Chosen-Instruction Attack Against Commercial Code Virtualization Obfuscators

Shijia Li[1], Chunfu Jia[1]‡, Pengda Qiu[1], Qiyuan Chen[1], Jiang Ming[2], Debin Gao[3]
[1]*Tianjin Key Laboratory of Network and Data Security Technology & College of Computer Science, Nankai University, China*
[2]*The University of Texas at Arlington, USA*
[3]*Singapore Management University, Singapore*
{sjli,pdqiu,nen9ma0}@mail.nankai.edu.cn, ‡cfjia@nankai.edu.cn, jiang.ming@uta.edu, dbgao@smu.edu.sg

***Abstract*—Code virtualization is a well-known sophisticated obfuscation technique that uses custom virtual machines (VM) to emulate the semantics of original native instructions. Commercial VM-based obfuscators (e.g., Themida and VMProtect) are often abused by malware developers to conceal malicious behaviors. Since the internal mechanism of commercial obfuscators is a black box, it is a daunting challenge for the analyst to understand the behavior of virtualized programs. To figure out the code virtualization mechanism and design deobfuscation techniques, the analyst has to perform reverse-engineering on large-scale highly obfuscated programs. This knowledge learning process suffers from painful cost and imprecision.**

In this project, we study how to automatically extract knowledge from the commercial VM-based obfuscator via a novel *chosen-instruction attack* (CIA) technique. Our idea is inspired by chosen-plaintext attack, which is a cryptanalysis attack model to gain information that reduces the security of the encryption scheme. Given a commercial VM-based obfuscator, we carefully construct input programs, proactively interact with the obfuscator, and extract knowledge from virtualized output programs. We propose using the anchor instruction and the guided simplification technique to efficiently locate and extract knowledge-related instructions from output programs, respectively. Our experimental results demonstrate that the modern commercial VM-based obfuscators are under the threat of CIA. We have discovered 760 anchor instructions and extracted 1,915 verified instruction mapping rules from the four most widely used commercial obfuscators. The extracted knowledge enables security analysts to understand virtualized malware and improve deobfuscation techniques. Besides, we also contributed the first fine-grained benchmark suite for systematically evaluating the deobfuscation techniques. The evaluation result shows that three state-of-the-art deobfuscation techniques are insufficient to defeat modern commercial VM-based obfuscators and can be improved by our extracted knowledge.

I. Introduction

Code virtualization is one of the strongest industrial-grade software obfuscation transformations [1], [2], which is inspired by system virtualization [3]. Instead of virtualizing the hardware and system components, code virtualization transforms selected instructions of the input program into a complex, customized process-level virtual machine (or interpreter). This process is analogous to the classical substitution ciphers. The VM-based obfuscator maps each original instruction to corresponding virtual handlers based on the internal customized mapping rules. Then, the obfuscator replaces original instructions with a customized virtual machine that uses a dispatcher to schedule virtual handlers for emulating equivalent semantics at runtime. The strength of customized VM depends on the complexity of the virtual components (i.e., virtual handlers, mapping rules, and VM structure). The sophisticated obfuscator can integrate with other obfuscation schemes (e.g., packing [4] and opaque predicates [5]) to protect virtual components. Since these internal mechanisms of the commercial obfuscator are undisclosed, attackers have to spend significant efforts in understanding them in a virtualized program. Therefore, the commercial VM-based obfuscator is effective in protecting legitimate programs from unauthorized usage [6], [7].

Code virtualization is also abused by malware authors [8], [9] and *advanced persistent threat* (APT) groups [10]–[12]. Intuitively, malware authors tend to use customized obfuscation to evade anti-virus scanning. For example, nearly 40%-50% of malware use customized binary unpacking instead of off-the-shelf packers [13], [14]. In contrast, malware authors prefer to use the commercial VM-based obfuscator rather than customized code virtualization. After analyzing 13,512 reports collected from malware analysis services [15]–[17] over the past three years, we find out that nearly 99% of virtualized malware samples are protected by commercial VM-based obfuscators, such as Themida [18], VMProtect [19], and Code Virtualizer [20]. To our best knowledge, FinFisher [10], [21] is the only known malware that uses the custom virtual machine. The reason is that code virtualization development is a time-consuming and error-prone process, and these costs may exceed the profits of malware. Malware authors need to make great efforts to design various virtual components properly. Otherwise, the customized virtual machine will be much weaker than the commercial ones. The report [10] shows that FinFisher's VM uses a simple decode-dispatch loop, a classic emulator structure, as the interpretation structure and only has 34 handlers. Security analysts can efficiently recover the semantics protected by this customized VM.

The complexity of code virtualization has also become an obstacle for reverse engineers and malware analysts to discover potentially malicious behaviors. To defeat the virtualized malware, researchers have proposed several techniques. These deobfuscation techniques simulate the process of manually reverse-engineering virtualized programs by skilled analysts. First, they locate the virtualized snippet (or area) from a long execution trace by detecting VM structure patterns (e.g., decode-dispatch loop structures [22]–[24] or context switch instructions [25]). Then, they use backward slicing [25], [26] or taint analysis [27], [28] to extract important instructions that are related to program behaviors (e.g., suspicious API calls) or

‡*Corresponding author: cfjia@nankai.edu.cn*

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24015
www.ndss-symposium.org

the native context. To further remove junk instructions introduced by semantic-based obfuscation [5], [29], they simplify the redundant instructions by performing symbolic execution [25], [30] or rule-based transformation [27]. Moreover, recent works also apply oracle-guided program synthesis [1], [31] to help analysts understand virtual handlers; they produce multiple simple formulas to represent the semantics of a virtual handler.

Unfortunately, the previous work has failed to specify that security analysts' knowledge is an essential factor when designing the deobfuscation techniques mentioned above. The existing deobfuscation research inevitably relies on knowledge-based (i.e., heuristics) algorithms. For example, both generic-deobfuscator [27] and VMHunt [25] adopt rule-based transformation (e.g., peephole optimization) to simplify junk instructions. This transformation requires experienced analysts to write appropriate rules. Another example is that VMHunt, the most effective research, uses two heuristics (*stack depth* and *execution transfer instruction*) to recognize virtualized snippets. To choose appropriate heuristics that work on different virtualized programs, the analyst has to learn the VM knowledge generated by different commercial obfuscators. Unfortunately, the internal mechanism of commercial VM-based obfuscator is still a black box. Especially, virtualized programs generated by the highly sophisticated commercial obfuscator are arduous to be understood. Hence the analyst inevitably has to examine large-scale virtualized programs to design generic heuristic algorithms. Since the analyst cannot systematically evaluate heuristics on various scenarios, the constructed heuristics suffer from poor generalizability and careless mistakes. If the obfuscator changes the obfuscation mechanism, the newly generated code virtualization can easily evade the deobfuscation technique guided by the outdated heuristics. For example, Xu et al. [25] have shown that the threaded interpretation structure [32], [33] impedes all prior deobfuscation techniques based on identifying the decode-dispatch loop of the interpreter. To find new heuristics, the analyst has to analyze the large-scale virtualized programs again. Therefore, the major challenge faced by researchers is how to automatically extract reusable knowledge from virtualized programs.

In this paper, we propose the *chosen-instruction attack* (CIA), a new approach to automatically extract reusable knowledge from commercial VM-based obfuscators. Similar to the *chosen-plaintext attack* (CPA) attacker [34], the CIA attacker also constructs arbitrary input programs to generate corresponding virtualized programs. The attacker aims to extract essential reusable knowledge (e.g., VM structure and mapping rules) from output programs. To reduce the cost of analyzing virtualized programs, we propose using the *anchor* instruction: it can separate the virtualized instructions related to original instructions from other irrelevant areas of the virtualized program. In each input program, the CIA attackers insert knowledge leaking code between two *anchor* instructions. Then, they can quickly locate virtualized knowledge leaking code from a long execution trace by searching the anchor. But these extracted instructions may still contain plenty of junk code. To further extract specific knowledge-related instructions, we design the *guided simplification* technique. It uses the semantics of original instructions as a guide to extract related virtualized instructions. The correctness of extracted instructions can be verified by comparing the symbolic formulas with the original instructions. Therefore, the CIA attacker can efficiently learn and reuse knowledge from the extracted virtualized instructions. The extracted knowledge can help security professionals to understand virtualized programs, select appropriate heuristics, and systematically evaluate deobfuscation techniques.

We have evaluated our CIA model on the multiple versions of four advanced commercial VM-based obfuscators (VMProtect [19], Code Virtualizer [20], Themida [18], and Obsidium [35]). The experiment shows that the existing commercial obfuscators are under the threat of CIA. We find out 760 anchor instructions from Intel ISA (detailed in Sec. VII-A) and extract 1,915 customized mapping rules from obfuscators. Based on the extracted knowledge, we construct the first fine-grained benchmark dataset which contains 5,745 virtualized programs. We have evaluated the correctness of three state-of-the-art deobfuscation approaches (VMHunt [25], generic-deobfuscator [27], and Syntia [1]). The evaluation results show that they are suffering from imprecision due to the lack of systematic evaluation. For example, we find out the decoy context switch introduced by Code Virtualizer and Themida causes VMHunt to miss correct virtualized snippets.

In summary, we make the following contributions:

- We propose chosen-instruction attack, a general approach to automatically extract reusable knowledge from VM-based obfuscators. We propose using the anchor instruction to locate real virtualized instruction sequences from a tedious execution trace, and guided simplification to extract knowledge-related instructions.
- We show that CIA threatens the advanced commercial VM-based obfuscators. We find 760 anchor instructions and extract 1,915 verified instruction mapping rules from four commercial obfuscators. The experimental results also show that the complexity of the obfuscation scheme used in commercial VM-based obfuscators far exceeds the ability of the existing deobfuscation research.
- We construct the first fine-grained benchmark dataset for evaluating the correctness of their deobfuscation techniques. We have evaluated three state-of-the-art deobfuscation techniques and demonstrated their weakness. We release the source code of CIA and benchmark dataset at (https://github.com/chosen-instruction-attack/).

II. Background and Motivation

In this section, we first outline the workflow of the commercial VM-based obfuscators. Then we summarize the knowledge (or heuristics) used in existing deobfuscation research works. We also discuss the challenges faced by researchers in designing the deobfuscation techniques.

A. Commercial VM-based Obfuscator

For each user-selected input program P, the commercial VM-based *obfuscator* $\mathcal{O}$ generates an obfuscated program $\mathcal{O}(P)$. The transformation process is shown in Fig. 1. According to the mapping rules, each selected native instruction is linearly transformed into a combination of virtual handlers. Then, $\mathcal{O}$ assembles the generated virtual handlers into VM structures and other obfuscation schemes. Finally, $\mathcal{O}$ uses a

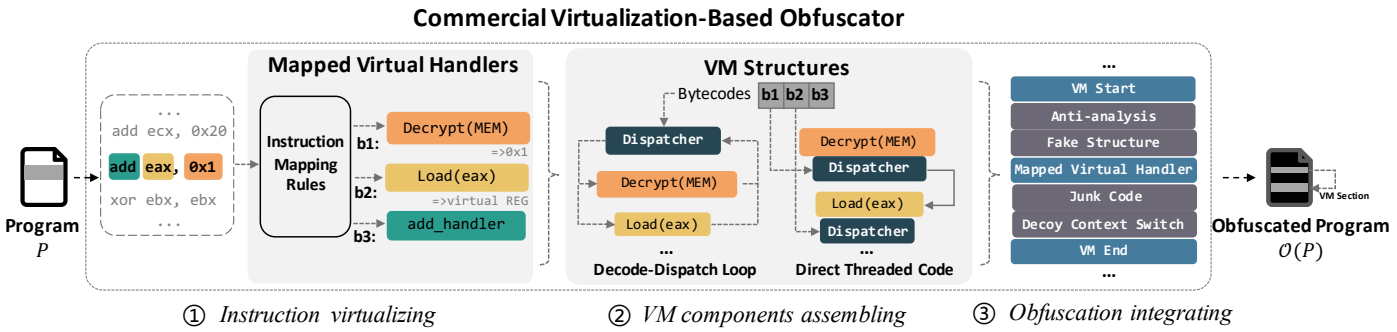


Fig. 1: The transformation process of commercial VM-based obfuscators. The obfuscator transforms each instruction selected in P into virtualized instructions. Then, virtualized instructions are combined into the VM structure and other obfuscation techniques.

custom VM to replace the selected area in P. Here, we discuss two essential components that are reused in each $\mathcal{O}(P)$.

