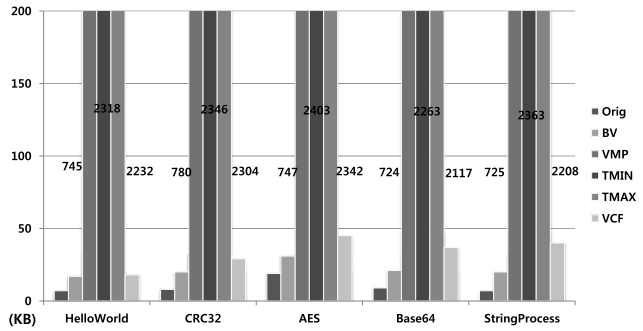


FIGURE 13. Memory overhead.

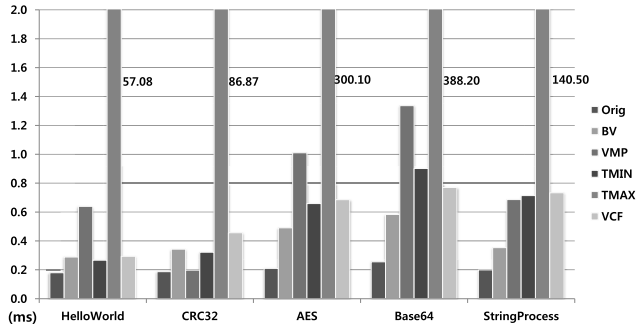


FIGURE 14. Performance overhead.

it reduces the virtual code while applying diversity to the dispatcher. In terms of execution time, our proposed scheme again improves on [14], which applies the protection scheme to the one virtual code unit, because it applies to the virtual code "block" unit. As memory usage and execution time of the *VCF* becomes improved and lightened, an additional protection scheme can be added. This part of the process is described in Section 9 (Discussion).

In order to evaluate performance overhead when applying *VCF*, we show the results of measuring the execution time after applying *x86obf* and *VCF*, Themida (with the fastest virtualization obfuscation option [Red-Tiger] with the strongest virtualization obfuscation option [Black-Eagle]), and VMProtect (with packing option) to various programs as shown in Table 2, Figure 13, and Figure 14.

In Figure 13, it can be seen that *VCF* does not increase the code size as much as the state-of-the-art tools. Thus, if virtualization obfuscation was applied in the same region, both VMProtect and Themida would consume much more memory. This is because both tools basically apply strong obfuscation to entrypoints, inside structures, handler code, and dispatchers to protect the whole virtualization structure.

At first glance, it may seem that there is no significant performance (i.e., execution time) improvement over commercial virtualization obfuscation tools, but it should be taken into account that the virtual code of *VCF* is drastically difficult in terms of analyzing because the folding is applied to the virtual code by the N levels shown in Table 2.

1) PERFORMANCE IMPROVEMENT WITH BLOCK MERGE OPTION

In our proposed scheme, it is also possible to control the N value in a lightweight environment (e.g., Internet of Things, embedded systems). We implemented an additional option that can also merge code block, and this option reduces the value N. Since *VCF* outputs the largest performance overhead in the XOR operation (i.e., *UnFold*) for generating code blocks (*NVIB*), by reducing N, the number of XOR operations can be greatly reduced, thereby improving performance even more.

When the block merge option is applied, the division unit of *VCF* is N' where in $= \lceil \frac{N}{m} \rceil$, m is the value of the block merge option. Results are shown in Table 3.

In Table 3, the $m = 1$ case is the same as applying *VCF*. However, in the $m = N$ case, although an *UnFold* operation (i.e., XOR operation) is never performed, performance is not the same as *x86obf* because of the other performance overhead factors. The values in parentheses indicate performance overhead of the *VCF* when compared to *x86obf*. As the value of m increases, protection strength decreases, and therefore, it becomes essential to set an appropriate m value.

B. SECURITY ANALYSIS

VCF is an additional protection scheme proposed in this paper to enhance the protection strength of virtualization obfuscation. An attacker can restore functionality of an original program when the basic virtualization structure is revealed statically. However, in *VCF*, even if the virtualization structure is revealed, the attacker cannot completely restore functionality because most of *VI* remains concealed.

