Invalidating Analysis Knowledge for Code Virtualization Protection Through Partition Diversity

GUANGHUI LI, Northwest University
KAIYUAN KUANG, Northwest University
ZHANYONG TANG, Northwest University
DI MA, University of Michigan-Dearborn
DINGYI FANG, Northwest University
XIAOJIANG CHEN, Northwest University
ZHENG WANG, Lancaster University

Code virtualization built upon virtual machine (VM) technologies are emerging as a viable method for implementing code obfuscation to protect programs against unauthorized analysis. State-of-the-art VM-based protection approaches use a fixed set of virtual instructions and bytecode interpreters across programs. This allows an experienced attacker to use knowledge extracted from other programs to quickly attack applications protected under the same code obfuscation scheme. In this paper, we propose a novel VM-code obfuscation system to protect the obfuscated code from reverse analysis using analysis knowledge and make the analyst's job hard. We achieve this by employing the instruction set randomization technique to obfuscate the relationship between the opcodes of bytecode instructions and their semantics. Our approach partitions the protected code region to multiple segments where each segment will be randomized in a different way. As a result, the same opcode of a bytecode instruction can be mapped to different semantics in different parts of the obfuscated code. We evaluate our approach on four real-world applications and compare it against two state-of-the-art VM-based code obfuscation approaches. Experimental results show that our approach provides stronger protection at the cost of little extra overhead.

CCS Concepts: •**Computer systems organization** → **Embedded systems;** *Redundancy;* Robotics; •**Networks** → Network reliability;

Additional Key Words and Phrases: Virtualized obfuscation, reverse engineering, instruction set randomization, analysis knowledge

1. INTRODUCTION

Unauthorized code reverse engineering is a major concern for software developers. This technique is widely used by adversaries to perform various attacks, including removing copyright protection to obtain an illegal copy of the software, taking out advertisement from the application, or injecting malicious code into the program. By making the program harder to be traced and analyzed, code obfuscation is a viable means to protect software against unauthorized code modification [Barak et al. 2001; Lynn et al. 2004; Linn and Debray 2003; Preda and Giacobazzi 2005; Van Oorschot 2003; Wu et al. 2010].

Correspondence authors addresses: Zhanyong Tang, School of Information Science and Technology, Northwest University, Xian, China, 710127.
Author's addresses: G. Li, K. Kuang, Z. Tang, D. Fang, X. Chen, School of Information Science and Technology, Northwest University, Xian, China, 710127; D. Ma, Department of Computer and Information Science, University of Michigan-Dearborn, MI, 48128; Z. Wang, School of Computing and Communications, Lancaster University, UK.

DOI: 0000001.0000001

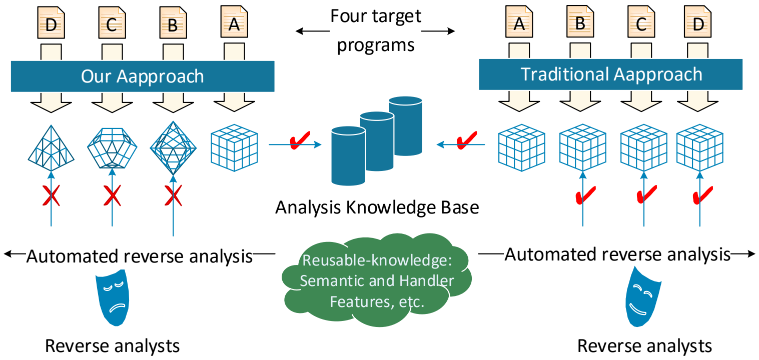


Fig. 1. The process of reusing attacking knowledge for code reverse engineering. Here we have four different target programs, A, B, C and D. In the first scenario, all programs are obfuscated using fixed semantic and structural features. This allows an attacker to reuse knowledge obtained from attacking one program to perform code reverse engineering of other programs. In the second scenario, the protection system ensures that different programs have different semantic and structural features. In this way, the attacker will not be able to reuse the previously extracted knowledge to perform attacks across programs.

Code virtualization based on a virtual machine (VM) is emerging as a promising way for implementing code obfuscation [Oreans-Technologies 2016a; VMProtect-Software 2016; Oreans-Technologies 2016b; Fang et al. 2011; Wang et al. 2014; Wang et al. 2013]. The underlying principal of VM-based protection is to replace the program instructions with bespoke virtual instructions and eventually encode into a bytecode program. These virtual bytecodes will then be translated into native machine code at runtime to execute on the underlying hardware platform. This forces the attacker to move from a familiar instruction set to an unfamiliar environment, which can significantly increase the time and effort involved in the attack.

Reverse engineering of VM-obfuscated code typically follows a number of steps. The attack first reverse-engineers the virtual interpreter to understand the semantics of individual bytecode instructions. The attacker then translates the bytecode back to native machine instructions or even high-level program languages to understand the program logic [Falliere et al. 2009; Rolles 2009]. Among these steps, understanding the semantics of individual bytecode instructions is often the most-consuming process, which involved in analysing the handler that used to interpret every bytecode instruction.

Numerous approaches have been proposed to protect VM handlers from reverse engineering. Most of them increase the diversity of program behaviour by obfuscating the handler implementation [Wang et al. 2014] or iteratively transforming a single program multiple times using different interpretation techniques. However, none of the prior work changes the semantics of the bytecode instruction. Such techniques are vulnerable for programs protected under the same obfuscation technique. In particular, an attacker can reuse the knowledge (termed *analysis knowledge* in this paper) of the handler implementation obtained from one program to attack another program.

Figure 1 depicts an attacking scenario where an attacker can resue the *analysis knowledge* to attack applications protected by VM-based code obfuscation. In this example, there are four different programs to be protected, which are labelled as A, B, C and D. In the first scenario, all the four programs are protected using an identical set of virtual instructions and bytecode handlers. Under such settings, an experienced attacker should be able to quickly reverse-engineer three programs by using the knowledge obtained from attacking one of the programs. In the second scenario, the protection system uses different protection schemes for different programs, so that they have different semantics and structural features across programs, so that the knowledge obtained from one program will be in unapplicable to others. This forces the attacker to start from the scratch when attacking each individual program.

To mitigate the effect of reuse of *analysis knowledge*, we propose DCVP (Code Virtualization Protection with Diversity), a enhanced VM-based code obfuscation system that obscures the semantics of bytecode instructions. We employ a technique called Instruction Set Randomization (ISR) [Kc et al. 2003] to randomly change the *opcodes* of bytecode instructions and their semantics. The randomization itself, however, is not sufficient for providing stronger protection, because it is similar to monoalphabetic substitution encryption and is easy to be bypassed according to the non-uniform frequencies of bytecode instructions. To overcome this issue, DCVP further partitions the bytecode program into several parts where the semantics of bytecode instructions in each part are randomized in different ways. As a result, the same bytecode instruction in different parts of the bytecode program is likely to have different semantics.

Of course, we have to admit that some malware authors also will use the above code virtualization technology to "protect" their malicious code and program. For example, Themida [Oreans-Technologies 2016b] is a popular tool of choice. In the same way, the reverse engineer isn't always the bad guy. However, in this paper, we put ourselves in the perspective of code protection, and aims to provide a mechanism for code protection to protect the legitimate interests of users from malicious infringement.

The key contribution of this paper is a countermeasure to address the issue of reusing *analysis knowledge* to perform code reverse engineering across programs. This is achieved by first partitioning the bytecode programs into several parts and randomizeing the semantics of bytecode instructions in each part in different ways. We compare our approach against VMProtect [VMProtect-Software 2016] and Themida [Oreans-Technologies 2016b] on `md5.exe`, `gzip.exe`, `bcrypt.exe` and `mat_mul.exe` applications. Experimental results show that DCVP is able to provide stronger protection at the cost of little extra overhead.

The rest of the paper is organized as follows. In Section II, we illustrate the internals of VM-based obfuscation. Section III gives our threat model, which describes the analyst's capabilities and goals, and we illustrate the process of reverse engineering a VM-obfuscated program. In Section IV, V, VI and VII, we present the details of DCVP. Section VIII evaluates DCVP for its effectiveness and its spatial and temporal overhead. Section IX presents the related work, and Section X presents the conclusion and discusses the future work.

