# Forced Execution for Malware Protected by Commercial Virtualization Obfuscators

Yifei Zhan, Yukun Cui, Shuofeng Hao, Dongnan He, Wei You*, Bin Liang, Jianjun Huang
School of Information, Renmin University of China, Beijing, China
{zyfkkk,cuiyukun666,haoshuofeng4869,hedongnan,youwei,liangb,hjj}@ruc.edu.cn

## Abstract

Forced execution is an effective technique for exposing hidden payloads in malware by systematically enforcing the outcomes of conditional transfer instructions. However, modern malware increasingly employs commercial virtualization obfuscators (e.g., VMProect, Themida, Enigma, Obsidium) to protect their code body. In particular, native instructions are transformed to customized virtual bytecode instructions, which are decoded and interpreted at runtime. As a result, native control transfers become obscure from forced execution tools, rendering conventional approaches largely ineffective. In this paper, we propose a novel forced execution technique specifically designed to address virtualization-based obfuscation. The technique does not require comprehensive deobfucation. Instead, it only identifies the instructions responsible for the virtualized conditional transfer instructions, leveraging a critical observation that the virtualization needs to follow specific data/control-flow to preserve the original semantics of conditional transfer. Once identified, these instructions are manipulated to simulate the effect of enforcing branch outcomes in the original plaintext code. Evaluations on the benchmarks with ground truth show that our technique achieves 100% recall and nearly 100% precision in identifying virtualized conditional transfers, and generalizes well across a variety of popular commercial virtualization obfuscators. Evaluations on real-world malware samples show that the number of behaviors exposed by our technique is 10.97 times of plain execution, and 1.89 times of native forced execution.

## CCS Concepts

• **Security and privacy → Software reverse engineering**; **Software security engineering**.

## Keywords

malware analysis, virtualization-based obfuscation, forced execution

**ACM Reference Format:**
Yifei Zhan, Yukun Cui, Shuofeng Hao, Dongnan He, Wei You*, Bin Liang, Jianjun Huang . 2026. Forced Execution for Malware Protected by Commercial Virtualization Obfuscators. In *Proceedings of the 33rd ACM Conference on Computer and Communications Security (CCS '26).* ACM, New York, NY, USA, 24 pages. https://doi.org/XXXXXXX.XXXXXXX

* Corresponding Author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*CCS '26, The Hague, The Netherlands*
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2025/09
https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Malware poses a significant threat to the security of computing systems. Statistics indicate that malware is responsible for 80% of cyber-attacks worldwide [21]. According to Acronis [20], the ALPHV ransomware gang, wanted by the FBI, has breached over 1,000 entities, demanded more than $500 million, and received over $300 million in ransom payments. Many malware samples do not release their payload until specific environmental conditions are satisfied (e.g., specific times, dates, and workloads). Consequently, their behaviors often resemble that of benign software, making malware detection and analysis particularly challenging.

Multi-path exploration [43] is a technique used to explore program paths and penetrate program behaviors. The most common methods for muli-path execution include symbolic execution [44], fuzzing [28], and forced execution [29, 45, 59]. *Symbolic execution* runs a program with symbolic inputs, generates constraints on predicates, and solves them to explore program paths. However, it faces difficulties when scaling to complex, real-world binaries. *Fuzzing* generates random inputs to explore program paths, but its effectiveness relies heavily on understanding the input format and the execution environment. This makes it impractical for zero-day malware, whose input and environment are unknown.

*Forced execution* explores program paths by forcibly setting the branch outcomes of a small set of predicates, such as those guarding payload execution. It provides a practical solution for exposing hidden malware behaviors without requiring specific inputs or environment configuration. However, the growing adoption of virtualization-based obfuscation in malware [27] has significantly diminished the effectiveness of forced execution. In such obfuscation schemes, original instructions are transformed into customized bytecode instructions, which are then decoded and interpreted at runtime. As a result, original conditional transfer instructions are no longer directly visible to conventional forced execution techniques.