Virtual handler and mapping rules. Similar to the micro-operations used to implement complex instructions in the CPU, the virtual handler is a minimal operator of the process-level VM. It is composed of multiple native instructions and is used to operate runtime context. Since most VMs use the RISC-like structure [1], [23], the semantic of single native instruction is emulated by multiple handlers. According to the customized mapping rules, the obfuscator transforms each selected native instruction into a combination of virtual handlers. For example, the handler `Decrypt` and `Load` shown in Fig. 1 store immediate `0x1` and the original value of register `eax` into virtual context. Then, the operation handler `add_handler` will perform addition. After the VM is executed, the calculation result stored in the virtual context will overwrite the native context. The virtual handler generated by sophisticated obfuscators is usually highly obfuscated to hide actual calculation behaviors. For example, VMProtect applies data encoding [29], [36] and metamorphism [37], [38] to protect the immediate operand and virtual program counter. Besides, the obfuscator only embeds required handlers that are mapped from original instructions into $\mathcal{O}(P)$, preventing analysts from discovering all kinds of virtual handlers from a single virtualized program.

VM structure. As shown in Fig. 1, it is used to establish the virtual environment and dispatch virtual handlers. To isolate the native area from the virtual environment, the virtual machine typically begins with context switch instructions to save the native context (e.g., registers) [25]. After initializing the virtual environment, the virtual machine dispatches handlers by fetching and decoding the bytecode [39], [40]. The traditional interpreter uses a central loop to fetch the bytecode and schedule corresponding handlers. But this loop structure is a conspicuous pattern that attackers can easily recognize. The advanced commercial obfuscator turns to adopt the direct threaded code structure [32], [33], [41], which merges the dispatcher into each virtual handler.

Furthermore, the sophisticated obfuscator will integrate multiple obfuscations to hinder analysts from locating virtual components. For example, the obfuscator can randomize and encrypt the bytecode [1] to conceal the control flow of virtual instructions from static analysis. The obfuscator can also insert multiple fake decode-dispatch loops into the virtualized program for camouflaging the real VM dispatcher [25]. Even worse, we notice that the commercial obfuscator also virtualizes other integrated obfuscations such as binary packing. It is difficult for the analyst to locate the real virtualized snippets that mapped from the original P, rather than the virtualized integrated obfuscations. Another approach to increase the cost of reverse-engineering is to diversify VM structures and mapping rules. For example, the latest version of Code Virtualizer and Themida provides six VM options. But obfuscator developers also must make great efforts to design various virtual components correctly.

TABLE I: Knowledge used in the existing research

Categories	Knowledge Used in Related Research
Detect VM Structure	Decode-dispatcher loop [22]–[24], [43], Call/Return pattern [26], Handler boundary [1], [31], Context switch [25]
Recover/Use Mapping Rules	Human intelligence [1], [23], [31], Virtual ISA [44], Dataset for evaluation
Simplify Obfuscation Schemes	Rule-based transformation [24]–[27], [45], Behaviour of concealed conditional jump [30]

B. Knowledge-based Deobfuscation

The design of the existing deobfuscation techniques relies on the researcher's knowledge of the code virtualization mechanism. To correctly simplify the virtualized program, the analyst needs the ability to distinguish between kernel *virtualized instructions* and *irrelevant (junk) instructions*. Note that the internal mechanism of commercial obfuscator is not publicly disclosed. Similar to the Man-At-The-End (MATE) [42] attackers, the analyst also performs reverse-engineering on the large-scale $\mathcal{O}(P)$ (or $\mathcal{O}$) to learn the knowledge and design appropriate heuristics. As shown in Table I, the knowledge used in existing research can be divided into three categories.

Detect VM structure. Most existing works have to locate virtualized snippets from the long execution trace based on the pattern of the VM structure. The prior research [22], [24] focuses on recognizing the conspicuous loop structure of the VM dispatcher from the execution trace. However, they are easily evaded by fake decode-dispatch loops and cannot recognize the VM that uses the direct threaded code [25]. VMHunt [25] turns to detect the context switch patterns for locating the virtualized snippet boundary. It pairs the context switch instructions based on two heuristics (*stack depth* and *execution transfer instruction*). Specifically, it assumes that the

context switch instructions must be in the same stack depth and followed by a control flow transfer instruction. In addition, Syntia [1] detects the boundary of virtual handlers based on the indirect branches.

Recover/Use mapping rules. To find essential (but not junk) virtualized instructions from $\mathcal{O}(P)$, the analyst must understand the mapping rules. Most research assumes the analyst is experienced in distinguishing between virtualized instructions and junk instructions. For example, the analyst has to manually construct inputs and select parameters when using the program synthesis [1]. Another example is that the replacement attack adversary [44] is assumed to be familiar with the virtual ISA to overwrite virtualized instructions correctly. Besides, the existing research also requires the virtualized samples as the dataset for evaluation. The mapping rules can help create a ground-truth dataset to systematically evaluate the correctness of a deobfuscation work. However, the mapping rules used in commercial obfuscators are undisclosed, and they have not been closely investigated.

Simplify obfuscation schemes. Since the obfuscation is hard to be recognized and removed, the existing research [24]–[27], [45] has to rely on the rule-based transformation (e.g., dead code elimination) to remove potential junk instructions. Even if the data dependence analysis can extract specific behavior (e.g., system call or native context) related instructions, the extracted result still contains junk instructions introduced by semantic-based obfuscation. For example, generic-deobfuscator [27] relies on more than five rule-based transformations to remove junk instructions from the extracted input-output flow. Another example is that VMHunt [25] uses a peephole optimizer to remove junk instructions for revealing consecutive context switch instructions. However, the selection of heuristics in existing research also requires tremendous efforts. Due to the lack of a systematic evaluation, the generality of these heuristics against various obfuscation schemes remains unknown.

C. Challenges of knowledge-based deobfuscation

Even for an experienced analyst, it is challenging to learn complete knowledge about the internal mechanism of obfuscators. Since the virtualized instructions only can be decrypted at runtime [1], the existing research relies on dynamic analysis to record execution traces. But the code virtualization and other integrated obfuscation schemes bloat the trace size dramatically. For example, we find out that the execution trace of a single instruction virtualized by Code Virtualizer's `tiger black` VM is nearly 172,663 lines on average. It is typically difficult for the analyst to examine the tedious execution trace of virtualized programs. The painful cost of finding appropriate heuristics is severely impeding the development of an effective deobfuscation method. Note that heuristics are usually only suitable for specific versions of obfuscators. If the internal obfuscation mechanism is changed, the newly generated virtualized program can easily circumvent the deobfuscation techniques that use outdated heuristics. Our experiment results indicate that the state-of-the-art deobfuscation research is ineffective in simplifying the virtualized programs generated by the latest commercial VM-based obfuscators. Therefore, how to automatically extract complete knowledge from commercial VM-based obfuscators is a great challenge.

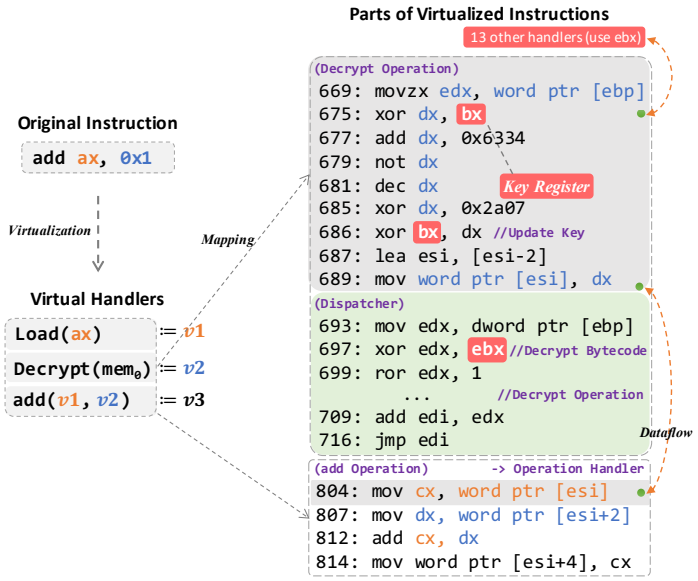


Fig. 2: A simplified example of virtualized `add ax, 0x1` generated by VMProtect 3.5 [19]. The virtualized immediate is decrypted from memory using the key stored in register `ebx`.

Furthermore, how to evaluate the deobfuscation techniques is another key obstacle. The existing research lacks a fine-grained benchmark for systematic evaluation. Most existing research only compares the representation of instructions such as the similarity of control flow graphs [22], [27] or instruction forms [26]. It cannot guarantee the semantic of simplified instructions is equivalent to the original program. VMHunt is the first work that verifies the correctness of the simplified instructions based on symbolic execution. But it uses an incomplete dataset for evaluation, which contains limited hand-written programs with a simple function call and `if-else` structure. This insufficient dataset cannot cover various mapping rules of the commercial obfuscator. Without the fine-grained benchmark, the analyst cannot systematically evaluate the effectiveness of heuristics. If some virtualized instructions are incorrectly simplified, a potential malicious behavior may be detected as benign.

Fig. 2 shows an example of virtualized instructions generated by VMProtect. For simplicity, junk instructions in the handler are not listed. The original instruction `add ax, 0x1` is transformed to three virtual handlers. The immediate operand is decrypted from memory based on the key stored in register `ebx`. But the key register `ebx` is also used to decrypt the bytecode. Note that data dependence analysis tracks every instruction that contributes to the target. It will be misled to track 13 irrelevant handlers that all use the key register. When facing the virtualized malware in the wild, the data dependence analysis will record more irrelevant virtual instructions. It increases the difficulty for analysts to understand the simplified results. Besides, the analyst also cannot discover the mistakenly simplified results without a fine-grained benchmark. For example, we discover that generic-deobfuscator [27] mistakenly transforms the instruction `add cx, dx` at line 812 to `mov cx, 0x2` (detailed in Sec. VII-D). It will cause the taint analysis cannot discover the data flow related to the instruction at line 807. Meanwhile, the coarse-grained binary similarity analysis (e.g., CFG comparison algorithm [22], [27])

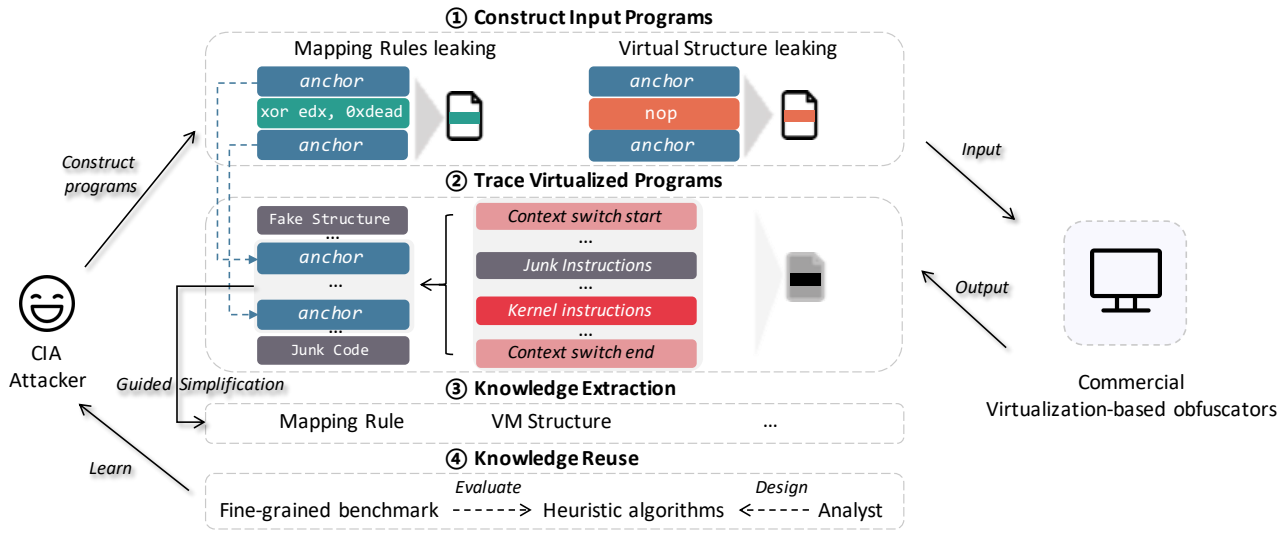


Fig. 3: An overview of our chosen-instruction attack.

is prone to false positives. Since it cannot discover the semantic differences caused by mistakenly simplified instructions, the correctness of the deobfuscation techniques that are only evaluated by the coarse-grained methods is unreliable.