2. BACKGROUND

Virtualization technique has been used in many fields, e.g. virtual memory for resource virtualization, VMware and VirtualBox for CPU virtualization, and Java bytecode and .Net CIL for application virtualization. This paper discusses another application of virtulization technique in protecting software programs from unauthorized analyses, namely code virtualized obfuscation or VM-based obfuscation, like VMProtect [VMProtect-Software 2016] and Code Virtualizer [Oreans-Technologies 2016a]. As software does not have uniform security requirements throughout its execution [Geneiatakis et al. 2012] and protecting the whole program is too costly, code virtualized obfuscation usually protects only the critical part(s) of the whole software program, which could be a critical algorithm or a processing logic. VM-based obfuscation protects a target program by transforming its native machine code into bytecode for a self-defined virtual instruction set architecture. At runtime, the execution instruction semantics of the original program are fulfilled by a native interpreter bundled

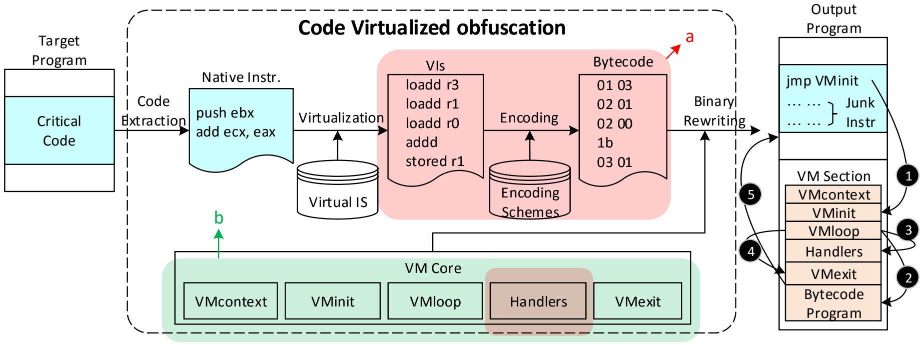


Fig. 2. The architecture of code virtualized obfuscation and the execution view of a VM-obfuscated program. The main work of this paper is to improve the core steps of VM-based protection (areas of "a" and "b"). In the "a" region, we adopt the partition bytecode encoding schemes, and obfuscate `handlers` to generate multiple sets of `handlers`. In the "b" region, we use a variety of methods of obfuscation and anti-taint analysis technology to protect the important components of virtual interpreter.

with the bytecode. In this section, we will look into the internal working mechanism of VM-based obfuscation.

Figure 2 shows the architecture of a VM-based obfuscation system. The core of a VM-based obfuscation are the virtual IS (Instruction Set) and the native interpreter. Virtual instructions are used to emulate native instructions. It is required that virtual IS be able to emulate all the semantics of native IS, or formally speaking, virtual IS should be Turing-equivalent to the native IS. The native interpreter is to fetch and execute bytecode instructions. It follows the *decode-dispatch* approach [Ghosh et al. 2012], and consists of a bundle of `handlers` and a `VMloop`. `VMloop` is the main *decode-dispatch* loop and for each loop, `VMloop` fetches a bytecode instruction, decodes it, and dispatches a `Handler` to interpret it. Different from native instructions, bytecode instructions are specific for a virtual context, namely `VMcontext`, which contains the virtual registers and flags. Virtual registers and flags are related to the native registers and flags. At runtime, `VMinit` first saves native context and uses them to initialize the virtual context. In the simplest implementation, the virtual context could be a block of memory and stores the exact values of the native context; this could be more complex [Falliere et al. 2009], but it should be guaranteed that the converting between native context and virtual context is reversible, since `VMexit` will restore the native context from the virtual context upon exiting the virtual interpreter.

Figure 2 also depicts the workflow of the obfuscation process. It starts from extracting the critical code from the target program. This is typically done with the help of the program author who will mark the location and scope of the critical code to be protected during programing; the obfuscation system then will search for the marks to locate the critical code at obfuscation time. The critical code is disassembled into native disassembly instructions to enable later conversion from native instructions to virtual instructions in an instruction-by-instruction fashion. The rules of conversions are set ahead of protection and are stable inside a VM-based obfuscation system. These rules depend on the semantics of the virtual IS and guarantee that the semantics of the resulted virtual instructions are equivalent to the native ones. Subsequently, virtual instructions are encoded into bytecode program. Finally, the bytecode program and other VM components are assembled into the target program through binary rewriting. This paper improves the core steps of code virtualization protection. We modify the encoding schemes and adopt the partition bytecode programming, and generate multiple sets of `handlers`. So the bytecodes will have different semantics in different parts of bytecode program. We also use a variety of methods of obfuscation and anti-taint analysis technology to protect the critical components of virtual interpreter (section 7).

At runtime, upon executing the "critical code", an instruction, `jmp VMinit`, transfers the control to `VMinit` (Step ❶). `VMinit` saves the native context and initializes the virtual context. Next, `VMloop` starts to work. It fetches a bytecode instruction, decodes it (Step ❷) and dispatches a `handler` to interpret it (Step ❸). Step ❷ and Step ❸ are iterated until all the bytecode instructions are interpreted. Then, `VMloop` transfers the execution to `VMexit` (Step ❹), where the native context is restored and the program jumps back to the native instruction following the critical code (Step ❺) and continue to execute the rest of the program code.

3. THREAT MODEL

In our threat model, an analyst owns a copy of the target VM-obfuscated software program and runs it in a malicious host environment [Collberg and Thomborson 2002]. In a malicious host environment, also referenced as the white-box attack context [Chow et al. 2003; Liem et al. 2008], an analyst has full privileged accesses to the system, and she can execute the software program at will and take advantages of any static and dynamic analysis tools (such as, "`IDA`"[1], "`OllyDbg`"[2] and "`Sysinternals Suite`"[3]) to help to trace and analyze instructions, monitor registers and process memory, and even change instruction bytes and control flows at runtime, etc. The analyst in our threat model is defined as an entity that seeks to reverse engineer and understand a software program's inner implementation and logic details, since code understanding is the basis for later tampering, cracking, or behavior pattern extraction. The ultimate goal of the analyst is to automate the entire analysis process. At present, there are two mainly types of methods to attack VM-based protection system. The first is based on the virtual execution analysis, the other is based on the behavior and semantic analysis.

3.1. Attack based on the virtual execution analysis

This method is proposed by Rolles et al. [Rolles 2009], which is based on the analysis of the interpretation process of the bytecode program, and requires an attacker to have a certain understanding of the principle of code virtualization.

It can be summarised as the following three steps. First, reverse engineering the virtual interpreter. The purpose of this step is to get the location and interaction information of each component of the virtual interpreter and the mapping relationship between the real CPU environment and `VMcontext`. Then using these informations to work out the semantics of individual bytecode instructions. Finally, recovering original program's logic embedded in the bytecode program, eliminating redundant information and restoring a program that is similar to the original program. Nicolas Falliere [Falliere et al. 2009] presented an example of the above analysis process which is used to analyze the Trojan.Clampi protected by VMProtect [VMProtect-Software 2016].

Since the virtual interpreter consists of `VMloop` and `Handlers`, an analyst needs to locate them and analyze how `VMloop` works and what each handler does. By tracing the *decode-dispatch* loop in `VMloop`, the analyst could figure out the correspondences between bytecode instructions and handlers, and thus the semantics of individual bytecode instructions. Combining the above information, the analyst can figure out what

[1] IDA Pro, https://www.hex-rays.com/index.shtml.
[2] OllyDbg, http://www.ollydbg.de/.
[3] Sysinternals Suite, https://technet.microsoft.com/en-us/sysinternals/bb842062/.

the bytecode program does, and after some simplification (constant folding and dead code elimination for example [FightingOreans'VM 2008]), she could reveal the original program's logic.

In the classical code virtualized obfuscation, the relationships between the opocodes of bytecode instructions and their semantics (the `handlers`) are stable, which means that a bytecode instruction in different obfuscated programs has identical semantics. Once an analyst gets aware of such relationships and semantics of each handler from previous analysis work or materials published by another analyst, she could reuse them for analyzing another VM-obfuscated program more efficiently, just like the first scenario in figure 1.

3.2. Attack based on the behavior and semantic analysis

This type of attack method can be used to attack not only code virtualization protection but also other confusion methods. Coogan et al. [Coogan et al. 2011] puts forward a behavior based analysis method, which aims to analyze the important behavior of code, but it does not pay attention to how to restore the original code. Its implementation steps are as follows: (1) Dynamically trace program execution process by using debugging tools, and collect some instruction execution information, such as the address of the instructions and register values. (2) Analyze and identify the system calls and its related parameters from the above information. (3) Further mark all of instructions that influence on the system calls. (4) Extract these labeled instructions and analyze their behavior.

This type of approach is usually used for malicious code analysis because it is based on analysis of the interaction between the program and system. The malicious code will interact with the system frequently in order to achieve a malicious purpose, but it is not necessary for the benign code. In other words, if the protected code does not interact with the system frequently, this approach will not be very effective on reverse analysis.

Another attack method based on semantic is proposed by the Yadegari et al. [Yadegari et al. 2015], which use taint propagation to track the flow of inputs values, and semantics-preserving code transformations to simplify the logic of the instructions. The implementation steps are as follows: (1) Dynamically trace program execution process by using debugging tools, identify the input and output of the program. (2) With the input of the program as a taint source to perform the taint propagation, and extract the affected instruction sequence. (3) Simplify the above instruction sequences by using code simplification techniques, then construct the control flow graph of the program and optimize it, and finally get the final result.

For the results of a run obtained, the function is equivalent to the original program, but it is only for one implementation, and does not cover all the execution branches. So the final control flow graph is only part of the original program. We need to perform analysis through multiple tracking and specify different input values each time, then comprehensive analysis to get a more complete control flow graph.