Malware developers tend to prefer commercial virtualization obfuscators over customized solutions. Statistics from a previous study [40] shows that 99% of virtualized malware samples are protected by commercial virtualization obfuscators. This preference arises from the fact that developing customized virtualization obfuscator is both time-consuming and error-prone. Crafting an effective customized solution requires significant effort to accurately implement various virtual components. Otherwise, the resulting obfuscation may be considerably less robust than that of commercial alternatives. While commercial virtualization obfuscators benefit malware developers, their inherent complexity poses a substantial challenge for analysts to understand the behavior of protected malware samples.

As detailed in Appendix H, although virtualized malware still constitutes only a fraction of all malware samples, it poses a critical and increasingly relevant challenge and has attracted substantial interest in the research community. Researchers have proposed various techniques to deobfuscate malware samples protected by commercial virtualization obfuscators. The mainstream studies of deobfuscation heavily rely on prior knowledge of the virtualization mechanism employed by the target obfuscator, such as bytecode features [5, 10], interpretation methods [32], context switch patterns [55] and mapping rules [50]. Although these techniques have shown great potential, their reliance on strict assumptions about the obfuscation limits their applicability. Furthermore, commercial virtualization obfuscators continually devise new methods (e.g., inserting fake decode-dispatch loops, introducing decoy context switches, and complicating mapping rules) to circumvent these detection efforts.

Recently, researchers proposed the chosen-instruction attack [40], which constructs a special program to extract reusable knowledge for analyzing other programs obfuscated by the same obfuscator. However, the reuse of knowledge across different obfuscated programs comes with limitations due to the diversity of obfuscation and the learnability of such knowledge. Specifically, knowledge of conditional transfers is neither learnable nor reusable due to its complex semantics. As discussed above, deobfuscating the obfuscated bytecode and then applying forced execution to the resulting plaintext code is not an effective solution.

While deobfuscation can be beneficial, enabling forced execution for behavioral analysis of malware protected by popular commercial virtualization obfuscators does not require full deobfuscation of the entire virtualized code body. Instead, it is sufficient to identify the places corresponding to conditional transfers in the original code body and manipulate these places accordingly to achieve the effect of setting branch outcomes. Moreover, we observe that in order to preserve the original semantics of conditional transfers, the virtualization needs to follow certain data/control-flow patterns, which can be leveraged for effective identification.

Specifically, virtual machines maintain a *virtual program counter* (*vpc*), which functions similarly to the native program counter and indicates the current virtual instruction being executed; as well as a *virtual flag register* (*vflag*), analogous to the native flag register, which holds various state bits. Some of these bits are used to determine the targets of virtualized conditional transfers. The virtualization of a conditional transfer involves information flows from the bytecode area to *vpc*, from *vflag* to *vpc*, and from *vpc* to the native program counter. These information flows remain invariant across different virtualization techniques.

Precisely identifying *vpc* and *vflag* itself is challenging, as it requires reverse-engineering the underlying virtualization mechanism. Existing deobfuscation-based approaches [34, 50] that rely on precise identification of *vpc* make strong assumptions about its format and storage. However, these assumptions no longer hold in the latest versions of obfuscators, rendering the existing approaches ineffective. Rather than striving for precise identification of *vpc* and *vflag*, this paper proposes an alternative approach that maintains a candidate list for both of them. Candidates that exhibit the aforementioned information flow invariant are flagged as likely true positives, and can subsequently be used to identify and manipulate virtualized conditional transfers. Based on the approach, we develop a prototype system VMForce, which can enhance the state-of-the-art forced execution systems, achieving more effective extraction of virtualized malware payloads.

Our contributions are summarized as follows.