At least N memory snapshots are required for the attacker to completely acquire the entire *VI* of the program with *VCF* applied. In addition, in order to analyze the corresponding handler code after acquiring the entire *VI*, it is necessary to map to $NVIB_i$ and D_i by analyzing a total of N dispatchers. If the attacker cannot properly match $NVIB_i$ and D_j (i.e., $i \neq j$), program functionality cannot be restored correctly.

1) RESILIENCE MEASUREMENTS WITH VMAttack

For this paper, we applied VMAttack to show the quantified protection strength of our scheme. VMAttack is the result of a study published in 2017 [18], and it can automatically analyze a stack-based virtualization obfuscation structure and output its information, such as the position of the dispatcher, virtual codes, handler codes. Additionally, it is free and open source for use with IDA. *x86obf* is a stack-based virtualization obfuscator, and our proposed scheme is based on it as well, so theoretically, our scheme can be analyzed by VMAttack.

In general, quantitative protection strength evaluation methods are still being studied and are not yet standardized for obfuscation techniques. Therefore, if a new obfuscation technique is presented, the protection strength evaluation method may be somewhat limited. With this in mind, one way is to measure how resilient the new technique is against

TABLE 3. Performance evaluation with block merge option.

	BV(*ms*)	*VCF* [$m = 1$](*ms*)	*VCF* [$m = 3$](*ms*)	*VCF* [$m = 5$](*ms*)	*VCF* [$m = N$](*ms*)
AES(N: 22)	0.4906	0.6856 (39.74% ↑)	0.5584 (13.82% ↑)	0.5272 (7.46% ↑)	0.5163 (5.23% ↑)
Base64(N: 26)	0.5833	0.7705 (32.09% ↑)	0.6875 (17.86% ↑)	0.6382 (9.41% ↑)	0.6041 (3.56% ↑)
StringProcess(N: 33)	0.3539	0.7336 (107.29% ↑)	0.5579 (57.64% ↑)	0.4380 (23.76% ↑)	0.4037 (14.07% ↑)

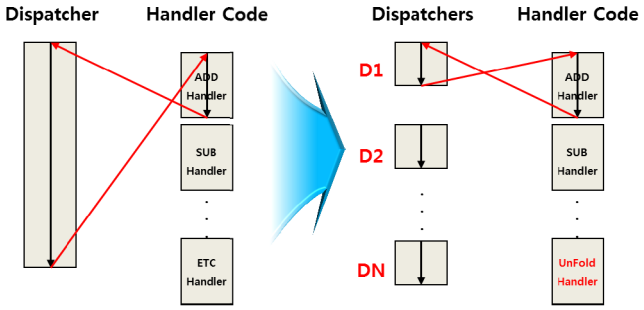


FIGURE 15. Virtualization structure of *x86obf* (left) and *VCF* (right).

previous deobfuscation tools. Therefore, it is more intuitive if quantitative scores can be calculated, but because of insufficient standards, existing analysis tools were used in this paper.

Toy-app is a test program used to show the applicability of VMAttack, as obfuscated with VMProtect, provided by VMAttack with an analysis time of under 3 seconds. The program applying *x86obf* was several times slower analysis time in terms of tens of seconds than the provided test program. Notably, *VCF* took longer than *x86obf* in terms of hundreds of seconds, equating to about 300 $\sim$ 1,000 times more than toy-app. In addition, *VCF* virtualization structure information, which is the output from VMAttack, was incorrect. For example, the location of the dispatcher or virtual code section of the program with *VCF* applied was not extracted correctly. Therefore, we concluded that modification or diversification of virtualization structures creates immunity (i.e., resilience) to automatic virtualization structure analysis tools.

2) COMPLEXITY MEASUREMENT WITH ENTROPY

As a result, it is somewhat confirmed that the resilience of our proposed scheme against VMAttack is strong. However, this result is insufficient, so the results of other protection strength measurement methods are needed. In this paper, we suggest the reason why the analysis of the proposed scheme is complex by using entropy [19].