In conclusion. The first type of attack method based on virtual execution is closely related to the principle and structure of the code virtualization, and has the most realistic and comprehensive results. The second method has wider applicability, but it is hard to get a comprehensive analysis results. So this paper mainly aims at the first kind of attack, but also will provide some measures to prevent the second kind of attack. And in our threat model, we assume that the analyst is familiar with the mechanism of code virtualized obfuscation and follows the above steps while reverse engineering a VM-obfuscated program. The ultimate goal of the analyst is to fully reverse engineer the VM-obfuscated application and automate the reverse analysis process.

4. DESIGN DETAILS

As we have described, the reuse of *analysis knowledge* about the semantics of bytecode instructions can save an analyst much time and effort. Since the longer the software stay uncompromised, the more benefit the software authors will get, the naive idea to harden code virtualized obfuscation is to disable the *analysis knowledge* and force analysts to analyze the obfuscated software programs from scratch. Our solution is randomizing the relationships between the bytecode instructions and their semantics for each obfuscation instance. In this section, we will present the details about our solution. For better understanding, we will start from the design of virtual IS and `handlers`.

Virtual IS and Handlers. Virtual IS (Virtual Instruction Set) and `handlers` are the basis of the VM-based obfuscation system. First analysis of the native instructions, design complete virtual instruction set according to the classification of native instructions, and then design the `handlers` that used to interpret the VI (Virtual Instruction). The details are described as section 5.

Native Instructions to VIs. Extract the native instructions of key code segment from the target program, and then convert native instructions to the corresponding VIs according to designed virtual IS. The details are described as section 6.

VIs to Bytecode. It is a very important step that convert virtual instructions into a bytecode program in the process of code virtualization protection. System will generate a special bytecode program based on special encoding schemes, and these bytecodes can only be interpreted by the special interpreter of the system itself. This paper adopt the method of partitioning and randomization of virtual instruction semantics, and the different parts of the bytecode program will have different semantics. So different partitions need a set of specific interpretation procedures, the system will generate multiple sets of `Handlers` with same functions but different forms. Our approach will also adopt some anti-taint analysis techniques to protect `Handlers` and other components from taint analysis. The details are described as section 7.

DCVP will convert all of the critical instructions into bytecode program that can only be interpreted by the system itself. Finally the bytecode program and other VM components (such as, VMloop, VMcontext et al.) are assembled into the target program through binary rewriting.

5. VIRTUAL INSTRUCTION SET AND HANDLERS

As the basis of a VM-based obfuscation system, virtual IS and the `handlers` should be the first considerations when designing such an system. The challenge lies in devising a feature-complete virtual IS that is Turing-equivalent to native IS, which means that any native instructions could be substituted with the virtual instructions. Virtual instructions ultimately will be executed by the hand-crafted `handlers`; these `handlers` are written in native instructions.

There are two main VM architectures: stack-based, and register-based. Examples of stack-based virtual machines are the Java Virtual Machine and the .Net CLR, and examples of register-based virtual machines are the Lua VM, and the Dalvik VM. In this paper, we choose stack-based architecture for the VM-based obfuscation system for the following reasons:

- In a stack-based VM, operations are carried out with the help of stack, where operands and results of operations are stored. This simplifies the addressing of operands and ultimately simplifies the implementation of `handlers`.
- The process of converting native x86 instructions to virtual instructions is simpler.
- Stack-based VMs require more virtual instructions for a given computation; this makes the instructions more complex and conforms to our objective of impeding reverse analysis.

To devise a virtual IS that is Turing-equivalent to the native IS, one naive approach is devising a virtual instruction for every single native instruction. However, this results in a very large size of handlers. As the basic idea of stack-based VMs implies, a native operation are carried out or virtualized by virtual instructions in a three-phase fashion: pushing operands into the stack, executing the aimed operation, and storing the result into the virtual context. Therefore, it is sufficient for the virtual IS to include the following instructions:

- `load` instructions and `store` instructions for data transfers. `load` instructions are for pushing operands into stack, and `store` instructions are for popping results out of the stack and store the results into the virtual context.
- Arithmetical and logical instructions. The variants of these kind of instructions are much smaller than their native ones, as the addressing mode of operands is simpler and uniform, i.e., stack-based addressing.
- Branch instructions for changing the control flow of bytecode program.

Other instructions that are not included in the above categories are defined as special virtual instruction - `undef`, which we will discuss later. We first discuss the different formats of these instructions and how to implement the handlers of them.

5.1. ”`load`” and ”`store`” Instructions

`load` and `store` instructions are used for preparing operands and storing the results of operations. They are used in the first and third phases of virtualizing a native instruction. In our virtual IS, they are the only ones that have operands. For a `load` instruction, the operand could be a virtual register, a memory addresses, or an immediate value, and for a `store` instruction, the operand is a virtual register or a memory address. Virtual registers are stored in the virtual context, i.e., the `VMcontext`. A naive construction of `VMcontext` is simply copying the values of native registers into the `VMcontext`. But the mapping between the virtual registers and the native registers is not necessarily one-to-one. To further impede the reverse analysis, the mapping mechanism could be made purposely more complex, as NISLVMP [Wang et al. 2013] does. In this section, we only consider the one-to-one mapping between the virtual registers and the native ones.

Besides the operand type, the operand size matters as well. In x86 architecture, the size of an operand could be 8-bit, 16-bit, and 32-bit. For example, given a memory address, the `load` instruction could fetch the first 8-bit, or the lower 16-bit value, or the entire 32-bit value that stored in that address. Therefore, it is better to design a virtual instruction for every distinct combination of operand type and size. However, in x86 architecture, `push` and `pop` operations do not support 8-bit operations. We decide to delay distinguishing different size operands to the second phase of virtualizing native instructions. Table I shows the virtual instructions of `load` and `store` and their corresponding `handlers`. In table I, there exists four special virtual instructions: `load_r8h` and `store_r8h` are used when we encounter an operation that manipulate the second least significant byte of a register, i.e., `ah`, `dh`, `ch`, `bh`. `load_ms` and `store_ms` have no operands and are used to process instructions with indirect memory addressing mode

Table I. The Virtual Instructions and Corresponding Handlers of load and store.

VI	Handler
load_r reg	movzx eax, byte [VPC] *;get vr index* add VPC, 1 push dword [VMcontext+eax*4]
load_r8h reg	movzx eax, byte [VPC] *;get vr index* add VPC, 1 movzx eax, byte [VMcontext+eax*4+1] push eax
load_m mem	mov eax, dword [VPC] *;get memory addr* add VPC, 4 push dword [eax]
load_i8 imm8	movzx eax, byte [VPC] *;get 8-bit imm value* add VPC, 1 push eax
load_i16 imm16	movzx eax, word [VPC] *;get 16-bit imm value* add VPC, 2 push eax
load_i32 imm32	mov eax, dword [VPC] *;get 32-bit imm value* add VPC, 4 push eax
load_ms	pop eax push dword [eax]
store_r reg	movzx eax, byte [VPC] *;get vr index* add VPC, 1 pop dword [VMcontext+eax*4]
store_r8h reg	movzx eax, byte [VPC] *;get vr index* add VPC, 1 pop edx mov byte [VMcontext+eax*4+1], dl
store_m mem	mov eax, dword [VPC] *;get memory addr* add VPC, 4 pop dword [eax]
store_ms	pop eax pop ebx mov dword [eax], ebx

Note: In the table, reg means register, mem memory address, imm immediate value, and vr virtual register. VPC is short for Virtual Program Counter and represents the address of the next bytecode instruction to interpret.

(memory address is stored in a register or presented as an expression). An example in table V illustrates the situation of using of load_ms.

5.2. Arithmetical and Logical Instructions

Arithmetical and logical virtual instructions are used to execute the aimed operations and they need not to worry about operands, since their operands have been pushed into the stack by load instructions. However, as we said before, these instructions must consider the size of the operands. These instructions are in similar forms and we take add operation as an example to illustrate. Table II lists the virtual instructions and handling procedures of add operation. Since the operations of these instructions could be 8-bit, 16-bit, or 32-bit, we design a virtual instruction for each of the operations of different operand size.

Table II. The Virtual Instructions and Handlers of add Virtual Instructions.

VI	Handler
add8	pop eax add *byte* [esp], al
add16	pop eax add *word* [esp], ax
add32	pop eax add *dword* [esp], eax

Note: Since the operations of these instructions could be 8-bit, 16-bit, or 32-bit, we design a virtual instruction for each of the operations of different operand size.

5.3. Branch Instructions

In native IS, the commonly used branch instructions include `jmp`, `jcc` (conditional jump), `call`, and `retn`. It is a big challenge to virtualize these instructions, since they have many different variants and each of these variants needs a virtual instruction. Considering the destination of a jump (branch) instruction, if its destination also resides in the critical code, we call it an *inner jump*; otherwise, we call it an *outer jump*. Figure 3 illustrates these two kinds of branch instructions. In DCVP, we ignore the situation where the destination of a branch instruction outside the critical code resides in the critical code, the start instruction of the critical code excluded. This situation will not occur if the critical code to be protected is well-structured.