- We propose a practical forced execution approach for analyzing malware samples protected by commercial virtualization obfuscators. To the best of our knowledge, it is the first forced execution work capable of circumventing commercial virtualization obfuscation.
- We design a novel algorithm to identify and manipulate conditional control transfer in the virtualized code without the need for deobfuscation. The algorithm makes only weak assumptions and is general to popular commercial virtualization obfuscators (e.g., VMProtect [18], Themida [13], Enigma [4], Obsidium [8]).
- We implement a prototype system and evaluate its effectiveness. The evaluation results show that our approach can accurately identify obfuscated conditional transfers and effectively expose hidden malware behaviors obfuscated by virtualization.

## 2 Motivation

We use an example to illustrate the technical challenges, and motivate the idea of VMForce. Listing 1 depicts a simplified code snippet from a virtualized variant of a real-world command and control (C&C) malware sample [52], with some modifications for illustration purposes. While the description is presented at the source code level for clarity, the actual analysis is performed at the binary level. Upon its initialization, the malware checks whether the running instance originates from a designated destination path (Line 12). If not, the malware replicates itself to the destination path (Line 13). Moreover, if the automatic startup mechanism is enabled (Line 14), the malware registers itself to run automatically during Windows startup (Line 15). During execution, the malware receives messages from a remote server (Line 21) and performs the corresponding actions. For example, when receiving a message that contains the "tiger" indicator (Line 23), the malware will execute the command line specified in the message (Lines 24-25); when receiving a message that contains the "monkey" indicator (Line 27), the malware will download a file from the URL specified in the message (Lines 28-29). As the command and control logic (Lines 21-32) is significantly important for malware, it is obfuscated via virtualization to thwart analysis.

Merely executing the malware sample in a sandbox environment cannot fully expose all of its malicious payloads, since they are safeguarded by highly specific conditions. In our example, if the automatic startup mechanism is disabled, the suspicious registry operation will not be exposed; if the C&C server is not reachable, the malicious command execution and file download operations will remain concealed. Manually configuring the trigger conditions is time-consuming and not practical for zero-day malware samples.

Forced execution provides a systematic solution to explore different program behaviors without any environment setup. The

```
1 #define KEY_NAME "SOFTWARE\\...\\CurrentVersion\\Run"
2 #define VAL_NAME "eset_update"
3 #define DEST_PATH "C:\\...\\Roaming\\eset_update.exe"
4 #define REMOTE_SERVER "general-second.org-help.com"
5 #define LEN 0x200
6
7 char path[LEN], msg[LEN], op[LEN], cmd[LEN], url[LEN];
8 HKEY key;
9
10 int main() {
11   path = get_current_path();
12   if (strcmp(path, DEST_PATH)) {
13     CopyFile(path, DEST_PATH, 0);
14     if (!RegOpenKey(..., KEY_NAME, ..., &key)) {
15       RegSetValue(key, VAL_NAME, ..., DEST_PATH, ...);
16       RegCloseKey(phkResult);
17     }
18   }
19   while (1) {
20     /* Virtualizatoin-Based Obfuscation - Begin */
21     if (receive_msg(REMOTE_SERVER, msg)) {
22       get_operation_from_msg(msg, op);
23       if (!strcmp(op, "tiger")) {
24         get_command_from_msg(msg, cmd);
25         WinExec(cmd, 0);
26       }
27       else if (!strcmp(op, "monkey")) {
28         get_url_from_msg(msg, url);
29         URLDownloadToFile(..., url, ...);
30       }
31       ...
32     }
33     /* Virtualizatoin-Based Obfuscation - End */
34   }
35 }
```

**Listing 1: Motivating example.**

existing forced execution tools [29, 45, 59] designed for native code can readily reveal the registry operation by forcibly setting the predicate at Line 14 to take the true branch. However, these tools encounter challenges when attempting to forcibly set the predicates at Lines 21, 23, and 27 (for revealing the command execution and file download operations), as they are obfuscated by virtualization. As virtualization is increasing used by malware developers, there is an urgent need for forced execution techniques that are effective on virtualization-obfuscated code.