III. Chosen-instruction Attack

A. Overview

The *chosen-instruction attack* is inspired by the *chosen-plaintext attack* (CPA) [34], a well-known model to evaluate the strength of cryptographic systems. The CPA attackers are given access (i.e., construct arbitrary plaintext-ciphertext pairs) to the encryption oracle. They can learn knowledge about encryption algorithms by analyzing plaintext-ciphertext pairs. Intuitively, the CPA model is similar to the scenario that an analyst wants to extract knowledge from commercial VM-based obfuscators. Security analysts can construct arbitrary programs as the input of obfuscators. Then, they can examine the input and output pairs to learn knowledge about the internal mechanism of obfuscators. Therefore, motivated by the idea of CPA, we propose a new chosen-instruction attack approach to extract knowledge from the VM-based obfuscator. Unlike the ciphertext that should not expose the form of plaintext, the virtualized program must be semantically equivalent to the original program. Although the obfuscator can randomly insert junk instructions, the essential components (VM structures and mapping rules) are still reused in each obfuscated program. The CIA attacker can construct a special program to leak knowledge from the obfuscator. Because the knowledge-related instructions that CIA attacker cares about are hidden in enormous highly obfuscated instructions. We propose the anchor and guided simplification to help the CIA attacker to extract reusable knowledge.

The workflow of the chosen-instruction attack is shown in Fig. 3. CIA attackers first construct a knowledge leaking program as the input of the obfuscator. They can generate different categories of Intel instruction (detailed in Sec. VI) and choose single instruction as knowledge leaking code to extract reusable knowledge. For example, they can insert single `xor edx, 0xdead` as knowledge leaking code for understanding the mapping rules of `xor REG, IMM`. To decrease the cost of locating virtualized instructions, they can put the knowledge leaking code between anchor instructions (detailed in Sec. IV-A). Then, the CIA attacker only needs to record the execution trace between two anchor instructions. Based on the guided simplification (detailed in Sec. V-A), CIA attackers can extract kernel instructions that contribute to interesting knowledge. By constructing and analyzing sufficient knowledge leaking programs, the attackers can extract knowledge (mapping rules, VM structures, and obfuscation schemes) from the obfuscator. For example, they can traverse the target machine ISA (e.g., Intel x86 ISA) to construct instructions as knowledge leaking code and extract the corresponding kernel virtualized instructions. Then, they can summarize complete mapping rules. Since the semantics of original instructions are known, the attacker can directly verify the correctness of mapping rules. Similarly, the CIA attacker can also learn about VM structures and other obfuscation schemes at a low cost. The extracted knowledge can be reused in multiple scenarios. For example, based on the extracted mapping rules, the CIA attacker can construct a fine-grained benchmark to evaluate the correctness of deobfuscation techniques.

B. Formal Definition of CIA

The CIA model is a tuple $(\mathcal{O}, \mathcal{A}, P, T, \mathtt{s}, \mathcal{K}, \mathtt{c})$, where

- $\mathcal{O}$ is a commercial VM-based obfuscator.
- $\mathcal{A}$ is a CIA attacker.
- P is the program constructed by the CIA attacker.
- T is the execution trace.
- $\mathtt{s}$ is a user-selected protection configuration such as packing and mutation. CIA attackers can choose different obfuscation options for analyzing specific obfuscation schemes. For example, they can select only the code virtualization option, without integrating any other anti-analysis techniques, to extract mapping rules.
- $\mathcal{K}$ is the reusable knowledge that CIA attackers want to extract. It includes various $\mathtt{k}$ (i.e., mapping rules, VM structure, and other obfuscation schemes).
- $\mathtt{c}$ is the knowledge leaking code used to trigger $\mathcal{O}$ to generate virtualized code that contains $\mathtt{k}$.

```
1 ... // VM execution
2 pop eax, ecx, ebp, edx, ebx, esi, edi
3 // exit virtualization
4 cmpxchg eax, eax // anchor
5 // reenter virtualization
6 push edi, esi, ebx, edx, ebp, ecx, eax
7 ... // VM execution
```

Fig. 4: An example of anchor `cmpxchg` in trace.

Under the CIA model, the adversary $\mathcal{A}$ aims to learn the knowledge by interacting with VM-based obfuscator $\mathcal{O}$. They can construct arbitrary program P that inserted special knowledge leaking code $\mathtt{c}$ between anchor instructions. Then, the $\mathcal{A}$ generates obfuscated $\mathcal{O}_{\mathtt{s}}(P_{\mathtt{c}})$ under the selected protection options $\mathtt{s}$. Based on the anchor, the $\mathcal{A}$ can record reduced execution trace $T^s_{\mathtt{c}}$ of $\mathcal{O}_{\mathtt{s}}(P_{\mathtt{c}})$. Guided by the information of $\mathtt{c}$, the $\mathcal{A}$ can extract part of knowledge $\mathtt{k} \in \mathcal{K}$ (e.g., instruction mapping rules) from $T^s_{\mathtt{c}}$. We define an experiment for VM-based obfuscator to represent the CIA model. The CIA experiment $\mathsf{VirO}^{CIA}_{\mathcal{A},\mathcal{O}_{\mathtt{s}}}(P)$:

1) *A configuration scheme* $\mathtt{s}$ *is chosen by* $\mathcal{A}$*. The adversary* $\mathcal{A}$ *is given oracle access to* $\mathcal{O}_{\mathtt{s}}(\cdot)$.
2) *The adversary* $\mathcal{A}$ *constructs a set of knowledge leaking programs* $P_{\mathcal{C}} = \{P_{\mathtt{c}_1}, P_{\mathtt{c}_2}, ..., P_{\mathtt{c}_n}\}$ *for learning the knowledge* $\mathtt{k}$.
3) *An arbitrary program* $P_{\mathtt{c}} \in P_{\mathcal{C}}$ *is chosen, and then a trace* $T^s_{\mathtt{c}} \leftarrow \mathcal{O}_{\mathtt{s}}(P_{\mathtt{c}})$ *is recorded and given to* $\mathcal{A}$.
4) *The output of the experiment is defined to be 1 if* $simplify(T^s_{\mathtt{c}}) \approx \mathtt{k}$*, and 0 otherwise.*

In case $\mathsf{VirO}^{CIA}_{\mathcal{A},\mathcal{O}_{\mathtt{s}}}(P) = 1$, we say that $\mathcal{A}$ succeeded. When the $\mathtt{k}$ is the mapping rules, the $\mathcal{A}$ can use single instruction $\mathtt{I}$ as $\mathtt{c}$ and generate $T^s_{\mathtt{I}}$. If the simplified result $simplify(T^s_{\mathtt{I}})$ is semantically equal to $\mathtt{I}$, it means $\mathcal{A}$ successfully extracts the knowledge of $\mathtt{I}$. After the $\mathcal{A}$ traverses all instructions in Intel ISA and gets the correct simplified result, they can extract a complete $\mathtt{k}$ of mapping rules. Similarly, when the $\mathtt{k}$ is the VM structures or obfuscation schemes, the $\mathcal{A}$ can use `nop` as $\mathtt{c}$ and generate $T^s_{\mathtt{c}}$. The $\mathcal{A}$ can learn $\mathtt{k}$ when different $simplify(T^s_{\mathtt{c}})$ is similar.

IV. CIA Sample Construction

In this section, we demonstrate the process of constructing $P_{\mathtt{c}}$ and generating $T^s_{\mathtt{c}}$. We propose the *anchor* instruction to locate the virtualized instructions that contain knowledge $\mathtt{k}$ from T. The *anchor* is inserted into $P_{\mathtt{c}}$ during the construction process. We also illustrate three types of knowledge leaking code $\mathtt{c}$ to extract different $\mathtt{k}$.

A. Anchor Instruction

Accurately locating the virtualized instructions that contain implicit knowledge $\mathtt{k}$ is still an open challenge. The heuristics used in existing works easily become invalid when the pattern of VM changed (discussed in Sec. II-C). In contrast, the CIA attacker $\mathcal{A}$ can introduce a unique pattern into P for efficiently locating the boundary of kernel virtualized instructions. It is a reasonable assumption since the $\mathcal{A}$ has oracle access to $\mathcal{O}$. Therefore, we propose a special type of instruction, called *anchor*, to create landmarks in $T(\mathcal{O}(P))$.

```
1 void KnowledgeLeaking(){
2     VIRTUALIZER_START // VM macro
3     __asm(
4     "cmpxchg eax, eax;" // anchor
5     "xor ebx, 0xdead;" // knowledge leaking code
6     "cmpxchg eax, eax;"
7     );
8     VIRTUALIZER_END // VM macro
9 }
```

Fig. 5: An example of the knowledge leaking code. The `xor ebx, 0xdead` can be replaced with other instructions.

The *anchor* is an instruction that is not (able to be) virtualized by $\mathcal{O}$. It is not included in the mapping rules and will be kept in $\mathcal{O}(P)$. Specifically, an instruction must satisfy two conditions to become an *anchor*:

- **Format Preserving.** The instruction representation (prefix, mnemonic, and operands) of virtualized *anchor* in $\mathcal{O}(P)$ is identical to the original *anchor* in P. It makes sure that the $\mathcal{A}$ can efficiently locate anchor from $T(\mathcal{O}(P))$.
- **Virtualization Repelling.** To execute instructions that cannot be virtualized, the VM has to be suspended (or terminated) and switched to the native environment. Hence the $\mathcal{O}$ is forced to generate context switch instructions around the anchor in $\mathcal{O}(P)$. It can help the $\mathcal{A}$ to efficiently extract virtualized instructions that contribute to native context, rather than painfully examine the relationship between virtual context and native context.

The *anchor* is intended to trigger the $\mathcal{O}$ statically generate context switch instructions. For example, if we insert *anchor* `cmpxchg eax, eax` in P, we can get a trace of $\mathcal{O}(P)$ illustrated in Fig. 4, which has been trimmed for better presentation. The anchor instruction at line 4 preserves the original format. It also triggers *virtualization repelling* at lines 2 and 6. Since we can remove the *anchor* during the dynamic instrumentation, the *anchor* will not pollute the original execution of P or $\mathcal{O}(P)$. Therefore, any instruction that satisfies the above conditions can be safely chosen as *anchor*.

To discover potential *anchor* instructions from $\mathcal{O}$, we traverse the Intel ISA to construct various types of instruction $\mathtt{I}$ and generate corresponding $T(\mathcal{O}(P_{\mathtt{I}}))$. Then, we search for anchors that meet the above conditions (detailed in Sec. VII-A).

B. Knowledge Leaking Code

As discussed in Sec. III, the CIA attacker $\mathcal{A}$ can construct different types of knowledge leaking code $\mathtt{c}$ to extract knowledge from the obfuscator. To locate the virtualized $\mathtt{c}$ from $T(\mathcal{O}(P_{\mathtt{c}}))$, the $\mathcal{A}$ can insert $\mathtt{c}$ between the anchor instructions. For example, as shown in Fig. 5, the knowledge leaking code `xor ebx, 0xdeadbeef` is surrounded by two anchor `cmpxchg eax, eax`. These assembly instructions are embedded within the source code of P. The VM macros are used to tell $\mathcal{O}$ the instructions that need to be virtualized (lines 4-6). Searching for instructions between the anchor `cmpxchg` in the recorded $T(\mathcal{O}(P_{\mathtt{c}}))$, the $\mathcal{A}$ can find virtualized instructions used to emulate `xor ebx, 0xdeadbeef`. For extracting different types of knowledge, the $\mathcal{A}$ can diversify the $\mathtt{c}$.

Considering the essential knowledge discussed in Sec. II-B, the types of `c` can be grouped into three categories.