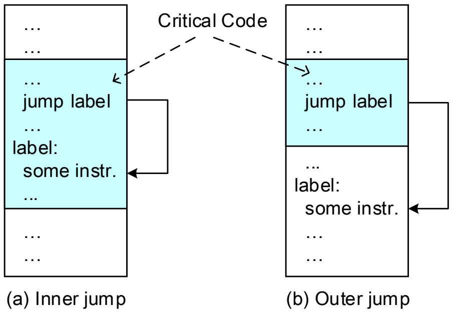


Fig. 3. The destination of a branch instruction could be in the critical code or outside the critical code. Branch instructions include `jmp`, `jcc`, `call`, and `retn`.

Besides the location of the destination instruction, we should also consider if the destination of a branch instruction can be determined statically. From this point of view, the branch instructions can be divided into two categories: one is *direct branches*, whose destinations are calculated in a PC (Program Counter)-relative mode and can be calculated statically. The other is *indirect branches*, whose destinations are stored in registers or memories. Their destinations are undefined statically and are determined at runtime. Table III classifies different forms of `jmp/jcc/call/retn` instructions considering the above two categories.

Direct Branches. Since `jmp` instructions are the basis of the other branch instructions, we elaborate on the introduction of the virtual instructions and handlers of `jmp` instructions. As we have illustrated, the destination of a direct `jmp` instruction can be

Table III. Examples of Direct and Indirect Branches.

Direct Branches	`jmp rel8/16/32`
	`jcc rel8/16/32`
	`call rel16/32`

Indirect Branches	`jmp reg32/mem32`
	`call reg32/mem32`
	`retn`

Note: In this table, `rel` means a PC-relative address. The numbers (8, 16, and 32) are the size (in bit) of the operands.

calculated statically. If the destination instruction of the `jmp` instruction resides in the critical code, the `jmp` is a *direct inner* `jmp`; otherwise a *direct outer* `jmp`. We designate virtual instructions for both of them, `jmp_di` for the former, and `jmp_do` for the latter. For *direct inner* `jmp`, it is able to obtain the corresponding bytecode instruction address of its native destination instruction during protection. Therefore, we set that bytecode instruction address as the operand of `jmp_di`, which is pushed into stack by `load_i`. The handler of `jmp_di` fetches the address from stack and assigns it to VPC (Virtual Program Counter). For *direct outer* `jmp`, we just need to jump to that destination instruction. Prior to that, we should restore the native context. Therefore, the operand of `jmp_do` is the address of the native destination instruction. Table IV presents the above two virtual instructions of *direct* `jmp` and their handlers.

Table IV. The Virtual Instructions and Handlers of `jmp` Instructions.

VI	Handler
`jmp_di`	*;operand: addr of the dest. bytecode instr.* `pop eax` *;get operand* `mov VPC, eax`
`jmp_do`	*;operand: addr of the dest. native instr.* `pop [mem]` *;get operand* . . . *;restore native context* `jmp dword [mem]`

Note: `jmp_di` is for *direct inner* `jmp` and `jmp_do` for *direct outer* `jmp`.

`jcc` and *direct* `call` instructions are similar to *direct* `jmp`. `jcc` has many different kinds of instructions for different conditions, and each needs a specific virtual instruction. In the `handlers`, some extra instructions are needed to check the state of the conditions and decide to jump or not. `call` instructions can be considered as a `push` instruction followed by a `jmp` instruction. The `push` instruction pushes the return address into the stack and the `jmp` instruction jumps to the address of the subroutine.

Indirect Branches. Since it is difficult to obtain the address of an indirect branch, we cannot decide whether the branch is an inner one or an outer one. One solution is to delay the decision until runtime. In such case, however, the implementation of the handler is complex. Hence, for these instructions, instead of using a similar idea as that for *direct branches*, we use a special virtual instruction `undef`, which will be introduced later.

5.4. Other Instructions

The above three categories cover the commonly used instructions. Although the other native instructions are rarely used, such as `bts`, `enter`, `int n`, and `out`, our native IS should consider them too. For these instructions, we define a special virtual instruction - `undef`. At runtime, when encountering such an instruction, it first restores the native context and exits the VM. Then, it executes that native instruction in native context and finally re-enters the VM and continues to execute the left bytecode instructions. The *indirect branches* are also processed in this way.

Table V. Examples of Native Instructions and Their Corresponding Virtual Instructions.

Native Instr.	VI
mov eax, ebx	load_r 3 store_r 0
mov eax, dword [esi+4]	load_r 6 load_i32 4 add32 load_ms store_r 0
add eax, edx	load_r 0 load_r 2 add32 store_r 0
jmp 4020a8h (direct inner jump)	load_i32 42a583h jmp_di

Note: 42a583h is the bytecode instruction address that corresponds to the native instruction at 4020a8h.

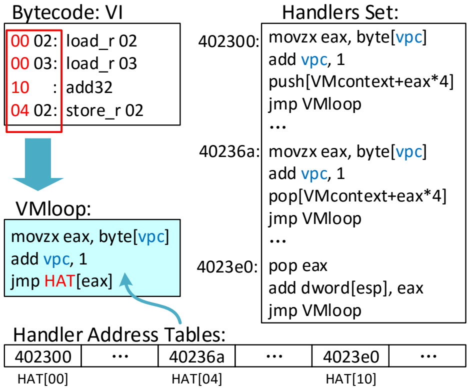
6. NATIVE INSTRUCTIONS TO VIRTUAL INSTRUCTIONS

At obfuscation time, we first convert native instructions into virtual instructions. This process follows a three-phase fashion: first, loading the operands into stack with `load` virtual instructions; then, executing the aimed operation; and finally, storing the result into virtual context or a certain memory address with `store` virtual instructions. Table V gives some examples of native instructions and their corresponding virtual instructions.

Data transfer instructions are mainly mapped into `load` and `store` instructions. Typical examples of these instructions are `mov`, `push`, and `pop`. Arithmetical and logical instructions strictly follow the three-phase processing. Branch instructions are mapped into a `load` instruction followed by a branch virtual instruction. Native instructions with complex addressing modes are processed iteratively with the above virtual instructions, for example, the `"mov eax, dword[esi+4]"` instruction in Table V.

7. VIS TO BYTECODES

Virtual instructions will be encoded into bytecodes in the end. It is similar to that an assembler assembles assembly instructions into machine code and only can be interpreted by virtual interpreter of VM-based protection system. We adopt an encoding scheme less compacted than the x86 instruction architecture which uses separate bytes for the *opcode* and *operand* of a virtual instruction. In practice, we assign each virtual instruction a distinct ID as its *opcode*. The ID is used by `VMloop` as an index to find the address of the `handler` of the virtual instruction in the address table recording the addresses of each `handler`. Since the number of virtual instructions is less than 256, thus one byte is sufficient to encode their IDs. As for the *operands*, since they



```
00 02: load_r 02
00 03: load_r 03
10   : add32
04 02: store_r 02
```

```
movzx eax, byte[vpc]
add vpc, 1
jmp HAT[eax]
```

```
402300: movzx eax, byte[vpc]
        add vpc, 1
        push[VMcontext+eax*4]
        jmp VMloop
        ...
40236a: movzx eax, byte[vpc]
        add vpc, 1
        pop[VMcontext+eax*4]
        jmp VMloop
        ...
4023e0: pop eax
        add dword[esp], eax
        jmp VMloop
```

402300	...	40236a	...	4023e0	...

Fig. 4. Examples of some virtual instructions and their bytecode. Each virtual instruction is encoded into a bytecode instruction, which consists of an *opcode* and optionally an *operand*. The bytecode instructions feed into `VMloop` and the *opcode* of each bytecode instruction is used by `VMloop` as an index to find the address of the corresponding handler in the HAT (Handler Address Table).

could be of different size[4], we use one, two, or four bytes to encode them correspondingly. Figure 4 shows some examples of virtual instructions and their bytecode. The figure also demonstrates how `VMloop` fetches and interprets the bytecode instructions.

7.1. Randomize the Semantics of Bytecode Instructions

From the above demonstration, if an analyst gets known the semantics of a bytecode instruction, the next time she encounters it, she does not bother to analyze its `handler` once again to figure out what it does[5]. For example, in Figure 4, the bytecode instruction `"10"` means an addition operation through analyzing `Handler_4023e0`. The next time we encounter a bytecode instruction of `"10"`, we could say that it does an addition operation immediately.

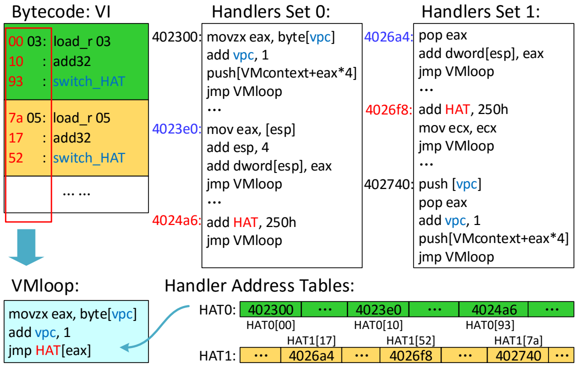
To mitigate the effect of reuse of previously obtained *analysis knowledge*, we randomize the semantics of virtual instructions. According to the encoding scheme we adopt, it is easy to achieve this goal. The idea is to change the relationship between the IDs (*opcodes*) and the virtual instructions, which is similar to [Kc et al. 2003]. Every time to encode the virtual instructions, the IDs are first shuffled once. Then the shuffled IDs are used to encode the virtual instructions. The addresses of `handlers` are also filled into the handler address table accordingly.