General software complexity is measured by applying instruction count, nesting count, cyclomatic complexity, and other methods. Although these can be applied to obfuscated software, it is difficult to see the effect on our proposed scheme because *VCF* is not a simple instruction or branch obfuscation but a virtualization obfuscation technique that has a specific structure. Reference [19] is study on software protection strength measurement that is applicable to specific structures, such as control-flow flattening, virtualization obfuscation, and so on. The entropy of the target program $H(P)$ suggested in [19] is as follows.

$$H(P) = \sum_{k=1}^{i} \left(\frac{Instr(BB_k)}{Instr(P)}\right) \times log_2\left(\frac{Instr(P)}{Instr(BB_k)}\right)$$
$$= \sum_{k=1}^{i} p(BB_k) \times log_2\left(\frac{1}{p(BB_k)}\right) = \sum_{k=1}^{i} H(BB_k)$$

Instr is the total number of executed instructions (i.e., [# of instructions] $\times$ [# of executions]), and BB_k is the basic block that constitutes the program P. That is, the uncertainty of the program can be represented by the sum of the basic block entropy. The uncertainty in this paper means that as the number of basic blocks increases due to the increasing branch (i.e., obfuscation), it becomes difficult to guess the execution path of the entire program.

We show the result of measuring the uncertainty of the *VCF* virtualization structure by applying entropy calculation. In this experiment, the entire virtualization structure corresponds to the program P, and the dispatcher and each handler code including *UnFold* handler code correspond to the basic block BB_k.

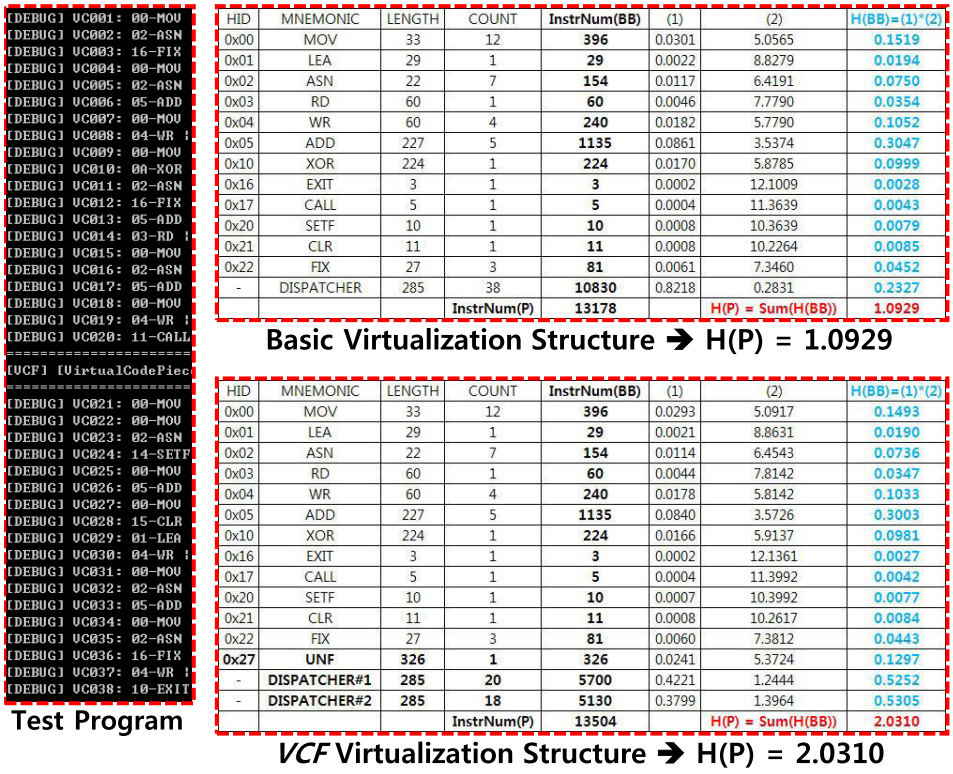
As shown in Figure 15, *VCF* additionally generates as many as *NVIB* (i.e., N) dispatchers along with *UnFold* handler code. Although the number of dispatchers does not affect the total number of executed instructions ($Instr(P)$),[5] the program uncertainty $H(P)$ increases because the number of reachable paths increases. For example, the entropy of virtualization structures *x86obf* and *VCF* with 38 virtual codes and 2 basic blocks is shown in Figure 16.