Mapping rules leaking. To precisely extract the instruction mapping rules between the original and virtualized instructions, the $\mathcal{A}$ can use a single instruction as `c`. For example, the knowledge leaking code `xor ebx, 0xdead` shown in Fig. 5 is used to leak mapping rules of `xor REG, IMM`. The $\mathcal{A}$ can replace it with other instructions to extract corresponding mapping rules.

Transformation strategy leaking. Considering the $\mathcal{O}$ may apply different transformation strategies to expand the mapping rules, the $\mathcal{A}$ should diversify the generation of $\mathcal{O}(P_c)$ to discover every possible transformation (detailed in Sec. V-B1).

VM structure leaking. For extracting the VM structure such as context switch, the T_c^s should contain a complete virtual machine. The $\mathcal{A}$ can use `nop` to replace `xor ebx, 0xdead` shown in Fig. 5. The generated T_c^s will only contain a pure VM without any other virtualized operation. Then, the $\mathcal{A}$ can extract VM structure from T_c^s.

After constructing the $\mathcal{O}(P_c)$, we use Intel Pin [46] to record the execution trace between anchor instructions. The recorded trace contains essential information of each executed instruction. It includes the memory address, length, and runtime information such as values of registers and accessed memory. The size of trace files is greatly reduced by the anchor. For example, the trace size of single native instruction virtualized by Code Virtualizer's `tiger black` VM is reduced from 172,663 to 23,421 instructions on average. Then, the CIA attacker can perform guided simplification on the trace.

V. Knowledge Extraction and Reuse

In this section, we present how we extract reusable knowledge from the T_c^s based on guided simplification. The knowledge we aim to extract includes mapping rules and VM structure. They are essential for analysts to design deobfuscation techniques (discussed in Sec. II-B). To extract different types of knowledge, CIA attackers use corresponding knowledge leaking code `c` to generate T_c^s and apply guided simplification to extract the related virtualized instructions. The extracted knowledge can assist security analysts to understand virtualized instructions and design appropriate deobfuscation techniques.

A. Guided Simplification

Since the T_c^s is still obfuscated, the CIA attacker $\mathcal{A}$ has to extract kernel virtualized instructions related to knowledge. Note that the information (e.g., semantic) of inserted knowledge leaking code `c` is transparent to the $\mathcal{A}$. Guided by this information, the simplification technique can precisely extract instructions related to knowledge `k`. The overview of our approach is shown in Fig. 6. Our guided simplification techniques include the following stages.

Trace Slicing. To extract the knowledge-related instructions, we perform slicing on the T_c^s. Our slicing algorithm is based on the enhanced backward slicing [47] used in VMHunt. It is reliable in handling virtualized instructions. When the `c` is used to extract mapping rules, the source of slicing is *native context*. As discussed in Sec. II-A, the context switch instructions overwrite the *native context* with the calculation results stored in virtual context. Therefore, to extract kernel virtualized instructions, we can perform backward slicing starting from the native context related to the embedded `c` (e.g., the destination register `ax` in Fig. 2). The slice $\mathcal{S}_c$ contains every virtualized instruction that contributes to emulate the semantic of the original `c`. Considering the virtualized instructions use the virtual memory environment to transfer values, the $\mathcal{S}$ will inevitably contain instructions of the VM structure such as handler dispatching instructions. For example, the computation of virtual memory index (e.g., `[esi]` in Fig. 2.) will be introduced into $\mathcal{S}$ by index-based slicing. Therefore, we perform the forward slicing to shrink the trace after normalization. It uses each operand of `c` as the source. When the `c` is used to extract VM structure, the source of slicing can be grouped into two types: *native context* and *virtual bytecode*. We can extract the context switch instructions by slicing all general registers when `c` is `nop` (shown in Fig. 5). Since the virtual bytecode guides the VM scheduling handlers, we can perform forward slicing to extract VM structure instructions.

Normalization. The operands of virtualized instructions are used to interact with the virtual environment (e.g., the operands of `mov cx, word ptr [esi]` in Fig. 2.). Especially, virtual registers (or memory) can be randomly allocated by the virtual machine at runtime. Hence we normalize the virtualized instructions into instructions that interact with native contexts. Similar to the generic-deobfuscator [27], we rewrite instructions in $\mathcal{S}$. Furthermore, we also use semantic preserving transformation (e.g., peephole optimization) to simplify the instructions after $\mathcal{S}$ is normalized.

B. Reusable Knowledge Extraction

In this section, we demonstrate how we perform guided simplification to extract `k` from commercial VM-based obfuscator $\mathcal{O}$ under the CIA model. The overview of the workflow is also shown in Fig. 6. As mentioned earlier, the CIA attacker focuses on two essential `k` (i.e., mapping rules and virtual machine structure).

Algorithm 1: Mapping Rules Extraction

Input: instruction `I` of Intel ISA (except anchor)
Result: virtual handlers mapping of `I`

Embed `I` as `c` in P;
T_I^s = trace of $\mathcal{O}^s(P_I)$ between anchor instructions;
`/* Guided Simplification */`
dst:= destination operand of `I`;
S:= BackwardSlicing(T_I^s, dst);
$kernel$:= NULL;
S_n:= Normalization(S);
for $oper \in$ *operands of* `I` **do**
  $kernel \leftarrow kernel$+ForwardSlicing(S_n, $oper$);
end
Exp_{new}:= Lift($kernel$);
Exp_{origin}:= Lift(`I`);
if Exp_{origin} *equal to* Exp_{new} **then**
  return (I, $kernel$, Exp_{new});
else
  return False;
end

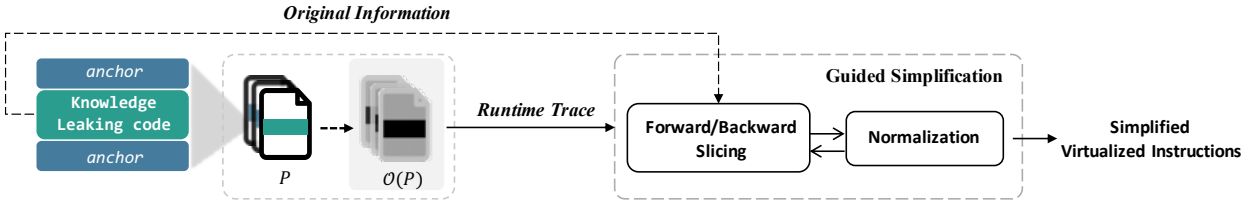


Fig. 6: The knowledge extraction process of CIA.

1) Mapping Rule: To recover the semantic of virtualized instructions, analysts need to understand the mapping from the source instruction `I` to kernel virtualized instructions, which are the minimum required instructions to emulate the semantic of the source instruction (e.g., the virtualized instructions mapped from `xor edx, ecx` shown in Appendix Fig. 10). Note that the obfuscator applies different mapping rules according to the type of source native instructions and randomly inserts junk instructions into the generated semantically equivalent virtualized instructions. We construct different P_c and analyze corresponding T_c, where `c` is a variety of single native instruction `I`. Then, we summarize each `I` and corresponding kernel virtualized instructions to construct the mapping rules, which can be simply represented by a tuple (`I`, *kernel*). This process is demonstrated in Algorithm 1. By traversing the Intel ISA, we choose a virtualizable instruction `I` as knowledge leaking code `c` each time. Then, we apply the guided simplification on record trace T_I of program $\mathcal{O}(P_I)$. We perform guided simplification to extract $\mathcal{S}$ that contains the kernel virtualized instructions related to `I`. To verify and reuse the extracted assembly instructions, we lift them into symbolic formulas and compare them with the original instruction `I`.

Transformation Strategy. The obfuscator usually combines multiple transformation strategies to complicate the mapping rules. Intuitively, the mapping transformation strategy could be divided into the following four categories.

- **One to One (O2O).** The O2O strategy is a basic transformation. It means that the obfuscator emulates the operation of single original instruction based on a single virtual handler, which contains a kernel virtualized instruction that uses the same mnemonic as the original instruction. An example is the native instruction `add ax, 0x1` and corresponding virtualized instructions shown in Fig. 2. In the $add(v1, v2)$ operation handler, after the VM loads the operands of original instruction into `cx` and `dx` (at line 804 and 807), the virtualized instruction at line 812 uses the same mnemonic to perform addition.
- **One to Multiple (O2M).** The O2M strategy is that the operation of single original instruction can be transformed into a combination of virtual handlers. Besides, the obfuscator may also have multiple mapping rules for the same operation. For example, VMProtect converts the `xor` instruction to multiple combinations of logical `NOR` and `NAND` handlers rather than a single `XOR` handler. To detect the O2M strategy, we use the same `I` as `c` and generate multiple $\mathcal{O}(P_c)$. Then, we perform Algorithm 1 on each $\mathcal{O}(P_c)$ and compare whether the simplification results are the same.
- **Multiple to One (M2O).** Similar to the optimization strategy, the M2O strategy can use a single handler to emulate the semantics of multiple instructions. Hence we use multiple redundant instructions as `c` and detect the combination of virtual handlers. For example, if $\mathcal{O}$ uses virtualized instruction `add eax, 0x10` to replace the knowledge leaking code `add eax,0x20; sub eax,0x10;`, it means that $\mathcal{O}$ adopts M2O strategy.
- **Multiple to Multiple (M2M).** Since the M2M strategy is a combination of M2O and O2M, the above strategies have already covered this scenario.

2) Virtual Machine Structure: As discussed in Sec. II-B, most deobfuscation techniques rely on two components of VM structure (i.e., context switch and dispatcher structure) to locate the virtualized area. In the following, we introduce the process of extracting virtualized instructions contributed to the context switch and classifying the type of dispatcher structures, respectively.

Context Switch. The context switch instructions transfer register values when entering or exiting the virtual environment. This feature is considered as the boundary of the virtualized snippet by VMHunt. To extract context switch instructions, we use the `nop` instruction as `c`. It can trigger $\mathcal{O}$ to generate pure VM without any other virtualized operation. When applying the guided simplification, we select all general registers as the start of the backward slicing. Then, we can extract entire context switch instructions from T_c^s. We also place another set of anchor instructions outside of the VM macros to discover potential obfuscation schemes. It can help the analyst to select an appropriate heuristic to discover context switch instructions.

Dispatcher Structure To detect the type of dispatcher structures, we insert multiple identical instructions as `c`. Then, we can classify the dispatcher structure by examining whether the identical handler is reused in VM. For example, we can insert multiple `add eax, 0x10` instructions as `c` into P_c. If the VM schedules the same virtual handler `add` multiple times, it means that the VM uses a decode-dispatch loop structure.

C. Verification

We apply multiple approaches to verify the extracted knowledge. When the `k` is the mapping rules, we generate symbolic formulas of $simplify(T_I)$ and original `I`, respectively. If the generated formulas are semantically equivalent, we consider that the extracted mapping rule is correct. As for other `k` such as VM structure, we manually verify the correctness and also compare it with the results of existing research.

D. Reuse of Extracted Knowledge

The extracted knowledge can be reused in three scenarios.

Human analyst. The extracted knowledge can help the analyst quickly understand virtualized malware. Given the extracted mapping rules, the analyst can learn the relationship between native and virtualized instructions and efficiently recover the semantics of virtualized instructions.

Heuristic Algorithm. The analyst can get a systematic insight into the internal mechanism of commercial VM-based obfuscators. Guided by the extracted knowledge, the analyst can choose appropriate heuristics and design deobfuscation techniques.

Benchmarks. We construct a fine-grained benchmark based on our extracted mapping rules. The analyst can systematically evaluate and improve their deobfuscation techniques.

VI. Implementation

We implement a prototype of our CIA model. The whole toolchain is made of 274 lines of C++/C code, 318 lines of Makefile, and 9k lines of Python code. In the CIA sample construction stage, we resolve Intel XED [48] to generate different categories (i.e., CPU, FPU, SIMD, and other types) of valid instructions. Since the x86 and x64 architecture use the same mnemonic, we only consider the Intel x86 instructions. We combine each mnemonic with multiple prefixes (e.g., `lock`) and operands, including register, immediate value, and memory operands with different granularity (32-bit, 16-bit, and 8-bit). The generated instructions are embedded into the sample code templates which are written in C We build a logger based on Intel Pin DBI framework [46] to record the execution trace. Our trace analysis framework is built on top of the binary analysis platform *angr* [49]. We have redesigned the trace-based slicing and normalization methods to implement guided simplification techniques. Besides, we leverage *angr*'s symbolic execution engine and Z3 SMT solver [50] component to verify the correctness of the generated formula.