7.2. Partition Bytecode Program

With the randomization of the semantics of bytecode instructions, an analyst can not directly reuse her *analysis knowlege* to work out what a bytecode instruction actually does. However, the effect of the randomization could be easily bypassed. As shown in table V, the frequencies of virtual instructions are not uniform, where `load_r` and `store_r` are two of the most frequently used virtual instructions. Thus, an analyst could infer the semantics of bytecode instructions based on the non-uniform frequencies of *opcodes*.

[4] The *operand* of a virtual instruction could be an index for virtual register, an immediate value, or a memory address. They could be of different size: a virtual register index being 8 bits, an immediate value being 8/16/32 bits, and a memory address being 32 bits.

[5] Since `handlers` could be mutated to hinder analysis, it saves an analyst a lot of time end effort without bothering to analyze them once again.



```
00 03: load_r 03
10   : add32
93   : switch_HAT
7a 05: load_r 05
17   : add32
52   : switch_HAT
       … …
```

```
402300: movzx eax, byte[vpc]
        add vpc, 1
        push[VMcontext+eax*4]
        jmp VMloop
        …
4023e0: mov eax, [esp]
        add esp, 4
        add dword[esp], eax
        jmp VMloop
        …
4024a6: add HAT, 250h
        jmp VMloop
```

```
4026a4: pop eax
        add dword[esp], eax
        jmp VMloop
        …
4026f8: add HAT, 250h
        mov ecx, ecx
        jmp VMloop
        …
402740: push [vpc]
        pop eax
        add vpc, 1
        push[VMcontext+eax*4]
        jmp VMloop
```

```
movzx eax, byte[vpc]
add vpc, 1
jmp HAT[eax]
```

```
HAT0: 402300 | … | 4023e0 | … | 4024a6 | …
HAT1: … | 4026a4 | … | 4026f8 | … | 402740 | …
```

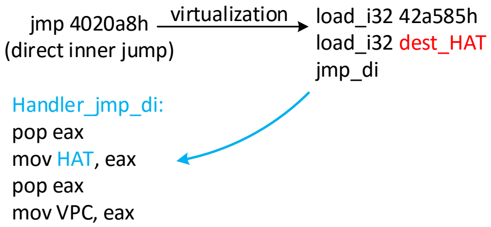
Fig. 5. Example of partitioning virtual instructions into several parts (two parts in this figure). Virtual instructions in different partitions are encoded differently and interpreted using different `handlers` set. The number of HAT increases accordingly. To switch the currently used (by `VMloop`) HAT to the next one, we add a new virtual instruction `switch_HAT`. The *operand* of `switch_HAT` is the size of a HAT.

7.2.1. Bytecode Program Partition Design. To frustrate the inferences based on the frequencies of *opcodes*, we partition all the generated virtual instructions into several parts, each part been encoded differently. Specifically, during obfuscation, instead of encoding the generated virtual instructions all at a time, we encode those resulted parts separately. And prior to each encoding process, we first randomly shuffle the IDs of virtual instructions and then use the results for encoding. The effect of the shuffles is that an identical *opcode* in different parts of the bytecode program probably reveals different semantics, thus the frequencies of *opcodes* are obscured . Figure 5 shows an example of partitioning the virtual instructions into two parts. The opcode of a virtual instruction is probably encoded differently in different parts. For example, `load_r` is encoded into `"00"` in the first part, while `"7a"` in the other.

ALGORITHM 1: Partition Bytecode Program

```
Input: VIs of critical code segment, partition number $N$
Output: Bytecode program with $N$ partition
Apply memory space M
VIs randomly divided into $N$ partitions
Generate $N$ sets of handlers by using algorithm 2
while $N \neq 0$ do
    Take a partition
    Randomly select a set of handlers and shuffle the IDs of virtual instructions
    Build a corresponding HAT.
    Use the shuffle results to encode bytecode program P.
    Store the P to M
    $N$ decrement operation
end while
```

Algorithm 1 demonstrated the implementation of the partition encoding. Partition the bytecode program, we will randomly divide virtual instructions into N partitions and obfuscate the original HAS (Handler Set) for N times to get N sets of `handlers`, and the detailed HAS obfusction approach are demonstrated in section 7.3. For each partition, we will randomly select one set of `handlers` and shuffle the IDs of virtual IS, and then use the results to encode the VIs into a specific bytecode program. In the



```
jmp 4020a8h
(direct inner jump)
```

```
load_i32 42a585h
load_i32 dest_HAT
jmp_di
```

```
Handler_jmp_di:
pop eax
mov HAT, eax
pop eax
mov VPC, eax
```

Fig. 6. The virtualization of a *direct inner* transfer instruction with HAT switching. The address of the destination HAT is pushed into stack by `load_i`, and is assigned to the HAT pointer used by `VMloop` at runtime.

end, a complete bytecode program is generated, which has multiple partitions and the same bytecode has different semantics in different partitions. These specific bytecodes only can be can be interpreted by system's virtual interpreter.

As the *opcodes* of bytecode instructions are used by `VMloop` as the indexes for the addresses of their corresponding `handlers`, and different partitions are encoded differently, each partition needs their own HAT. At the end of a partition, the HAT used by `VMloop` should be switched to the HAT of the next partition. This is done by a new virtual instruction - `switch_HAT`. Since `switch_HAT` is always added to the end of a partition and the orders of HATs are in accordance with that of the partitions, `switch_HAT` needs to add the size of a HAT to the HAT pointer used by `VMloop` (as `Handler_4024a6` does in figure 5). In our prototype, the number of `Handlers` is `148` and the address of a `Handler` is `4` bytes, thus the size of a HAT is `592` (`250h in hexadecimal`) bytes.

The switchings of HATs is not limited to the end of partitions. A branch instruction also causes the switching when its destination resides in a different partition. Branch instructions change the control flow of a program through changing the VPC. When encounter such an instruction, we cannot simply append a `switch_HAT` to it, since the `switch_HAT` may not get interpreted by `VMloop` if the VPC is changed to a location in another partition. Hence, we put the code for switching inside the `Handlers` of the branch instructions. Here, the branch instructions indicate the *direct inner* ones, as *direct outer* branches and *indirect* branches all leave the virtual context and need not to worry about the switching of HATs. During protection, for each *direct inner* branch instruction, we first calculate its destination, and then figure out which partition the destination resides. The address of the HAT of that partition is pushed into stack by `load_i` and will be used by the `Handler` of the branch instruction to set the value of the HAT pointer used by `VMloop`. Figure 6 shows the virtualization of a *direct inner* branch instruction, compared to that in table IV.

7.2.2. Security Analysis of Partition Design. Assume there is an attacker now, which uses the attack method based on virtual execution that introduced in section 3 to reverse the partition bytecode program. First, the attacker needs to perform dynamic debugging the target program, and then spend some time to locate the address of `VMloop` from the obfuscated virtual interpreter. Next he needs to collect the bytecode program by analyzing the parameters of `VMloop`.

To restore the logical function of the original code, the attacker needs to analyze the semantics of these extracted bytecodes. But for the target program, which generated by using a special encoding schemes and its bytecode program is divided into multiple partitions. So the attacker must first obtain the partition informations of critical code to further reverse analysis, but it is difficult for him. After the NIs is converted to a VIs, there is a HAT switching operation by using VI - `switch_HAT` at the end of each partition. However, virtual instruction is just an intermediate language in the process of protection, and it does not appear in the final program. All of the instructions will eventually be in the form of bytecodes, and the bytecode semantics of each partition is randomly changed (such as bytecode “93” and “52” in figure 5). So the attacker cannot use this feature to delineate the partition. He can only spend a lot of time through continuous tracking and debugging to predict the partition of the transformation.

Assume that the attacker gets the partition information after a lot of analysis. Next the attacker need to analyze the bytecode of each partition and extract the semantic information implied by the `handlers`. Because the bytecode of each partition has a different semantics, and their mapping `handlers` are also with different forms. Therefore, the attacker cannot reuse the previous *analysis knowledge* to attack the next partition, and for different target program that protected by DCVP is more like this. Unique partition configuration and different `handler` sets, so that the attacker cannot achieve the batch automatic attack by building attack knowledge base. The attacker has to spend a lot of time to analyze every detail for each program.

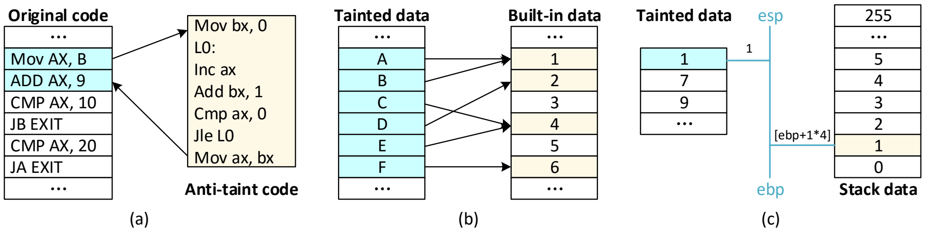
7.3. Obfuscate the Handler Sets

To further impede automated reverse analysis, we can obfuscate the `handlers`, and use different obfuscation strategies for `handlers` in different partitions. As a result, a `handler` in different partitions will looks different. An analyst cannot immediately recognize them as the the same one and needs to analyze each of them, which increases the workload of the analyst. At the same time also can prevent the attacker use attack knowledge base to match these `handlers` to achieve automated reverse analysis.