As shown in Figure 16, the uncertainty of the virtualization structure has nearly doubled. Although entropy certainly increases as the number of dispatchers increases, the number of *UnFold* handler code executions does not significantly affect entropy. Also, because the number of executions of other handler codes such as MOV and LEA remains the same, entropy is not affected.

IX. DISCUSSION

In this section, we describe methodologies that can be applied to further enhance the protection strength of *VCF*. Applying the proposed methodologies can provide an additionally strengthened protective effect but also increase performance overhead. In this paper, we do not show the results of the proposed methodologies.

[5]The total number of dispatcher executions is equal to the number of virtual codes, which is not changed in *VCF*



HID	MNEMONIC	LENGTH	COUNT	InstrNum(BB)	(1)	(2)	H(BB)=(1)*(2)
0x00	MOV	33	12	**396**	0.0301	5.0565	0.1519
0x01	LEA	29	1	**29**	0.0022	8.8279	0.0194
0x02	ASN	22	7	**154**	0.0117	6.4191	0.0750
0x03	RD	60	1	**60**	0.0046	7.7790	0.0354
0x04	WR	60	4	**240**	0.0182	5.7790	0.1052
0x05	ADD	227	5	**1135**	0.0861	3.5374	0.3047
0x10	XOR	224	1	**224**	0.0170	5.8785	0.0999
0x16	EXIT	3	1	**3**	0.0002	12.1009	0.0028
0x17	CALL	5	1	**5**	0.0004	11.3639	0.0043
0x20	SETF	10	1	**10**	0.0008	10.3639	0.0079
0x21	CLR	11	1	**11**	0.0008	10.2264	0.0085
0x22	FIX	27	3	**81**	0.0061	7.3460	0.0452
-	DISPATCHER	285	38	**10830**	0.8218	0.2831	0.2327
			InstrNum(P)	**13178**		H(P) = Sum(H(BB))	1.0929

Basic Virtualization Structure ➔ H(P) = 1.0929

HID	MNEMONIC	LENGTH	COUNT	InstrNum(BB)	(1)	(2)	H(BB)=(1)*(2)
0x00	MOV	33	12	**396**	0.0293	5.0917	0.1493
0x01	LEA	29	1	**29**	0.0021	8.8631	0.0190
0x02	ASN	22	7	**154**	0.0114	6.4543	0.0736
0x03	RD	60	1	**60**	0.0044	7.8142	0.0347
0x04	WR	60	4	**240**	0.0178	5.8142	0.1033
0x05	ADD	227	5	**1135**	0.0840	3.5726	0.3003
0x10	XOR	224	1	**224**	0.0166	5.9137	0.0981
0x16	EXIT	3	1	**3**	0.0002	12.1361	0.0027
0x17	CALL	5	1	**5**	0.0004	11.3992	0.0042
0x20	SETF	10	1	**10**	0.0007	10.3992	0.0077
0x21	CLR	11	1	**11**	0.0008	10.2617	0.0084
0x22	FIX	27	3	**81**	0.0060	7.3812	0.0443
0x27	**UNF**	**326**	**1**	**326**	0.0241	5.3724	0.1297
-	**DISPATCHER#1**	**285**	**20**	**5700**	0.4221	1.2444	0.5252
-	**DISPATCHER#2**	**285**	**18**	**5130**	0.3799	1.3964	0.5305
			InstrNum(P)	**13504**		H(P) = Sum(H(BB))	2.0310

VCF Virtualization Structure ➔ H(P) = 2.0310

FIGURE 16. Entropy of virtualization structure of *x86obf* (top) and *VCF* (bottom).