## 2.1 Background and Term Definitions

Virtualization-based obfuscation works by transforming native instructions to bytecode instructions in a customized instruction set and then interpreting the bytecode using native instructions. In particular, a native instruction is described by one or multiple bytecode instructions. Each bytecode instruction is interpreted by a number of handler functions implemented in native instructions. We call the instructions to be virtualized the "*original instructions*", and the native instructions that emulate bytecode the "*emulating instructions*".

Figure 1 illustrates the workflow of commercial virtualization obfuscators (e.g., VMProtect, Themida, Enigma, and Obsidium), which consists of two phases: transformation and interpretation. In the transformation phase, given a program with user-specified functions to be obfuscated, the obfuscator transforms each selected native instruction into a sequence of byte code instructions, which can be interpreted by a set of handler functions. The resulting obfuscated program is self-contained, embedding both the virtualized bytecode and the corresponding virtual handlers. Each virtual handler comprises two essential code components: the execution code, which performs the intended operations; and the jump code, which transfers control to subsequent handlers. A virtual handler may be used in emulating multiple original instructions.

During interpretation, when an obfuscated function is invoked, its bytecode is interpreted by a virtual machine in a manner specific to the obfuscator. A classic pattern involves a *decode-dispatch loop* [30], in which the interpreter iteratively decodes the bytecode and directs the execution flow to the appropriate handler by looking up a jump table. More sophisticated *threaded interpretation* [35] has been proposed to improve efficiency and hinder reverse engineering. This method removes the central decode-dispatch loop and inlines the decoding and dispatching logic to virtual handlers. By using *pre-decoding* [42], the decoding logic is performed only once per unique instruction, and the decoded results are reused for subsequent executions of that instruction. *Direct threading interpretation* [24] combines threaded interpretation and pre-decoding, removing jump table lookups by storing the addresses of virtual handlers in the pre-decoded results.

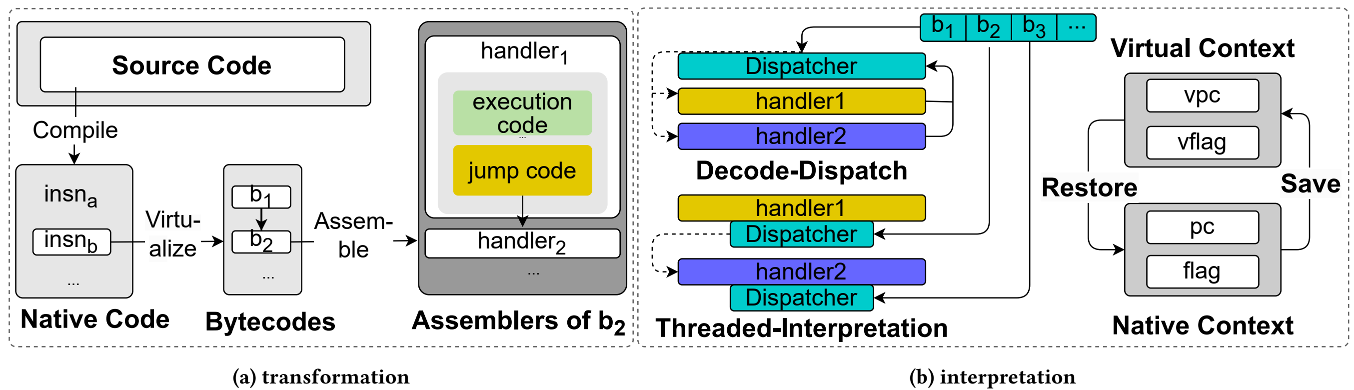
Regardless of the interpretation method, the virtual machine generally maintains a virtual context for executing the obfuscated bytecode. In particular, the virtual context contains a virtual program counter ($vpc$) that points to the next bytecode instruction to be executed and a virtual flag register ($vflag$) that represents the state of various conditions after arithmetic and logical operations. Upon invoking an obfuscated function, the native registers are saved into the virtual context. When the obfuscated function returns, the native registers are restored from the virtual context. Appendix A illustrates how conditional transfers are virtualized.