ALGORITHM 2: Virtual Interpreter Obfuscation

```
Input: The original HAS, partition number $N$
Output: $N$ equivalent HASs with different forms
while $N \neq 0$ do
    Apply memory space M1
    Take the first handler from HAS
    while There are still untreated handler do
        Randomly select multiple obfuscation methods and using order of them
        Obfuscate handler and store the results to M1
        Take the next handler from HAS
    end while
    Apply memory space M2
    Using the anti-taint analysis technique to protect the code in M1
    Store the results to M2, and release the memory space of M1
    $N$ decrement operation
end while
```

The obfuscation approach we take is shown in Algorithm 2. The number of HAS obfuscated is determined by the number of partitions. Our system will randomly select several methods from the obfuscation method library which contains the junk instructions injection, equivalent instruction substitution [Nicolaou George 2009], code out-of-order [Birrer et al. 2007] and control flow flattening [Wang et al. 2001]. Then the system will use the selected method in a random order to obfuscate the `handler`. Finally we have multiple equivalent but different forms of HASs. We will also adopt some anti-taint analysis techniques (some details are presented in section 7.4) to protect the HASs that after obfuscated. This can effectively prevent the virtual interpreter from being attacked by some de-obfuscation methods.



```
...
Mov AX, B
ADD AX, 9
CMP AX, 10
JB EXIT
CMP AX, 20
JA EXIT
...
```

```
Mov bx, 0
L0:
Inc ax
Add bx, 1
Cmp ax, 0
Jle L0
Mov ax, bx
```

Fig. 7. Examples of three anti-taint analysis techniques. Respectively as, (a) the transfer of naive model, (b) the transfer of control dependencies and (c) the indirect transfer of stack pointer.

For example, HAS (Handler Set) as an original handler set that consists of m handler. We use a HAT to store the address of these handlers, and their index corresponds to the *opcode* of virtual instruction. HAS will be obfuscated for n times with different strategies, n is dependent on the number of partitions. Then we get multiple HASs and which are semantic equivalence but have different forms. At this time, all of the equivalent handlers still have the same index. This is a type of insecure and direct mapping relationship. Therefore, according to the method that partition bytecode program and randomized the semantics of bytecode instructions in the upper section 7.2, we first randomly shuffle the IDs of virtual instructions and then use these results to generate a new HAT for each partition (as shown in figure 5). The effect of shuffles is that an identical *opcode* in different parts of the bytecode program probably reveals different semantics. The relationship of these equivalent handlers in different HASs should be:

$$HAS_1(i) \Leftrightarrow HAS_2(j) \Leftrightarrow \ldots \Leftrightarrow HAS_n(k),\ 1 \leq i,\ j,\ k \leq m.$$

This various semantics of bytecode instructions and different forms of handlers can effectively prevent the attacker from using the attack knowledge base matching to realize the automated reverse analysis. The attacker has to spend a lot of time to analyze every detail.

7.4. Anti-Taint Analysis

Simply obfuscating the handlers, however, cannot prevent an attacker from reverse engineering completely. There are several existing de-obfuscating techniques that can be used to counter the traditional virtualization protection. Such as, Coogan et al. [Coogan et al. 2011], who use equational reasoning about assembly-level instruction semantics to simplify away obfuscation code from execution traces of emulation-obfuscated programs. Yadegari et al. [Yadegari et al. 2015], which use taint propagation to track the flow of values from the programs inputs to its outputs, and semantics-preserving code transformations to simplify the logic of the instructions that operate on and transform values through this flow.

However, the work of Coogan et al. is based on equational reasoning about assembly level instruction semantics, and such a approach has some drawbacks that is difficult to simplify the complex equation, making it hard to separate out the different components of nested loops or complex control flow. So it is difficult to extract the implicit logic from the high level structures, such as control flow graphs or syntax trees. Besides, this is a kind of behavior based analysis method, which has good applicability to the analysis of malicious code, but the result of the analysis of the benign program is not good. The results can not fully reflect the function of the protected code.

Work of Yadegari et al. can be effective even when applied to previously unseen obfuscations, but there are still some shortcomings that code simplification should be first to identify input-to-output flows by using taint analysis. They rely on the taint propagation and forward/backward taint analysis to identify the explicit flow of values from inputs to outputs, then use control dependence analysis to identify implicit flows. Further on this basis to do trace simplification. Therefore, we can strengthen the `handlers` obfuscation approach with the anti-taint analysis to impede this kind of de-obfuscate approach.

In order to protect the obfuscated virtual interpretation procedure, we will adopt the anti-taint analysis techniques to protect the data flow of the `handlers` from taint analysis. Specific ways are as follows:

The transfer of naive model. A simple case is given in Figure 7-(a), assuming that there is a variable B that has been marked by taint, we need to transfer the value of B to `AX`. If we use the `MOV` instruction, `AX` will also be tainted. The transfer of naive model is that to do subtraction operation for B, at the same time to do addition operation for `AX`. When the value of B is reduced to ”0”, the value of AX increased from ”0” to B, and `AX` will not be marked as a tainted data.

The transfer of control dependencies. Control dependence analysis is an important step towards the process of taint analysis. For a set of tainted data, to carry out anti-taint analysis processing. As shown in Figure 7-(b), we can launder tainted data by assigning the data that non tainted directly to the tainted data. This process needs to match the tainted data and not tainted data, in order to ensure the correctness of the data will not be affected.

The indirect transfer of stack pointer. As we can see from Figure 7-(c). The indirect transfer of stack pointer took advantage of the working principle of the stack to implement the anti-taint analysis. The principle of this method is to first put a set of non tainted data into the stack, and then through the address pointer of stack data to access the stack data, and finally use the stack data to replace the tainted data to implement the anti-taint analysis.

The above three methods can effectively prevent the spread of tainted data by laundering tainted data, and then can effectively resist the taint analysis.

8. EVALUATION

In this section, we will evaluate the effectiveness of our method for mitigating the effect of reuse of *analysis knowlege* and the spatial and temporal overhead of our method.

8.1. Effectiveness Evaluation

DCVP is effective for impeding reverse analysis by invalidating existing *analysis knowledge*. From Section 2, we learned that understanding the semantics of bytecode instructions is essential for reverse engineering a VM-obfuscated program. Semantics are encapsulated in `handlers`, and the work of extracting them form `handlers` is tedious and error-prone. Therefore, it will save analysts lots of time and energy if the semantics of bytecode instructions are accessible, without bothering to trace and analyze the `handlers` once again. DCVP's aim is to frustrate this attempt and force analysts to analyze the `handlers` every time. Through randomizing the semantics of bytecode instructions, the same bytecode instruction probably means different obfuscated instances, and even different in the same obfuscated program by adopting different encoding schemes for different partitions of the bytecode program, which can largely confuse analysts and increase their workload.

Assuming that the number of virtual instructions, or the number of `handlers` in other words, is H, then the probability of two identical bytecode instructions in two obfuscated programs having the same semantics is $\frac{1}{H}$. The total number of distinct shuffle of the *opcodes* is $H!$. In an obfuscated program, supposing the bytecode program

Table VI. Statistics of the Target Programs.

Target Program	Version	Size(KB)	Critical Code	N1	N2	Ratio	N3
md5.exe	2.3	11	Transform()	1327	563	42.36%	85013
gzip.exe	1.2.4	56	deflate()	10181	153	1.50%	539082
bcrypt.exe	1.1.2	68	Blowfish_Encrypt()	2997	54	1.80%	1735710
mat_mul.exe	-	184	ijkalgorithm()	49327	60	0.12%	84325

Note: The 5th and 6th column gives the number of instructions in the entire program and these critical functions respectively, The 7th column gives the proportion of critical code. We use Pin[10] to count the number of the dynamically executed instructions in the critical functions and the results are shown in the last column. We chose only 60 instructions of `mat_mul` is special to verify the impact of DCVP on the program overhead when protecting a small amount of code. The results are shown in figure 9 and the impact is not obvious.

is partitioned into N parts, then the probability that a bytecode instruction in different partitions having the same semantics is $\left(\frac{1}{H}\right)^{N-1}$. Therefore, we believe DCVP can effectively remove the *analysis knowlege* about the semantics of bytecode instructions.

8.2. Overhead Evaluation

To evaluate the spatial and temporal overhead of our method, we implemented a prototype, namely DCVP, for obfuscating x86 PE executables on the Windows platform. In the implementation, we devised `148` virtual instructions and their corresponding `handlers`. We conducted all the experiments on a Dell Optiplex 960h with an Intel®Core$^{\text{TM}}$ 2 Duo Processor E8400 at 3.00GHz with 4.00GB of RAM. The operating system environment is Windows 7 Enterprise.