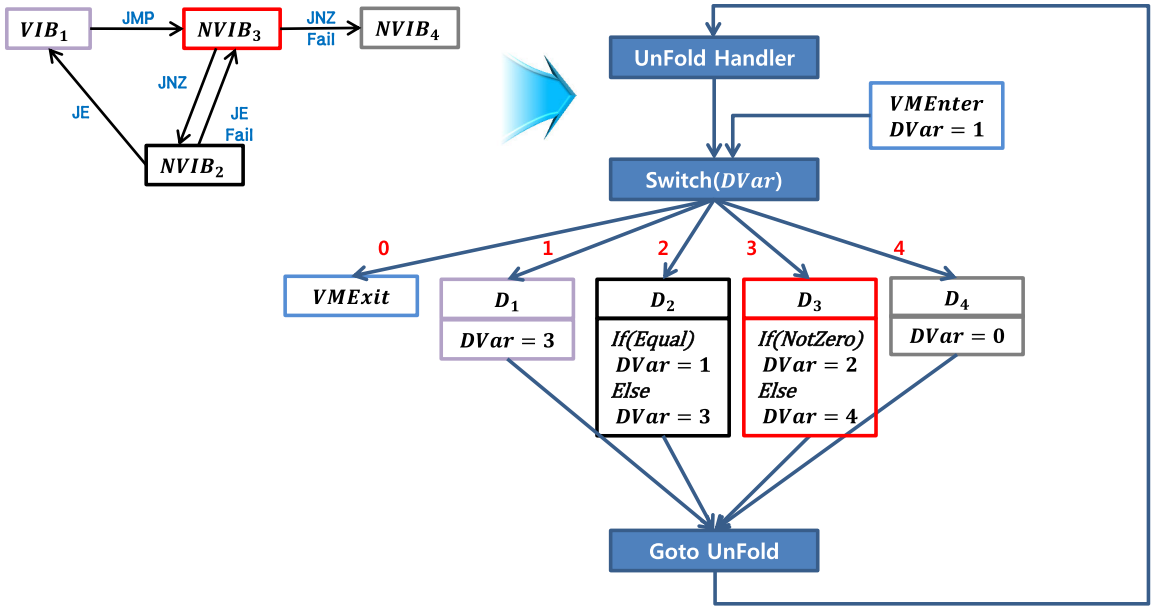


FIGURE 17. Example of *VCF* and its basic control flow.

A. VIRTUAL CODE FOLDING WITH CODE-BASED DYNAMIC KEY

The key K used in code encryption is usually a static key that is stored in the target program. Although the key-generation algorithm is not shown in *VCF*, it is possible to provide an additional protective effect when the dynamic key generated at execution time is used as the unfolding key. In particular, a dynamic key has the effect of providing integrity according to the object used for key generation, and the objects to be utilized are mainly code, time, and flow.

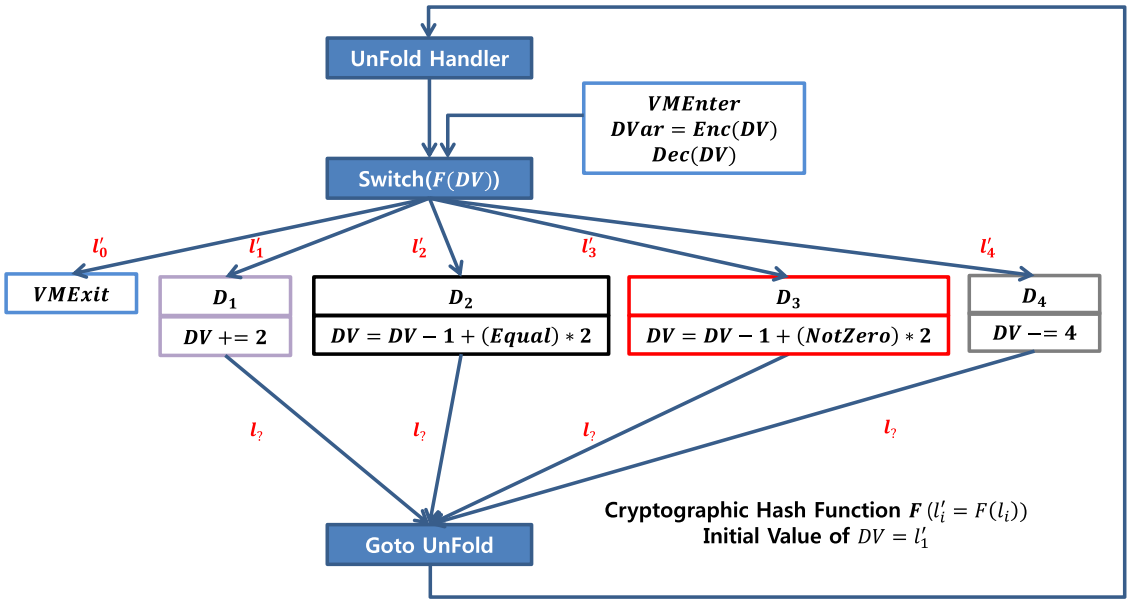


FIGURE 18. Example of *VCF* with control flow flattening [22].