VII. Evaluation

In this section, we demonstrate the effectiveness of the chosen-instruction attack and the usability of the extracted knowledge. Our experiments seek to answer the following research questions (RQs).

- **RQ1:** How many usable anchor instructions can be found from real-world commercial VM-based obfuscators?
- **RQ2:** Which type of knowledge can the CIA attacker extract from real-world commercial VM-based obfuscators?
- **RQ3:** Can the extracted knowledge help the analyst to improve the existing deobfuscation techniques?

For answering RQ1, we traverse the instructions in the Intel x86 ISA to search for anchor instructions suitable for different obfuscators. As the response to RQ2, we perform CIA on the four most widely used commercial VM-based obfuscators to extract two kinds of essential knowledge. In RQ3, we construct a fine-grained benchmark based on the extracted knowledge and evaluate the state-of-the-art deobfuscation techniques.

Target of CIA. The selected commercial VM-based obfuscator includes: VMProtect [19], Code Virtualizer [20], Themida [18], and Obsidium [35]. Themida and Code Virtualizer are sharing the same VMs, which include six VMs (tiger, fish, shark, puma, dolphin, and eagle). These VMs can also be combined with three levels of obfuscation complexity, represented by colors (white, red, and black). We only evaluate the first three VMs and different complexity. One reason is that when we apply dolphin VM to generate the virtualized $\mathcal{O}(P_c)$, both Code Virtualizer and Themida would crash randomly. Wang et al. [51] also reported a similar problem. Besides, the traces generated by the shark, puma, and eagle VM are significantly larger than the traces of other VMs. For example, the trace of shark white T_I is 100x larger than the trace of other VMs (shown in Table III). The trace of puma white and eagle white VM is nearly 2x and 3x larger than the trace of shark white, respectively. Due to our limited computing resources, it takes nearly nine hours to finish CIA on our testbed for a trace of puma white VM. Therefore, we only evaluate the shark white VM among the last four VMs. But we have run and verified that our tool works properly on randomly selected samples generated by puma and eagle VMs. As for the configuration (e.g., packing) of each obfuscator, we have verified the effectiveness of CIA under different obfuscation configurations. Only the anti-debug option can affect CIA by impeding Intel Pin. It is a common limitation for any dynamic binary instrumentation framework. For simplicity, we only present the evaluation result when applying a single virtualization option to generate virtualized programs.

TABLE II: Number of anchor instructions in Intel x86 ISA.

	CPU	FPU	SIMD	Other	Total
Anchor	142	102	508	8	760
Total	875	193	508	8	1584

Our experiments are running on a testbed machine with Intel i7-6700 CPU, 32GB RAM, 1.8TB Hard Disk, running Windows 10.

A. Anchor Instruction

The anchor instruction is the key component of CIA to locate the virtualized knowledge leaking code. One single anchor instruction is enough to launch a chosen-instruction attack. Since the mapping rules of each commercial obfuscator are dissimilar, the usable anchor instructions are also different. We search the anchor from four obfuscators and use the subset ($anchor_{\texttt{VMProtect}} \cap anchor_{\texttt{CodeVirtualizer}} \cap anchor_{\texttt{Themida}} \cap anchor_{\texttt{Obsidium}}$) as the final list. We generate different categories of valid instructions `I` based on the syntax resolved from Intel XED [48] (discussed in Sec. VI). The generated `I` is inserted into the sample code templates (e.g., example shown in Fig. 5) to compile unprotected testbed programs. For each obfuscator, we use every single testbed program as input to generate three corresponding virtualized programs. Then, we record the runtime trace of virtualized programs and count the number of anchor instructions. In every trace that belongs to the generated `I`, if the `I` satisfies the property of anchor (see Sec. IV-A), we will consider it as a usable anchor.

The number of usable anchor instructions for each obfuscator is shown in Table III. We notice that VMProtect

TABLE III: The features of virtual structure and virtual ISA are extracted from four obfuscators based on CIA. In "Obfuscators" column, "CV" means the Code Virtualizer. "$T(\mathcal{O}(P_{\mathtt{I}}))$" represents the number of instructions in the trace of virtualized program $\mathcal{O}(P_{\mathtt{I}})$ (without anchor), and $T_{\mathtt{I}}$ represents the number of instructions that we use the anchor to locate the real virtualized snippet from $T(\mathcal{O}(P_{\mathtt{I}}))$. "simplify($T_{\mathtt{I}}$)" represents the number of instructions that extracted from $T_{\mathtt{I}}$ by our guided simplification.

Obfuscators	Anchor	Average Trace Size (ins)			Virtual ISA		VM Structure	
		$T(\mathcal{O}(P_{\mathtt{I}}))$	$T_{\mathtt{I}}$	simplify($T_{\mathtt{I}}$)	Extracted Mapping Rules	Transformation	Context Switch	Dispatcher Structure
VMProtect							obfuscated (semantic-based)	
v3.5	783 (ins)	5,758	2,116	28	240/241 (ins)	O2M, O2O		threaded
v2.13.8	794 (ins)	18,330	4,405	26	231/232 (ins)	O2M, O2O		decode-dispatch-loop
CV / Themida v3.0.7								
tiger white	1,144 (ins)	42,828	9,131	16	138/138 (ins)	O2O	obfuscated+decoy	threaded+fake-dispatch-loop
tiger red	1,147 (ins)	63,388	12,208	31	138/138 (ins)	O2O	obfuscated+decoy	threaded+fake-dispatch-loop
tiger black	1,144 (ins)	71,876	13,522	37	138/138 (ins)	O2O	obfuscated+decoy	threaded+fake-dispatch-loop
fish white	1,143 (ins)	45,161 [§]	45,309	43	138/138 (ins)	O2O (obfsucated)	obfuscated	threaded
shark white	1,146 (ins)	3,696,909	3,673,495	532	138/138 (ins)	O2O (obfsucated)	obfuscated+decoy	threaded+fake-dispatch-loop
CV / Themida v2.2.2								threaded+fake-dispatch-loop
tiger white	1,144 (ins)	99,663	17,622	17	138/138 (ins)	O2O	obfuscated+decoy	
tiger red	1,144 (ins)	151,532	22,326	40	138/138 (ins)	O2O	obfuscated+decoy	
tiger black	1,108 (ins)	172,663	23,421	46	138/138 (ins)	O2O	obfuscated+decoy	
fish white	1,143 (ins)	90,715	75,749	45	138/138 (ins)	O2O (obfsucated)	obfuscated	
shark white	1,146 (ins)	6,562,094	5,737,988	955	138/138 (ins)	O2O (obfsucated)	obfuscated+decoy	
Obsidium 1.6.7	1,175 (ins)	13,905	6,234	19	64/64 (ins)	O2M, O2O	no obfuscation	decode-dispatch-loop

[§] The fish VM does not adopt decoy context switch and fake dispatch loop. Since the obfuscator randomly insert junk instructions into $\mathcal{O}(P_{\mathtt{I}})$, the $T_{\mathtt{I}}$ can be greater than $T(\mathcal{O}(P_{\mathtt{I}}))$.

can virtualize more instructions than other obfuscators. There are fewer anchor instructions found in VMProtect_v3 than VMProtect_v2. The reason is that VMProtect_v3 can handle part of `lock` prefix instructions such as `add` and `or`. Other obfuscators have more anchor instructions because they cannot virtualize complex instructions such as `bts`. Besides, Obsidium cannot handle most instructions that use 8-bits and 16-bits operands. It is also a reason that we find much more anchor instructions in Obsidium. The reason why different colored `tiger` VMs have different anchors is that some samples are failed to be generated by the obfuscator. In total, there are 760 anchor instructions feasible for all obfuscators. The obtained anchor instructions can be categorized into four types (CPU, FPU, SIMD, and other special instructions), which are shown in Table II. We find that most CPU anchor instructions have complex semantics. It is hard (or impossible) for the obfuscator developers to write virtual handlers to emulate the original instruction. For example, the `cmpxchg` and decimal arithmetic instructions (e.g., `daa`) have complex behaviors. If the instruction (e.g., `cpuid` and `sysenter`) needs to interact with the operating system or hardware, it is impossible for VM to emulate the hardware behavior and construct the correct return value. As for the FPU instructions set, we discover that VMProtect only can replace the general registers used in instruction (e.g. `fild [esp]`) with virtual registers. If the operands of instruction only use non-general registers (e.g., `st(0)`), it can still become an anchor.

Since the anchor is used to trigger the obfuscator to generate *virtualization repelling* (i.e., context switch instructions) during the virtualized programs generation process, there is no need to execute the anchor instruction at runtime. We can use the `INS_Delete` function, provided by Intel Pin, to remove the anchor instruction before the CPU executes it. It will not violate the execution environment. Therefore, without the need of carefully constructing instructions to avoid polluting the original execution flow, any anchor instructions can be directly inserted into the program. As for the potential mitigation of anchor, we discussed in **`PM-1`** of Sec. VIII-B.

Answer to RQ1: We have discovered 760 anchor instructions from four state-of-the-art commercial VM-based obfuscators. There are 681 anchor instructions retrieving information from the system kernel or hardware. They cannot be virtualized by the process-level VM.

B. Knowledge Extraction

As described in Sec. V, we aim to extract two essential knowledge k (mapping rule and VM structure) from four obfuscators. The experiment results are summarized in Table III.

1) Mapping Rule: The CIA attacker has to construct various instructions as knowledge leaking code to extract kernel virtualized instructions and summarize mapping rules (discussed in Sec. V-B1). Considering that the mapping rules of some native instructions (e.g., control transfer instructions) are not reusable, we choose 241 instructions from the instruction dataset discussed above. It mainly contains arithmetic, shift, and logical operation (detailed in Appendix A).

The number of mapping rules extracted from different VMs is shown in Table III. We successfully extract 1,915 instruction mapping rules from four obfuscators in total. We have applied Z3 to verify whether each extracted virtualized instruction is semantically equivalent to the original corresponding instruction. The only instruction that failed to be extracted is `bswap`. This instruction reverses the byte order of the register operand. It requires our tool to track the data flow between multiple sources and destinations in bit-wise. We will support it in future work.

We also discover that obfuscators apply different strategies to virtualize the mnemonic and operands of native instructions. All four obfuscators can encrypt (or encode) the immediate operand during virtualizing native instructions, and the generated VM will decrypt it into a virtual register at runtime. Only VMProtect applies context-sensitive encryption to protect important values such as bytecode [1] and immediate operand. The context-sensitive key will be updated by

VM after every encryption and decryption. It increases the difficulty to track the data flow (discussed in Sec. II-C) and correctly recover the original operands. The different levels of obfuscation (i.e., white, black, and red) provided by Code Virtualizer and Themida only change the numbers of inserted junk instructions but will not influence the complexity of the mapping rules between original instructions and kernel virtualized instructions. In particular, the operation virtual handler of kernel virtualized instructions still uses the same mnemonic as `c` in the virtualized instructions. Unlike Code Virtualizer (and Themida), VMProtect and Obsidium can use a different mnemonic to emulate the operation of the original mnemonic. For example, the `inc edx` instruction is emulated by `ADD(edx, Decrypt(mem)⇒1)`. We also discover that the obfuscator tends to reuse the mapping rules. For example, the latest version of Code Virtualizer, Themida, and VMProtect reuse the mapping rules adopted in their older versions.

Transformation Strategy. For each input program P_c, we generate three $\mathcal{O}(P_c)$ under the same obfuscator options `s`. By comparing the results extracted from multiple programs, we notice that only VMProtect and Obsidium have adopted O2M transformation strategies. VMProtect uses multiple operation virtual handlers to emulate the operation of the original mnemonic (including the affected status flag). For example, the `xor edx, ecx` is transformed to `NOR(OR(edx,NOT(ecx)),OR(NOT(edx),ecx))` (detailed in Appendix Fig. 10), a Mixed Boolean-Arithmetic expression (MBA) [36]. We discover that 116 instructions can be transformed into a combination of multiple operation handlers (e.g., `NAND`, `NOR`). Besides, VMProtect has applied multiple but limited mapping rules to emulate the same native instruction. For example, the same `xor` instruction as above can also be transformed to `NAND(NAND(edx,ecx),AND(edx,ecx))`. Although Code Virtualizer and Themida only adopt O2O transformation, they use heavyweight obfuscation to protect virtualized instructions used in `fish` and `shark` VMs.