We use DCVP to protect four x86 PE executables, namely `md5.exe`[6], `gzip.exe`[7], `bcrypt.exe`[8], `mat_mul.exe`[9]. The first three are used to process a text file (`test.txt`) of 10KB and `mat_mul` is used to calculate the product of two 5×5 matrices. Table VI shows the statistics of these executables. For each program, we choose a piece of critical code to protect, as shown in table VI. The programs are protected 10 times, each time with a parameter that specifies a different number (1∼10) of partitions.

Figure 8 shows the size of the obfuscated programs. The horizontal axis specifies the number of partitions in the obfuscated programs and "`0`" means the original program. As the partitions increase, the increased bytes mostly come from the added HATs and HAS. Since the size of a HAT is only 592 bytes, the sizes of the HATs increase slowly. Besides, the sections in PE executables are aligned to a value (4096 or 512 usually)[Pietrek 1994], the increase of sizes may not reveal immediately as the partitions increase. The filesize of the program is mainly affected by the number of HAS, and it increases with the increase of HAS regularity.

To evaluate the runtime overhead that DCVP introduces, we run the obfuscated programs for several times and calculate the average execution time of them: `md5`, `gzip`, and `bcrypt` are used to process a text file (`test.txt`) of 10KB; `matrix_mul` is used to calculate the product of two 5×5 matrices. The average execution time is shown in figure 9. Among them, `bcrypt` has the largest increase of execution time from original program[11] to the obfuscated program with one partition. This resulted from that the critical instructions in `bcrypt` is executed much more times than others (as shown in

[6] MD5: Command Line Message Digest Utility. http://www.fourmilab.ch/md5/.

[7] gzip. http://www.gzip.org/#sources.

[8] Bcrypt - Blowfish file encryption. http://sourceforge.net/projects/bcrypt/.

[9] Matrix multiplication. https://github.com/MartinThoma/matrix-multiplication.

[10] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[11] The execution time of the original program is specified by "`0`" on the horizontal axis.

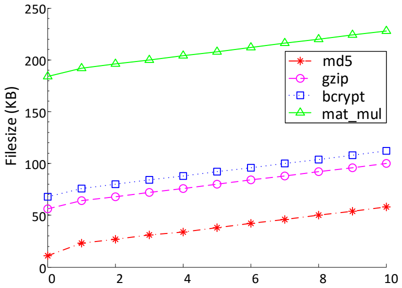


Fig. 8. The impact on file size (KB) of DCVP. The file size slightly increased with the increase of number of partitions

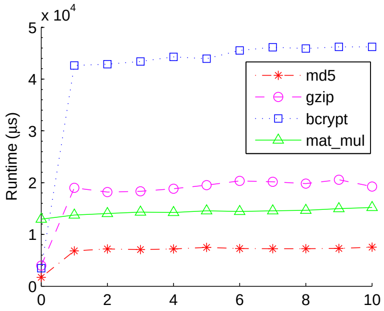


Fig. 9. The impact on runtime performance (μs) of DCVP with different partitions.

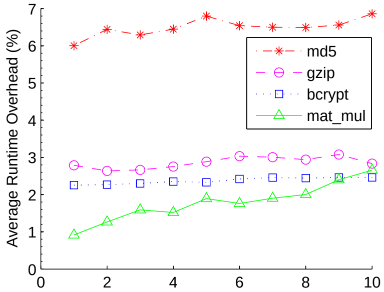


Fig. 10. The average runtime overhead per dynamically executed critical instruction.

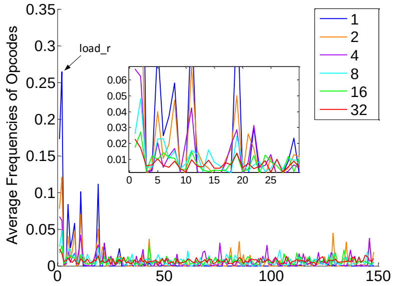


Fig. 11. The average frequencies of *opcodes*. The horizontal axis specifies the *opcodes*.

the last column in table VI). Besides, the execution time changes little as the number of partitions increases. From section 7.2 we can learn that if the number of partitions increases by one, the program only needs to execute an extra `handler` to interpret the `switch_HAT` instruction. The introduced runtime overhead is negligible. In some situations, the execution time of an obfuscated program with more partitions may have a slightly lower runtime overhead. This probably results from the locality of reference[12].

We evaluate the average runtime overhead per dynamically executed instruction. We use T_{ob} to denote the execution time of a obfuscated program, T_o for that of the original program, and C_e for the count of the critical instructions been dynamically executed. The average runtime overhead per dynamically executed instruction is calculated by

$$(T_{ob} - T_o)/C_e.$$

The results are shown in figure 10. From figure 10, we can learn that `md5` has the largest average runtime overhead per dynamically executed instruction. The reason is that the critical code of `md5` is full of arithmetical and logical instructions, which takes a longer time to interpret.

We also put the four target programs together and count the average frequencies of *opcodes*. We take the obfuscated programs with 1, 2, 4, 8, 16, and 32 partitions for comparison. The results are presented in figure 11. As we can see, as the number

[12] Locality of reference. https://en.wikipedia.org/wiki/Locality_of_reference

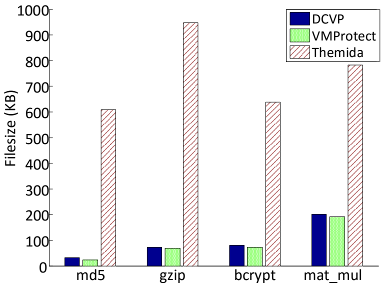


Fig. 12. The comparison of impact on file size (KB) with VMProtect and Themida.

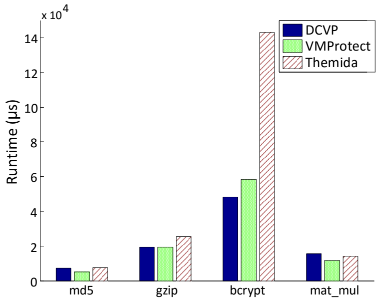


Fig. 13. The comparison of runtime performance (μs) with VMProtect and Themida.

of partitions increases, the frequencies of *opcodes* tend to be closer. When the partition number is 1, we can easily get the instruction that has the highest frequency is “load_r”, since it is the most commonly used instruction.

Finally, we use two commercial code virtualization protection system for comparison, VMProtect [VMProtect-Software 2016] and Themida [Oreans-Technologies 2016b]. Figure 12 shows the impact of the three virtualization protection system on the file size of the four target programs. The cost of Themida is much greater than the other two system, and the impact of DCVP and VMProtect is similar. This result should be related to the design of virtual instructions and handlers. Runtime overhead as shown in Figure 13. In general, the effects of the three protection systems are similar. Special, the runtime overhead of bcrypt that protected by Themida is far greater than the other target programs.

9. RELATED WORK

Software protection is used to protect the intellectual property encapsulated within software programs from been understood and modified, by transforming target program into a more obscure and hard-understanding one. The earliest published works in software protection can be traced back to 1980, Kent [Kent 1981] addressed the security requirements of software vendors: protection from software copying and modification. During the past decades, numerous approaches of software protection have been proposed, e.g. junk instructions, equivalent instructions [Nicolaou George 2009], packer (such as, “ASProtect”[13], “UPX”[14]), code encryption, control flow and data flow obfuscation [Collberg and Thomborson 2002; Liem et al. 2008; Ge et al. 2005], etc. These conventional approaches alone provide only limited obscurity, and one protection system usually integrates several of these and makes them work together to provide a certain level of obscurity.

In recent years, there have been many different code protection technology. Such as, control flow integrity [Zhang et al. 2013; Zhang and Sekar 2013] provide a strong protection against modern control flow hijacking attacks, including those based on Return-Oriented Programming (ROP). Code Randomization, Stephen Crane et al. [Crane et al. 2015] presents a practical, fine-grained code randomization defense, called Readactor, resilient to both static and dynamic ROP attacks. At the same time, the code reverse analysis technology is developing. Symbolic and concolic execution [Yadegari and Debray 2015] and Data flow tracking and tanit analysis [Jee et al. 2012; Balliu et al. 2014; Yadegari et al. 2015] can find important applications in a number of security-related

[13] Application Protection Tools for Software Developers, http://www.aspack.com/asprotect32.html.
[14] Ultimate Packer for eXecutables (UPX), http://upx.sourceforge.net/.

program analyses, including analysis of malicious code. This paper focuses on the code virtualization protection and its reverse analysis technology.

Code virtualized obfuscation, has recently been used to protect software from malicious reverse engineering [Oreans-Technologies 2016a; VMProtect-Software 2016; Fang et al. 2011; Wang et al. 2014; Wang et al. 2013]. In Section 2 and 3 we have already introduced the details of VM-based obfuscation and possible attacks. Existing researches on VM-based obfuscation focus on foiling reverse analysis. TDVMP [Wang et al. 2014] obfuscates `handlers` to generate multiple semantics-equivalent but appearance-different ones and embeds them into target program. At runtime, it will execute different instructions since the execution of each `handler` is chose from the several candidates randomly, i.e., the obfuscated program has a per-process execution diversity. This is especially useful for defeating dynamic analysis. Fang et al. [Fang et al. 2011] presented a multi-stage obfuscation method that iteratively transforms a program for many times in using different interpretations. DCVP focuses on invalidating the analysis knowledge about the semantics of bytecode instructions and preventing from quick and large scale reverse analyses, which is orthogonal to the above approaches and is complementary to them.