## 2.2 Limitations of Existing Techniques

Forced execution tools designated for native code are unable to forcibly execute the virtualized bytecode. One might consider offline modification or on-the-fly instrumentation of the virtualized bytecode to influence the original program's logic, similar to the forced execution approaches [41, 54] used in the analysis of Android malware. However, these approaches are ineffective for analyzing virtualization-obfuscated binary malware. On the one hand, unlike the Android Dalvik/ART virtual machine, which has a publicly available instruction set, the virtualizers used by malware samples employ custom instruction sets that vary from case to case. On the other hand, advanced virtualizers encrypt the original bytecode, and any incorrect modification or instrumentation of the bytecode will lead to the immediate termination of the program.

Considering the extensive research [32, 50, 55] dedicated to the deobfuscation of virtualization-obfuscated program, a natural approach is to deobfuscate the obfuscated bytecode and then apply forced execution to the resulted plaintext code. Unfortunately, such an approach faces practical issues stemming from deobfuscation.

The mainstream studies of deobfuscation heavily rely on ad-hoc knowledge of the virtualization mechanism employed by the target




(a) transformation (b) interpretation

**Figure 1: Workflow of commercial virtualization obfuscators.**

obfuscator (e.g., bytecode features, interpretation methods, context switch patterns, and mapping rules). FKVMP [5] and Oreans UnVirtualizer [10], working as OllyDbg [9] plugins, utilize pattern matching to identify the emulating and bytecode instructions within the samples protected by VMProtect and Themida, respectively. VMAttack [32] recognizes a conspicuous interpreter loop structure that includes decoding and dispatching behaviors. VMHunt [55] pairs the context saving instructions with the restore instructions to identify the virtualized snippet between them. Rotalumé [50] detects the mapping between virtual bytecode and handlers by reverse engineering the syntax and semantics of bytecode. These deobfuscation approaches are effective when the virtualization mechanism meets certain assumptions (e.g., the interpreter uses a decode-dispatch loop), yet they exhibit limited generalization in scenarios where these assumptions are unmet [48]. Furthermore, some commercial obfuscators have adopted sophisticated techniques to deliberately circumvent deobfuscation techniques [47]. For example, latest versions of VMProtect and Themida provide advanced options to insert multiple fake decode-dispatch loops and decoy context switches into the virtualized program for camouflaging the real dispatcher and misleading the detection of virtualized snippet boundaries. They also combine multiple transformation strategies to complicate the mapping rules.

Another line of work proposes generic deobfuscation techniques that do not make strong assumptions about the underlying virtualization mechanism. For example, Generic Deobfuscator [57] uses semantics-preserving program transformations to simplify away obfuscation code; Syntia [25] uses program synthesis in combination with Monte Carlo tree search to simplify execution traces for semantics learning. These simplification-based deobfuscation approaches can reduce the syntactic complexity of code and improve readability. However, they are not well-suited for recovering individual virtualized instructions. Specifically, these approaches focus on simplifying the code of each virtual machine handler. For simple arithmetic instructions, the virtualized bytecode corresponds to a single handler, making it easy to recover the original instructions. In contrast, virtualized conditional transfers involve more complex logic and must be emulated by multiple handlers. Separately understanding the semantics of each handler offers limited help in recovering the original conditional transfers due to the lack of a global perspective.