2) Virtual Machine Structure and obfuscation schemes: To learn the VM structure and related obfuscation schemes from obfuscators, we use the `nop` instruction as knowledge leaking code and extract the VM structure related instructions. The result is also shown in Table III.

Context Switch. Most VMs use `mov` or `push/pop` to load and restore the native context. But the VMs provided by Code Virtualizer and Themida also use `xchg`. We discover that they also insert multiple decoy context switch into virtualized snippets, even the virtualized instructions between the anchor instructions. The decoy context switch is in the same stack depth as the real context restoring instructions. It can deactivate the context pairing algorithm used in VMHunt. As for VMProtect, it uses semantic-based obfuscation to protect the generated context switch instructions. It also greatly hinders rule-based transformation techniques such as peephole optimization.

Dispatcher Structure. We find out that most obfuscators have replaced the decode-dispatch loop with the threaded code structure. Similar to the discovery of VMHunt, our experiments also show that Code Virtualizer and Themida have applied fake dispatch loop to mislead the existing techniques.

Furthermore, we discover that the sophisticated VM-based obfuscator also uses code virtualization to protect other integrated obfuscation schemes (e.g., the runtime unpacking process). For example, if the obfuscator user applies packing and code virtualization to protect the program at the same time, the virtualized program will contain multiple virtualized areas. But it may only contain a single virtualized area related to the semantics of the original program. If the deobfuscation technique only relies on the VM structure pattern to locate virtualized snippets, it cannot distinguish which virtualized area is related to the original program. Considering the binary packing technique is widely adopted by malware authors [13], [14], this will greatly damage the correctness of the existing research in the real world.

Answer to RQ2: The CIA attacker can learn mapping rules and VM structures from the state-of-the-art commercial VM-based obfuscators. We have extracted 1,915 verified mapping rules and features of VM structures from four obfuscators.

C. Reuse of Extracted Knowledge

To demonstrate the effectiveness of extracted knowledge, we reuse the mapping rules and VM structure in two scenarios. We construct the first fine-grained benchmark. The security analysts can systematically evaluate their deobfuscation techniques on our benchmark. Besides, we also generate analysis reports for each sample to help the analyst understand the VM mechanism and improve the deobfuscation technique.

1) Fine-grained Benchmark: As we discussed in Sec. II-B, without a fine-grained benchmark, the existing research cannot systematically evaluate the effectiveness of deobfuscation techniques. The coarse-grained evaluation (e.g., CFG similarity) may cause researchers to ignore defects such as over-simplify and over-taint. Our fine-grained benchmark is a complement to existing evaluation methods.

Datasets. The dataset of the fine-grained benchmark is similar to the samples from extracted mapping rules (detailed in Sec. VII-B1). It contains 5,745 virtualized programs $O(P_c)$ generated by four obfuscators and corresponding extracted virtualized instructions that have been verified. In each P_c program, it contains a standard input-output data flow which involves the instruction of `c`. It can be recognized by the data dependence analysis such as the dynamic taint analysis used in generic-deobfuscator [27]. Besides, we use the extracted mapping rules as the baseline information. It contains the original instructions, verified virtualized instructions, and symbolic formulas. The deobfuscation technique can compare their simplified results with the baseline information.

Metrics. We propose two metrics to evaluate the simplified trace $\mathcal{T}$ generated by performing the target deobfuscation techniques on virtualized samples. (1) *Recovery Rate.* This metric is used to evaluate the correctness and completeness of the simplified result. We first search verified virtualized instructions, provided by baseline information, from $\mathcal{T}$ to examine whether $\mathcal{T}$ contains the correct virtualized snippet. Then, we compare the symbolic formulas of $\mathcal{T}$ with those of the original instruction. It can discern whether $\mathcal{T}$ is semantically equivalent to the original instruction. If $\mathcal{T}$

TABLE IV: The result of benchmarking two state-of-the-art deobfuscation techniques. In "Deobfuscator" column,"*without anchor*" means that the input of deobfuscation techniques is their own trace and "*with anchor*" means that the input is virtualized snippets located based on anchor instructions.

Deobfuscator	VMProtect 3.5					Code Virtualizer 2.2.2 (tiger white)				
	Recovery Rate			Average Size of Simplified Trace	Average Redundancy	Recovery Rate			Average Size of Simplified Trace	Average Redundancy
	Full	Partial	None			Full	Partial	None		
without anchor										
VMHunt [25]	0	0	0	-	-	14	-	124	3329 ins	98.77%
generic-deobfuscator [27]	52	13	172	480 ins	95.45%	13	-	125	445 ins	95.18%
Syntia [1]	-	-	-	-	-	-	-	-	-	-
with anchor										
VMHunt	237	0	0	389 ins	88.98%	138	-	0	181 ins	92.20%
generic-deobfuscator	52	13	172	147 ins	61.70%	13	-	125	37 ins	55.87%
Syntia	4	17	51	-	-	0	12	60	-	-

only contains parts of our verified virtualized instructions and the formulas are unequal, we consider $\mathcal{T}$ only have partial semantics of original instructions. (2) *Redundancy Rate.* This metric helps us measure the redundancy of the simplified trace. The redundancy rate of simplified trace $\mathcal{T}$ is computed as

$$redundant(\mathcal{T}) = 1 - \frac{I_{kernel}}{I_{total}}$$

where I_{kernel} is the number of instructions that can be found both in $\mathcal{T}$ and our baseline, and I_{total} is the total number of instructions in $\mathcal{T}$. If the simplified trace has a high redundancy rate, it means that the trace still contains enormous redundant instructions which are irrelevant to the semantics of `I`.

Target for Comparison. To demonstrate the importance of our benchmark, we select three open-source state-of-the-art deobfuscation techniques (*VMHunt* [25], *generic-deobfuscator* [27], and *Syntia* [1]) as the target. Generic-deobfuscator is the latest research that uses a generic approach to simplify malware protected by unknown obfuscation. VMHunt is the state-of-the-art deobfuscation technique used to defeat virtualized malware. Syntia uses stochastic program synthesis to recover approximate semantics of virtual handlers. We only show the evaluation results of deobfuscation techniques performed on the samples virtualized by VMProtect and Code Virtualizer's `tiger white` VM. There are two reasons: 1) the above three research all adopt these two VMs for evaluation; 2) `tiger white`, the weakest VM, is selected as the borderline case. Besides, we only evaluate Syntia on 72 samples randomly selected from our benchmark dataset. Because Syntia requires analysts to manually extract handlers from tedious execution traces as the input, and identify the correct result from multiple outputs generated by the program synthesis. The best we can do is to manually select various handlers and examine corresponding synthesized results. This cumbersome process leads us to evaluate Syntia only on a limited dataset.

Since Syntia requires manually extracted handler as the input, we discuss it in section Sec. VII-C2. We first perform VMHunt and generic-deobfuscator on our virtualized samples to generate the simplified trace. Then, we compare the result with our baseline information. The evaluation results are shown in Table IV (*without anchor* category). We discover that neither generic-deobfuscator nor VMHunt can successfully recover all instructions in the benchmark. The generic-deobfuscator is limited by the unsound rule-based transformation which will mistakenly simplify obfuscated instructions. The mistakenly simplified instructions cause the dynamic taint analysis used by generic-deobfuscator to overlook the correct data flow (detailed in *case study #1*). The experiment results show that VMHunt only can recover parts of instructions generated by `tiger white` VM. One reason is that the decoy context switch will mislead VMHunt to locate the fake virtualized snippets. VMHunt's context pairing algorithm cannot distinguish the real virtualized snippet that mapped from `c` in $P_{\mathtt{c}}$. Besides, the peephole optimizer used in VMHunt cannot remove semantic-based obfuscation among the context switch instructions generated by VMProtect. Thus, VMHunt is unable to locate context switch instructions and virtualized snippets. As for the redundancy rate, neither generic-deobfuscator nor VMHunt can fully simplify the instructions. The redundant instructions greatly thwart the analyst to understand the $\mathcal{T}$. It also prohibitively increase the complexity of symbolic formulas generated by VMHunt. We find out most redundant instructions are introduced by the semantic-based obfuscation such as the key register adopted by VMProtect. We discuss the detail in *case study #2*.

2) Assistance to Deobfuscation Techniques: To reveal the ability of extracted knowledge in assisting deobfuscation techniques, we use the anchor to improve *VMHunt* and generic-deobfuscator. Specifically, we use the anchor to help the deobfuscation technique correctly locate the virtualized snippets from the trace. The evaluation result is also shown in Table IV (*with anchor* category). When we provide correct virtualized snippets to VMHunt, it can work normally. Likewise, Syntia can work properly when it uses handlers that are located based on our anchor instructions. But Syntia cannot simplify the handlers that mapped from complex instructions such as `div` and `ror`. Syntia also cannot recover the semantics of instructions virtualized by O2M transformations (detailed in *case study #3*). The anchor can also help generic-deobfuscator to reduce the redundancy of simplified results, but cannot improve the recovery rate. Although the dynamic taint analysis used in generic-deobfuscator can correctly find virtualized snippets related to input and output, it is misled by erroneous transformation.

Furthermore, our extracted knowledge can help security analysts to understand complex code virtualization mechanisms. We generate an HTML report for each analyzed native instruction. Each report contains the information of knowledge leaking code, the original trace of virtualized instructions, extracted virtualized instructions, and lifted symbolic expressions. Each

instruction in the original trace is colored according to different types. An example of the generated report is discussed in Appendix A.

Answer to RQ3: We construct the first fine-grained benchmark based on our extracted mapping rules and evaluate the effectiveness of three state-of-the-art deobfuscation techniques [1], [25], [27]. We also provide an HTML report for each analyzed instruction. These can assist security analysts in understanding the virtualized instructions, selecting an appropriate heuristic, and designing deobfuscation techniques.

D. Case Study

Case Study #1: Over Simplification. Fig. 7 presents a virtual handler $add(v1, v2)$ in Fig. 2 mistakenly simplified by generic-deobfuscator. The value of the original register operand `ax` is represented by virtual register `ecx`, which connects the data flow between standard input and output. Before taint analysis, generic-deobfuscator applies multiple transformation rules to simplify virtualized instructions. Some inappropriate transformation rules convert the instruction `add cx, dx` to `mov cx, 0x2` according to the concrete value of registers. It terminates the data flow from the register `cx` and `dx` to the memory. Because generic-deobfuscator uses the standard input and output as the taint source and sink, respectively. The taint analysis cannot identify data flow and related instructions.

Case Study #2: Data flow Misleading. As shown in Fig. 2, the `Decrypt` virtual handler decrypts the masked immediate operand `0x1` into the virtual context during VM runtime execution. The secret key stored in register `ebx` is also used to decrypt the bytecode and other important values. This context-sensitive key value is updated after every encryption and decryption. Hence the data dependence analysis will be misled to collect enormous junk instructions that use the key register.

Case Study #3: Handler Combination. An example of the handler combination (i.e., O2M transformation) generated by VMProtect is shown in Fig. 8. The handlers are used to represent the semantic of kernel virtualized instructions. The semantic of `xor` operation is emulated by a combination of `NAND` handlers. Syntia only can synthesize the semantic of a single handler such as the `NAND` handler. But the `NAND` handler is also used to emulate other native instructions such as `adc`. It is hard to recover the complete semantic of the original instruction.

VIII. Discussion

A. Heuristic Algorithms used in Deobfuscation

The heuristic algorithm is widely used in existing deobfuscation techniques. But the source and effectiveness of heuristics have never been systematically evaluated. The existing heuristic selection process is built on the analyst's experience, which is learned from painfully examining the large-scale virtualized programs. In particular, the internal mechanism of the commercial VM-based obfuscator is complicated and undisclosed. Meanwhile, the obfuscator developer can easily eliminate the patterns used by heuristic algorithms. The outdated heuristic makes the deobfuscation techniques also become obsolete quickly. To the best of our knowledge, our research is the first attempt to solve this dilemma.