A code-based dynamic key encrypts the software with the hash value of a specific code section. When that software is executed, the decryption routine generates the key with the hash function. If the code section is damaged at execution time, the correct dynamic key cannot be generated, and the software does not decrypt correctly. That is, it effectively provides the integrity of code (i.e., tamper-resistance). A time-based dynamic key [20] also does not generate the key correctly if a large time difference occurs compared to the previously measured execution time threshold. That is, it has the effect of providing integrity of time (i.e., anti-debugging). Furthermore, a flow-based dynamic key does not generate the key correctly if the control flow of the software is damaged or modified compared to the previously stored execution control flow. This means that it can be said to provide integrity of flow (i.e., control flow integrity).

If $N-1$ keys are generated using the hash values of important routines, such as anti-debugging and license-validation routines, an attacker could not modify the code of those routines. If the attacker were to modify those routines, the program would crash because the *VI* would not unfold correctly. By providing tamper-resistance, both static and dynamic analyses by the attacker are simultaneously delayed. Note that performance overhead is increased in this scenario, as compared to using the static key, although the code-based, dynamic key-generation algorithm can be composed of the hash function with low overhead.

B. DISPATCHERS WITH CONTROL-FLOW FLATTENING

The control-flow flattening technique [21] is a control-flow obfuscation technique that makes it difficult to grasp the code control flow of the target program by flattening the control-flow graph and generating the dispatcher. *VCF* is structured by selecting and unfolding the $NVIB_i$ to be executed next in the *UnFold* handler code and selecting the dispatcher D_i accordingly. Figure 17 illustrates an example of when the dispatcher D_i selection algorithm is revealed statically.

As the process of setting the dispatcher variable *DVar* is statically revealed, the data flow analysis can be applied to *DVar* to restore the imperfect but approximate control flow of *VI*. However, applying the control-flow flattening technique proposed by Cappaert and Preneel [22] makes it more difficult to guess the execution order of the dispatcher by static analysis, as the value of *DVar* can be virtually concealed by the branch function in the method and the one-way function F. Figure 18 is an example of *VCF* with this control-flow flattening.

As the process of setting the dispatcher variable (*DVar*) is statically revealed, the data flow analysis can be applied to *DVar* to restore the imperfect but approximate control-flow of *VI*. However, applying Cappaert's control-flow flattening technique [22] makes it more difficult to guess the execution order of the dispatcher by static analysis because the value of *DVar* can be effectively concealed by the branch function proposed in [22] and one-way function F. Figure 18 is an example of *VCF* with Cappaert's control-flow flattening.

In this case, because the analysis of the execution order of dispatchers through static analysis is difficult, the attacker should rely more on dynamic analysis.

X. CONCLUSION

In this paper, we propose *VCF* for the protection of virtualization structures and present its implementation and evaluation results. *VCF* can effectively increase the time needed for an attacker to analysis a system by reducing the amount of

statically revealed virtual code. In particular, since the amount of virtual code can be reduced and diversity can be provided simultaneously, the difficulty of analyzing the program is further amplified.

The proposed method decreases performance overhead with the same degree of protection strength when compared with other research models or commercial tools that provide stronger virtualization obfuscation. Our method is expected to be applicable on platforms with limited power, such as embedded or Internet of Things applications, because a virtualization structure provides additional protection strength with a reduced performance overhead.

REFERENCES

[1] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Dept. Comput. Sci., Univ. Auckland, Auckland, New Zealand, Tech. Rep. 148, 1997.

[2] B. B. Rad, M. Masrom, and S. Ibrahim, "Camouflage in malware: From encryption to metamorphism," *Int. J. Comput. Sci. Netw. Secur.*, vol. 12, no. 8, pp. 74–83, 2012.

[3] C. K. Behera and D. L. Bhaskari, "Different obfuscation techniques for code protection," *Procedia Comput. Sci.*, vol. 70, pp. 757–763, Jan. 2015.