Has been also have some deobfuscation techniques of code virtualization was put forward. Representative like, Sharif et al. [Sharif et al. 2009] used dynamic data-flow and taint analysis to identify data buffers containing the bytecode program and extract the syntactic and semantic information about the bytecode instructions. Coogan et al. [Coogan et al. 2011] proposed an approach to identify instructions that related to system calls, and automatically extract an approximate dynamic trace of the original code. Yadegari et al. [Yadegari et al. 2015], proposed an approach to track the flow of inputs values, and then use semantics-preserving code transformations to simplify the logic of the instructions. These approach, however, or can only extract some execution characteristics of target program and can not restore the structure of the original code completely, or needed taint analysis to track and analyze the data flow, and this process may be impeded by enforcing dataflow obfuscation to the handling procedures [Liem et al. 2008]. These methods are limited more or less.

DCVP adopts ISR (Instruction Set Randomization) techniques while generating randomized and distinct virtual instruction sets. ISR has been used to prevent code injection attacks by randomizing the underlying system instructions [Barrantes et al. 2003; Kc et al. 2003; Portokalidis and Keromytis 2010]. In this approach, instructions are encrypted with a set of random keys and then decrypted before being fetched and executed by the CPU. ISR is effective for defeating code injection attacks but cannot prevent from reverse engineering attacks. As in our attack model, software programs are executed in a malicious host environment, where attackers are able to trace and log the decrypted instructions for later analysis. DCVP employs an approach similar to ISR while generating random virtual instruction sets, by changing the relationship between the opcodes and the virtual instructions [Kc et al. 2003], but it never "decrypts" the virtual instructions back into their original ones. Instead, DCVP uses `handlers` to interpret the virtual instructions. and the `handlers` of virtual instructions are more complex than their corresponding native instructions. Besides, DCVP uses the multiple partitions and has the different ISRs in a single program, making the reverse analyses even more difficult and tedious.

10. CONCLUSION

In this paper, we have introduced the internals of code virtualized obfuscation and the process of reverse analyzing a VM-obfuscated program. Although such obfuscation poses a great obstacle to an analyst, the presence of *analysis knowlege* can ease the work of the analyst. To mitigate the effect of reuse of *analysis knowlege*, we present D-

CVP, which disables the *analysis knowlege* by randomizing the semantics of bytecode instructions. Furthermore, DCVP partitions the generated virtual instructions into several parts, each been encoded differently. This defeats the attempts to infer the semantics of bytecode instructions based on the non-uniform frequencies of *opcodes*. Our evaluations show that DCVP is effective for mitigating the effect of reuse of *analysis knowlege*. As future work, we plan to: (1) use existing virtual instructions to switch the HAT instead of the specific `switch_HAT` for sake of the stealthiness of the switching; (2) work on data flow obfuscation to defeat semantics-based de-obfuscation.

ACKNOWLEDGMENTS

This work was partial supported by projects of the National Natural Science Foundation of China (No. 61373177, No. 61572402), the Key Project of Chinese Ministry of Education (No. 211181), the International Cooperation Foundation of Shaanxi Province, China (No. 2013KW01-02, No. 2015KW-003, No. 2016KW-034), the China Postdoctoral Science Foundation (grant No. 2012M521797), the Research Project of Shaanxi Province Department of Education (No. 15JK1734), the Research Project of NWU, China (No. 14NW28), and the UK Engineering and Physical Sciences Research Council under grants EP/M01567X/1 (SANDeRs), EP/M015793/1 (DIVIDEND).

REFERENCES

Musard Balliu, Mads Dam, and Roberto Guanciale. 2014. Automating Information Flow Analysis of Low Level Code. In *ACM Sigsac Conference on Computer and Communications Security*. 1080–1091.

Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2001. On the (im) possibility of obfuscating programs. In *Advances in Cryptology - CRYPTO 2001*. Springer, 1–18.

Elena Gabriela Barrantes, David H Ackley, Trek S Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*. ACM, 281–289.

B. D. Birrer, R. A. Raines, R. O. Baldwin, and B. E. Mullins. 2007. Program Fragmentation as a Metamorphic Software Protection. In *International Symposium on Information Assurance and Security*. 369–374.

Stanley Chow, Phil Eisen, Harold Johnson, and Paul C Van Oorschot. 2003. A white-box DES implementation for DRM applications. In *Digital Rights Management*. Springer, 1–15.

Christian S. Collberg and Clark Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *Software Engineering, IEEE Transactions on* 28, 8 (2002), 735–746.

Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and Communications Security (CCS)*. ACM, 275–284.

S. Crane, C. Liebchen, A. Homescu, and L. Davi. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. (2015), 763–780.

Nicolas Falliere, P Fitzgerald, and E Chien. 2009. Inside the jaws of Trojan.Clampi. *Rapport technique, Symantec Corporation* (2009).

Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. 2011. Multi-stage binary code obfuscation using improved virtual machine. In *Information Security*. Springer, 168–181.

FightingOreans'VM. 2008. Fighting Oreans' VM (Code Virtualizer flavour). (2008). http://www.woodmann.com/forum/showthread.php?t=12015-Fighting-Oreans-VM-(code-virtualizer-flavour)

Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. 2005. Control flow based obfuscation. In *Proceedings of the 5th ACM Workshop on Digital Rights Management (DRM)*. ACM, 83–92.

Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P Kemerlis, and Angelos D Keromytis. 2012. Adaptive defenses for commodity software through virtual application partitioning. In *Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS)*. ACM, 133–144.

Sudeep Ghosh, Jason Hiser, and Jack W Davidson. 2012. Replacement attacks against VM-protected applications. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 203–214.

Kangkook Jee, Georgios Portokalidis, Vasileios P Kemerlis, Soumyadeep Ghosh, David I August, and Angelos D Keromytis. 2012. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. *Proc of NDSS* (2012).

Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*. ACM, 272–280.

Stephen Kent. 1981. Protecting externally supplied software in small computers. (1981).

Clifford Liem, Yuan Xiang Gu, and Harold Johnson. 2008. A compiler-based infrastructure for software-protection. In *Proceedings of the third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 33–44.

Cullen Linn and Saumya Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and Communications Security (CCS)*. ACM, 290–299.

Benjamin Lynn, Manoj Prabhakaran, and Amit Sahai. 2004. Positive results and techniques for obfuscation. In *Advances in Cryptology-EUROCRYPT 2004*. Springer, 20–39.

Glafkos Charalambous Nicolaou George. 2009. *Applied binary code obfuscation*. Technical Report. http://www.intelligentexploit.com/articles/Applied-Binary-Code-Obfuscation.pdf

Oreans-Technologies. 2004-2016a. Code Virtualizer. (2004-2016). http://www.oreans.com/codevirtualizer.php

Oreans-Technologies. 2004-2016b. Themida. (2004-2016). http://www.oreans.com/themida.php

Matt Pietrek. 1994. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. (1994). https://msdn.microsoft.com/en-us/magazine/ms809762.aspx

Georgios Portokalidis and Angelos D Keromytis. 2010. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*. ACM, 41–48.

MD Preda and Roberto Giacobazzi. 2005. Control code obfuscation by abstract interpretation. In *Software Engineering and Formal Methods, 2005 (SEFM). Third IEEE International Conference on*. IEEE, 301–310.

Rolf Rolles. 2009. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*. USENIX Association, 1–1.

Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic reverse engineering of malware emulators. In *Security and Privacy (S&P), 2009 30th IEEE Symposium on*. IEEE, 94–109.

Paul C Van Oorschot. 2003. Revisiting software protection. In *Information Security*. Springer, 1–13.

VMProtect-Software. 2003-2016. VMProtect. (2003-2016). http://vmpsoft.com

Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. 2001. Protection of Software-Based Survivability Mechanisms. In *International Conference on Dependable Systems and Networks*. 0193–0193.

Huaijun Wang, Dingyi Fang, Guanghui Li, Na An, Xiaojiang Chen, and Yuanxiang Gu. 2014. TDVMP: Improved Virtual Machine-Based Software Protection with Time Diversity. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*. ACM, 4.

Huaijun Wang, Dingyi Fang, Guanghui Li, Xiaoyan Yin, Bo Zhang, and Yuanxiang Gu. 2013. NISLVMP: Improved Virtual Machine-Based Software Protection. In *Computational Intelligence and Security (CIS), 2013 9th International Conference on*. IEEE, 479–483.

Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. 2010. Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the 17th ACM conference on Computer and Communications Security (CCS)*. ACM, 536–546.

Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *The ACM Sigsac Conference*. 732–744.

Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. (2015), 674–691.

Zhang, Chao, Wei, Tao, Chen, Zhaofeng, Duan, Lei, Zou, and Wei. 2013. Practical Control Flow Integrity & Randomization for Binary Executables. *IEEE Symposium on Security & Privacy* (2013), 559–573.

Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Usenix Conference on Security*. 337–352.

Received October 2016; revised March 2017; accepted June 2017