B. Potential Mitigations for CIA

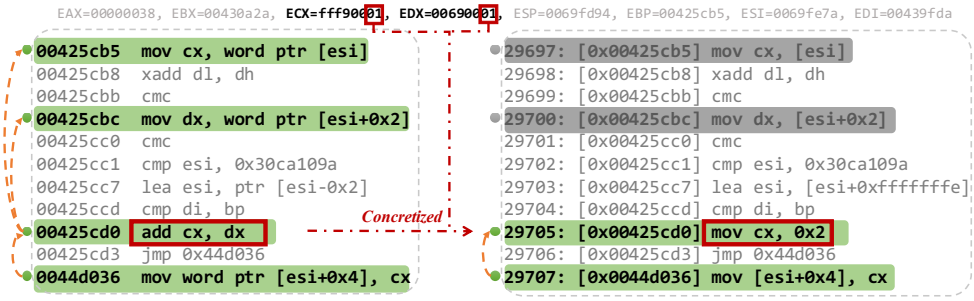
The potential mitigations can be categorized into two main types: against the component/concept of CIA (**`PM-1`**, **`PM-2`**, **`PM-3`**) and the implementation of CIA (**`PM-4`**, **`PM-5`**, **`PM-6`**).

`PM-1`: ***Deactivate every anchor.*** The obfuscator developers may attempt to destroy or reduce the usability of the anchor. Intuitively, the developers may prefer to emulate every anchor instruction. But they can only emulate part of anchor instructions that have complex semantics (e.g., `cmpxchg`), and pay a great cost on writing lots of virtual handlers. The process-level VM cannot emulate the instructions that interact with hardware. For example, processor-related (e.g., `cpuid`), privilege-related (e.g., `syscall`), `FPU`, and `SIMD` instructions. Therefore, the obfuscator developers may attempt to alter the representation of anchor instruction. They can replace the general registers used in the anchor instruction with virtual registers. For example, as mentioned in Sec. VII-A, we have found that VMProtect adopts this technique to handle part of `FPU` instructions (e.g. `fild`). But this method cannot deactivate the anchor that only uses non-general registers (e.g., `st(0)`). Note that Barak et al. [52] have proven perfect obfuscation is impossible. The CIA attackers can always find anchor instruction (e.g., `syscall`) that cannot be virtualized by obfuscators.

`PM-2`: ***Diversify mapping rules.*** The obfuscator developers can diversify mapping rules to prevent the CIA attackers from extracting complete rules. For example, developers can write more O2M transformation rules for different native instructions. But it will also increase development costs and impair the flexibility of mapping rules. On the other hand, the developer can also insert junk instructions into the mapping rules. We only find that Code Virtualizer and Themida have applied this technique. But the inserted garbage instruction can be easily removed through peephole optimization. If the obfuscator integrates other heavy-weight data obfuscation such as diversified MBA expressions [36], [53], it may impede the CIA attackers from extracting mapping rules. But the performance overhead of virtualized programs will be excessively increased.

`PM-3`: ***Use custom code virtualization.*** Malware authors can customize code virtualization to prevent analysts from reusing the extracted knowledge to understand virtualized malware. It is one of our limitations. But it is difficult to design customized code virtualization as powerful as commercial VM-based obfuscators. Malware authors must pay huge development costs, which may severely harm their profits. Our statistical result also shows that most malware authors prefer the commercial VM-based obfuscator (detailed in Sec. I). To our best knowledge, FinFisher [21] is the only known malware that applies the custom code virtualization. However, Kaspersky recently discovered that FinFisher switches to VMProtect [54].

`PM-4`: ***Impede anchor locating and guided simplification.*** To increase the difficulty of locating anchor instructions and performing guided simplification, the obfuscator developers can apply pattern recognition and heavy-weight obfuscation. But the virtualized programs may become unstable and suffer heavy performance overhead. Correspondingly, the CIA



(a) virtual handler of `add(v1, v2)` (b) Intermediate results (before taint analysis)

Fig. 7: An example of instructions mistakenly simplified by *generic-deobfuscator*. The concretized `mov cx, 0x2` causes taint analysis cannot track data flow from register `cx` to `dx`.

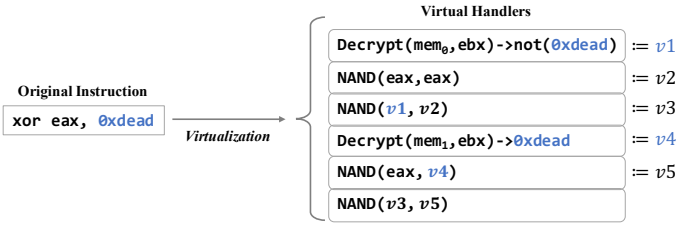


Fig. 8: An example of mapping between `xor eax, 0xdead` and corresponding virtual handlers.

attackers can diversify the combination forms of anchor instructions and knowledge leaking code. Meanwhile, they can also combine other simplification methods to improve guided simplification techniques.

PM-5: ***Impede DBI-based dynamic analysis.*** Since the trace recording process of our framework is currently based on Intel Pin, the obfuscator developer can apply anti-debugging techniques to defeat the DBI-based dynamic analysis. But analysts can adopt the hardware-assisted tracing technique [55], [56] to evade the detection of anti-debugging techniques. Furthermore, the obfuscator developer may attempt to increase the number of (junk) virtualized instructions to enlarge the execution trace size. But it will greatly increase the runtime overhead of virtualized programs. Besides, the CIA attackers can increase their computing resources and optimize the algorithm.

PM-6: ***Run as a cloud service and use access control.*** If the commercial obfuscator runs as a cloud service (e.g., Android APK packing service [57], [58]), traditional MATE attackers [42] cannot directly reverse-engineer obfuscators to learn complete knowledge. Since the CIA attackers only need input and output programs, they can still effectively extract knowledge by interacting with the obfuscator cloud services. Intuitively, the obfuscator deployed in the cloud can record and restrict the interactive behavior of authorized users. The obfuscator can also scan the pattern of anchor instructions to discover a potential CIA threat, and stop providing services to potential attackers. However, it is difficult for the obfuscator to learn the complete semantics of input programs. The CIA attackers can also disguise the knowledge leaking program as a benign program. If the obfuscator assumes that any anchor instruction is a component of CIA, it will suffer high false positives.

C. Is chosen-instruction attack short-lived?

As discussed in Sec. VIII-B, if the manufacturers of obfuscators only apply **PM-4**, **PM-5**, and **PM-6** to impede the implementation of our CIA toolchain, the CIA attackers can always find a way to improve analysis methods for defeating these mitigations. Therefore, the manufacturers have to protect obfuscators against essential components (i.e., anchor instruction, knowledge extraction, and obfuscator's accessibility) of the CIA model. Otherwise, the CIA model will keep threatening the commercial VM-based obfuscators. However, as discussed in **PM-1**, manufacturers cannot virtualize every anchor such as `syscall`. **PM-2** is a promising direction for the manufacturers, but the CIA attackers can also improve the analysis method and traverse the diversified mapping rules. **PM-3** is an expensive choice for software (or malware) authors.

IX. Limitations and Future Work

The prerequisite for performing CIA is that the attackers can interact with obfuscators. Inaccessible obfuscators are out of the scope of the CIA model. For example, malware authors can design a custom code virtualization scheme (e.g., the custom VM in FinFisher [21]). But the customized code virtualization scheme is weak and rarely observed in the wild (detailed in Sec. I). Besides, the custom VM also inevitably inherits the same design philosophy of code virtualization. For example, FinFisher's VM uses the classic decode-dispatch based interpretation. The knowledge we extracted from commercial obfuscators can help analysts to understand the customized virtualized malware. To defeat our trace-based analysis technique, obfuscator developers can also widely adopt the anti-debugging technique or maximize the size of virtualized instructions. It is a common limitation for any DBI-based dynamic analyses. Moreover, the CIA model is designed to assist analysts in extracting knowledge from commercial VM-based obfuscators, rather than directly simplifying virtualized malware. We leave it to future work. Another limitation of the CIA model is that the selection of anchor instructions depends on the architecture. For example, the reduced instruction set, such as RISC, may limit the number of anchor instructions. Considering Android malware also uses code virtualization to evade anti-virus detection [59], we plan to migrate the CIA to the Android platform in the future.

X. Conclusion

In the competition with code virtualization, the existing deobfuscation techniques inevitably rely on heuristic algorithms. Note that the heuristics are selected by the analysts through painfully examining enormous virtualized programs. They are suffering from inaccuracies and incompleteness. This paper proposes a new approach, called *chosen-instruction attack*, to extract reusable knowledge automatically by interacting with the obfuscator. The experiment results indicate that four state-of-the-art commercial obfuscators are under the threat of CIA. Based on the extracted mapping rules, we construct a fine-grained benchmark to evaluate the effectiveness of the existing works. The evaluation results show that the extracted knowledge is a complement to the deobfuscation techniques.

Acknowledgments

We sincerely thank NDSS 2022 anonymous reviewers for their insightful comments and Dr. Marcus Peinado for helping us improve the paper throughout the shepherding process. This work was supported by National Natural Science Foundation of China (62172238, 61972215, and 61972073); Natural Science Foundation of Tianjin (20JCZDJC00640); Tianjin Research Innovation Project for Postgraduate Students (2019YJSS092); and National Key R&D Program of China (2018YFA0704703). Jiang Ming was supported by the National Science Foundation (NSF) under grant CNS-1850434 and CNS-2128703. This project was also supported by the National Research Foundation, Singapore and National University of Singapore through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office under the Trustworthy Software Systems - Core Technologies Grant (TSSCTG) award no.NSOE-TSS2019-02.

References

[1] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz, "Syntia: Synthesizing the Semantics of Obfuscated Code," in *Proceedings of the 26th Usenix Security Symposium (Usenix Security)*. USENIX Association, 2017, pp. 643–659.

[2] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*. ACM, 2016, pp. 189–200.

[3] Jim Smith and Ravi Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, 1st ed. Morgan Kaufmann, 2005.

[4] Kevin A. Roundy and Barton P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, pp. 1–32, 2013.

[5] Christian Collberg, Clark Thomborson, and Douglas Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM Press, 1998, pp. 184–196.

[6] VMProtect Software, "VMProtect and DENUVO GmbH," (accessed on 2019-04-12). [Online]. Available: www.vmpsoft.com/20170606/vmprotect-and-denuvo-gmbh/

[7] Skanda Hazarika, "How to use Samloader to download updates for your Samsung Galaxy device," (accessed on 2021-05-21). [Online]. Available: https://www.xda-developers.com/samloader-download-updates-samsung-galaxy

[8] ALEXANDER SEVTSOV, "Tyupkin ATM Malware: Take The Money Now Or Never!" (accessed on 2021-05-21). [Online]. Available: https://www.lastline.com/labsblog/tyupkin-atm-malware/

[9] Malwarebytes Threat Intelligence Team, "Fake "Corona Antivirus" distributes BlackNET remote administration tool," (accessed on 2021-05-21). [Online]. Available: https://blog.malwarebytes.com/threat-analysis/2020/03/fake-corona-antivirus-distributes-blacknet-remote-administration-tool/

[10] Filip Kafka, "ESET'S GUIDE TO DEOBFUSCATING AND DEVIRTUALIZING FINFISHER," ESET, Tech. Rep., 2018.

[11] FireEye, "APT38 Un-usal Suspects," 2018, (accessed on 2021-06-22). [Online]. Available: www.fireeye.com/content/dam/fireeye-www/current-threats/pdfs/pf/apt/rpt-apt38-2018.pdf

[12] Filip Kafka (ESET), "New traces of Hacking Team in the wild," (accessed on 2021-05-21). [Online]. Available: https://www.welivesecurity.com/2018/03/09/new-traces-hacking-team-wild/

[13] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas, "SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015, pp. 659–673.

[14] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel, "When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features," in *Proceedings 2020 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2020.