[4] 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 Comput. Surv.*, vol. 49, no. 1, pp. 1–37, Jul. 2016.

[5] R. Rolles, "Unpacking virtualization obfuscators," in *Proc. 3rd USENIX Workshop Offensive Technol. (WOOT)*, 2009, pp. 1–7.

[6] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proc. 30th IEEE Symp. Secur. Privacy*, May 2009, pp. 94–109.

[7] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf.*, vol. 41, 2005, pp. 1–6.

[8] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: A semantics-based approach," in *Proc. 18th ACM Conf. Comput. Commun. Secur. (CCS)*, 2011, pp. 275–284.

[9] M. Liang, Z. Li, Q. Zeng, and Z. Fang, "Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization," in *Proc. Int. Conf. Inf. Commun. Secur.* Cham, Switzerland: Springer, 2017, pp. 313–324.

[10] D. Xu, J. Ming, Y. Fu, and D. Wu, "VMHunt: A verifiable approach to partially-virtualized binary code simplification," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 442–458.

[11] J. Salwan, S. Bardin, and M.-L. Potet, "Symbolic deobfuscation: From virtualized code back to the original," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2018, pp. 372–392.

[12] H. Wang, D. Fang, G. Li, X. Yin, B. Zhang, and Y. Gu, "NISLVMP: Improved virtual machine-based software protection," in *Proc. 9th Int. Conf. Comput. Intell. Secur.*, Dec. 2013, pp. 479–483.

[13] A. Averbuch, M. Kiperberg, and N. J. Zaidenberg, "Truly-protect: An efficient VM-based software protection," *IEEE Syst. J.*, vol. 7, no. 3, pp. 455–466, Sep. 2013.

[14] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, and Z. Wang, "Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling," *Comput. Secur.*, vol. 74, pp. 202–220, May 2018.

[15] *Themida: Oreans.com*. Accessed: Jul. 29, 2020. [Online]. Available: https://oreans.com/

[16] X. Cheng, Y. Lin, D. Gao, and C. Jia, "DynOpVm: VM-based software obfuscation with dynamic opcode mapping," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.* Cham, Switzerland: Springer, 2019, pp. 155–174.

[17] *x86obf: Blog of Zubcic.re*. Accessed: Jul. 29, 2020. [Online]. Available: http://zubcic.re/blog/x86obf-source-code-released/

[18] A. Kalysch, J. Götzfried, and T. Müller, "VMAttack: Deobfuscating virtualization-based packed binaries," in *Proc. 12th Int. Conf. Availability, Rel. Secur.*, Aug. 2017, pp. 1–10.

[19] R. Giacobazzi and A. Toppan, "On entropy measures for code obfuscation," in *Proc. ACM SIGPLAN Softw. Secur. Protection Workshop*, 2012.

[20] K. Lee, S. Kim, and D. H. Lee, "Anti-debugging scheme with time-based key generation," *J. Secur. Eng.*, vol. 10, no. 3, pp. 291–304, 2013.

[21] T. László and Á. Kiss, "Obfuscating C++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, no. 1, pp. 3–19, 2009.

[22] J. Cappaert and B. Preneel, "A general model for hiding control flow," in *Proc. 10th Annu. ACM Workshop Digit. Rights Manage. (DRM)*, 2010, pp. 35–42.

JAE HYUK SUK received the B.S. degree in electrical and computer engineering from Seoul University, Seoul, South Korea, in 2012, and the M.S. degree in information security from Korea University, Seoul, in 2014, where he is currently pursuing the Ph.D. degree in information security with the Graduate School of Information Security. His research interests include software protection, program obfuscation, program deobfuscation, reverse engineering, and malware analysis.

DONG HOON LEE (Member, IEEE) received the B.S. degree from Korea University, Seoul, South Korea, in 1985, and the M.S. and Ph.D. degrees in computer science from The University of Oklahoma, Norman, OK, USA, in 1988 and 1992, respectively. Since 1993, he has been with the Faculty of Computer Science and Information Security, Korea University. He is currently a Professor with the Graduate School of Information Security, Korea University. His research interests include cryptographic protocol, applied cryptography, functional encryption, software protection, mobile security, vehicle security, and ubiquitous sensor network security.