[15] CrowdStrike, "Free Automated Malware Analysis Service - powered by Falcon Sandbox." [Online]. Available: https://www.hybrid-analysis.com/

[16] VirusTotal, "VirusTotal." [Online]. Available: https://www.virustotal.com/

[17] ESET, "ESET Virusradar," (accessed on 2021-06-11). [Online]. Available: https://www.virusradar.com/

[18] Oreans Technologies, "Themida Overview," (accessed on 2019-04-12). [Online]. Available: www.oreans.com/themida.php

[19] VMProtect Software, "VMProtect Software Protection," (accessed on 2019-04-12). [Online]. Available: https://www.vmpsoft.com

[20] Oreans Technologies, "Code Virtualizer Overview," (accessed on 2019-04-12). [Online]. Available: www.oreans.com/codevirtualizer.php

[21] FinFisher, "FinFisher," (accessed on 2021-01-08). [Online]. Available: https://finfisher.com/FinFisher/index.html

[22] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee, "Automatic Reverse Engineering of Malware Emulators," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2009, pp. 94–109.

[23] Rolf Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX conference on Offensive technologies (WOOT)*. USENIX Association, 2009, pp. 1–7.

[24] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller, "VMAttack: Deobfuscating Virtualization-Based Packed Binaries," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ACM, 2017, pp. 1–10.

[25] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu, "VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018, pp. 442–458.

[26] Kevin Coogan, Gen Lu, and Saumya Debray, "Deobfuscation of virtualization-obfuscated software," in *Proceedings of the 18th ACM conference on Computer and communications security (CCS)*. ACM Press, 2011, pp. 275–284.

[27] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray, "A Generic Approach to Automatic Deobfuscation of Executable Code," in *Proceedings of the 36th IEEE Symposium on Security and Privacy IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015, pp. 674–691.

[28] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet, "Symbolic Deobfuscation: From Virtualized Code Back to the Original," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference (DIMVA)*, vol. 10885 LNCS, 2018, pp. 372–392.

[29] Jasvir Nagra and Christian Collberg, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[30] Babak Yadegari and Saumya Debray, "Symbolic Execution of Obfuscated Code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2015, pp. 732–744.

[31] Robin David, Luigi Coniglio, and Mariano Ceccato, "QSynth - A Program Synthesis based approach for Binary Code Deobfuscation," in *Proceedings 2020 Workshop on Binary Analysis Research (BAR)*, no. February. Internet Society, 2020.

[32] James R. Bell, "Threaded code," *Communications of the ACM*, vol. 16, no. 6, pp. 370–372, 1973.

[33] Paul Klint, "Interpretation Techniques," *Software: Practice and Experience*, vol. 11, no. 9, pp. 963–973, 1981.

[34] Jonathan Katz and Yehuda Lindell, "Introduction to Modern Cryptography," in *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2007, ch. 3.5, pp. 81–85.

[35] Obsidium Software, "About Obsidium — Obsidium Software Protection System," (accessed on 2021-05-14). [Online]. Available: https://www.obsidium.de/

[36] Yongxin Zhou, Alec Main, Y.X. Gu, and Harold Johnson, "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms," in *Proceedings of the 8th International Workshop on Information Security Applications*, 2007, pp. 61–75.

[37] Anthony Desnos, "Dynamic, Metamorphic (and opensource) Virtual Machines," in *Hack.lu 2010*, 2010.

[38] Donabelle Baysa, Richard M. Low, and Mark Stamp, "Structural entropy and metamorphic malware," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 4, pp. 179–192, 2013.

[39] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron, "The case for virtual register machines," in *Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators (IVME)*. ACM Press, 2003, pp. 41–49.

[40] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee, "Impeding Malware Analysis Using Conditional Code Obfuscation," in *Proceedings of the 15th Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2008, pp. 321–333.

[41] Ian Piumarta and Fabio Riccardi, "Optimizing direct threaded code by selective inlining," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation (PLDI)*, vol. 33, no. 5, 1998, pp. 291–300.

[42] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan, "Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions," *Journal of Network and Computer Applications*, vol. 48, pp. 44–57, 2015.

[43] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang, "Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization," in *Proceedings of the 19th International Conference on Information and Communications Security (ICICS)*, 2017, pp. 313–324.

[44] Sudeep Ghosh, Jason Hiser, and Jack W. Davidson, "Replacement attacks against VM-protected applications," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE)*, vol. 47, no. 7. ACM Press, 2012, pp. 203–214.

[45] Jason Raber, "Virtual Deobfuscator: Removing virtualization obfuscations from malware," in *Black Hat USA*, 2013.

[46] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, vol. 40, no. 6. ACM Press, 2005, pp. 190–200.

[47] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu, "BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proceedings of the 26th Usenix Security Symposium (Usenix Security)*. USENIX Association, 2017, pp. 253–270.

[48] Intel, "Intel XED," (accessed on 2021-01-04). [Online]. Available: https://intelxed.github.io/

[49] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna, "SOK: (State of) the Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016, pp. 138–157.

[50] MICROSOFT RESEARCH, "The Z3 Theorem Prover," (accessed on 2021-01-08). [Online]. Available: https://github.com/Z3Prover/z3

[51] Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu, "Translingual Obfuscation," in *Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 128–144.

[52] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai Ucla, Salil Vadhan, and Ke Yang, "On the (Im)possibility of obfuscating programs," *Journal of the ACM*, vol. 59, no. 2, pp. 1–48, 2012.

[53] Yongxin Zhou and Alec Main, "Diversity via Code Transformations: A Solution for NGNA Renewable Security," *The National Cable and Telecommunications Association Show*, 2006.

[54] Kaspersky Lab Global Research & Analysis Team, "FinSpy: unseen findings," (accessed on 2021-09-29). [Online]. Available: https://securelist.com/finspy-unseen-findings/104322/

[55] Marcus Botacin, Paulo Lício De Geus, and André Grégio, "Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging," *ACM Transactions on Privacy and Security*, vol. 21, no. 1, pp. 1–30, 2018.

[56] Binlin Cheng, Jiang Ming, Erika A. Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean Yves Marion, "Obfuscation-resilient executable payload extraction from packed malware," in *Proceedings of the 30th USENIX Security Symposium (Usenix Security)*, 2021, pp. 3451–3468.

[57] JEB Decompiler, "Reversing DexGuard, Part 3 – Code Virtualization," (accessed on 2021-01-04). [Online]. Available: https://www.pnfsoftware.com/blog/reversing-dexguard-virtualization/

[58] GuardSquare, "DexGuard: Android App Security," (accessed on 2021-01-04). [Online]. Available: https://www.guardsquare.com/en/products/dexguard

[59] Lei Xue, Yuxiao Yan, Luyi Yan, Muhui Jiang, Xiapu Luo, Dinghao Wu, and Yajin Zhou, "Parema: an unpacking framework for demystifying VM-based Android packers," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, vol. 13. ACM, 2021, pp. 152–164.

APPENDIX

Fig. 9 presents part of a colored trace in our generated report. The instructions colored with yellow are the kernel instructions. The blue-colored instructions is used to decode the virtual bytecode and jump to the next handler. Other instructions colored with grey are the redundant instructions. With the help of our report, the analyst can easily find out the pattern of the `NOR` handler.

The mnemonic we choose to generate native instructions, which is used to extract mapping rules, includes: `adc, add, and, bswap, btc, btr, bts, cbw, cwd, cdq, cwde, dec, div, idiv, imul, inc, movsx, movzx, mul, neg, not, or, rcl, rcr, rdtsc, rol, ror, sahf, sal, sar, sbb, seta, setae, setb, setbe, sete, setg, setge, setl, setle, setne, setno, setnp, setns, seto, setp, sets, shl, shld, shr, shrd, sub, xadd, xor.`

```
1338 0x458eac mov ecx, dword ptr [ebp]
1339 0x458eb0 rol al, cl
1341 0x458eb4 bsr dx, sp
1342 0x458eb8 mov eax, dword ptr [ebp + 4]
1343 0x458ebb and dx, 0x7fe3
1344 0x458ec0 rcl dx, cl
1345 0x458ec3 not ecx
1346 0x458ec5 not eax
1347 0x458ec7 or ecx, eax
1348 0x458ec9 cmovp dx, bx
1349 0x458ecd mov dword ptr [ebp + 4], ecx
1350 0x458ed0 bswap dx
1351 0x458ed3 movsx edx, dx
1352 0x458ed6 pushfd
1353 0x458ed7 pop dword ptr [ebp]
1354 0x458edb lea edi, [edi - 4]
1355 0x458ee1 mov edx, dword ptr [edi]
1358 0x42e277 xor edx, ebx
1359 0x42e279 dec edx
1360 0x42e27a xor edx, 0x53c1455d
1362 0x42ff6e rol edx, 1
1366 0x455ba0 sub edx, 0xf51294
1369 0x455ba8 xor ebx, edx
1372 0x455bad add esi, edx
1374 0x4614ef jmp esi
```

Fig. 9: An example of colored trace.

```
10: 0x45afa9, mov dword ptr [0x69fe64], edx
16: 0x44962c, mov dword ptr [0x69fe58], ecx
114: 0x4274c6, mov ecx, dword ptr [0x69fe58]
132: 0x486e5d, mov dword ptr [0x69fdd0], ecx
250: 0x41c3ae, mov ecx, dword ptr [0x69fe64]
273: 0x4530ca, mov dword ptr [0x69fd9c], ecx
623: 0x47e005, mov edx, dword ptr [0x69fdd0]
630: 0x47e01d, mov dword ptr [0x69fe7c], edx
682: 0x454375, mov edx, dword ptr [0x69fdd0]
684: 0x45437e, mov dword ptr [0x69fe78], edx
715: 0x468770, mov ecx, dword ptr [0x69fe78]
716: 0x468772, mov eax, dword ptr [0x69fe7c]
719: 0x46877c, not ecx
720: 0x46877e, not eax
724: 0x468785, or ecx, eax
728: 0x468791, mov dword ptr [0x69fe7c], ecx
813: 0x420b17, mov edx, dword ptr [0x69fd9c]
818: 0x420b27, mov dword ptr [0x69fe78], edx
851: 0x47cc76, mov ecx, dword ptr [0x69fe78]
854: 0x47cc80, mov eax, dword ptr [0x69fe7c]
856: 0x47cc86, not ecx
859: 0x47cc8f, not eax
861: 0x43f6d1, or ecx, eax
862: 0x43f6d3, mov dword ptr [0x69fe7c], ecx
948: 0x470be6, mov edx, dword ptr [0x69fdd0]
952: 0x470bf5, mov dword ptr [0x69fe78], edx
1007: 0x46ad2a, mov edx, dword ptr [0x69fd9c]
1012: 0x46ad3b, mov dword ptr [0x69fe74], edx
1062: 0x488ebf, mov edx, dword ptr [0x69fd9c]
1069: 0x488ed8, mov dword ptr [0x69fe70], edx
1098: 0x482dac, mov ecx, dword ptr [0x69fe70]
1101: 0x482db4, mov eax, dword ptr [0x69fe74]
1103: 0x482dba, not ecx
1104: 0x482dbc, not eax
1107: 0x482dc2, or ecx, eax
1108: 0x482dc4, mov dword ptr [0x69fe74], ecx
1182: 0x4929c7, mov ecx, dword ptr [0x69fe74]
1183: 0x4929c9, mov eax, dword ptr [0x69fe78]
1186: 0x47b629, not ecx
1187: 0x47b62b, not eax
1189: 0x48144f, or ecx, eax
1193: 0x481459, mov dword ptr [0x69fe78], ecx
1276: 0x46de79, mov ecx, dword ptr [0x69fe78]
1278: 0x46de7c, mov eax, dword ptr [0x69fe7c]
1282: 0x46de87, not ecx
1284: 0x46de8f, not eax
1287: 0x46f28f, or ecx, eax
1291: 0x44601d, mov dword ptr [0x69fe7c], ecx
1377: 0x460b26, mov ecx, dword ptr [0x69fe7c]
1400: 0x461907, mov dword ptr [0x69fdc0], ecx
1719: 0x4547bf, mov edx, dword ptr [0x69fdc0]
1724: 0x4547cf, mov dword ptr [0x69fe68], edx
1932: 0x41613e, mov edx, dword ptr [0x69fe68]
```

Fig. 10: An example of extracted virtualized instructions that generated by VMProtect v3.5. The original instruction is the `xor edx, ecx`. It is transformed to `NOR(OR(edx,NOT(ecx)),OR(NOT(edx),ecx))` represented by virtualized instructions.