Recently, researchers proposed the chosen-instruction attack (CIA) [40], which automatically extracts reusable knowledge by constructing a special program designed to leak internal information from the target obfuscator. The extracted knowledge can facilitate the analysis of other programs obfuscated by the same obfuscator. In the scenario of deobfuscating mapping rules, the reusable knowledge is the simplified emulating instructions, which are the minimum instructions required to emulate the semantics of a given original instruction. However, the reuse of knowledge across different obfuscated programs comes with limitations due to the diversity of obfuscation and the learnability of such knowledge. On one hand, different obfuscation options yield different mapping rules for the same original program. It is non-trivial to identify the obfuscation options used by an obfuscated program, a critical aspect for determining which knowledge-leaking program (using the same obfuscation options) should be used to learn the mapping rules. On the other hand, certain types of original instructions may lack learnable knowledge. For instance, given that the interpreter maintains its own flag register within the virtual context, there is no direct linkage between the original conditional jump instruction and its corresponding emulating instructions. Consequently, it is infeasible to extract kernel emulating instructions for conditional jump instructions.

## 2.3 Observations

Our technique is inspired by three observations, each corroborated by our evaluation (§4.2). First, *full-fledged deobfuscation is unnecessary for the purpose of forced execution.* Forced execution is dedicated to disclosing hidden malicious behaviors protected by highly specific conditions. The core task is to identify the original conditional transfers from the emulating instructions and reverse the corresponding predicate outcomes. Emulating instructions unrelated to the original conditional jump instructions are not the focus of forced execution, and therefore do not need to be deobfuscated.

Second, *the emulating instructions for conditional transfers preserve semantics regardless of the obfuscation variants.* The semantic invariant is evident in the interaction between the virtual context and the native context, as well as between the bytecode area and

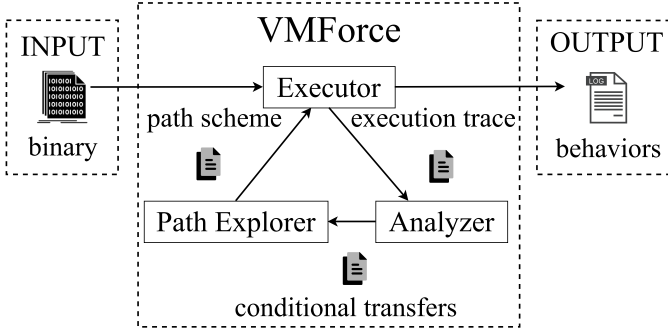



**Figure 2: Architecture of VMForce.**

the native code. To emulate the conditional transfers, the emulating instructions should retrieve the addresses of the true and/or false branches from the bytecode area, determine which branch to take (in the virtual context) based on certain bits' values of the virtual flag register, and finally transfer control (in the native context) to the native handlers responsible for the target branch.

Third, *the semantic invariant of virtualized conditional transfer is observable from the execution trace of the obfuscated program.* Virtualized conditional transfers exhibit three properties with respect to the information flow, which comply with the underlying virtualization mechanism that emulates CPU behavior: ① as a virtualized instruction, there should be information flow from $vpc$ to the native program counter; ② as a virtualized transfer, there should be information flow from the bytecode area to $vpc$; ③ since the virtualized transfer is conditional, there should be information flow from $vflag$ to $vpc$. Other native or virtualized instructions are less likely to exhibit all three properties of information flow simultaneously. Although advanced obfuscators may complicate the format or storage of $vpc$ and $vflag$, and obscure the information flow, the fundamental information flow pattern persists to preserve semantics. Consequently, the semantic invariant of virtualized conditional transfers can be discerned from the execution trace by recognizing this distinct information flow pattern.

## 3 Design

Figure 2 presents the architecture of VMForce, which consists of three components: the path explorer, the analyzer and the executor. Given a target binary, the path explorer systematically selects a subsequence of the currently identified virtualized conditional transfers (termed a *path scheme*), whose predicate outcomes are to be reversed. Note that VMForce enforces only a very small number of virtualized conditional transfers, while the others are evaluated as usual. For each path scheme, the executor runs the target binary to generate an execution trace, forcibly reversing the predicate outcomes of the virtualized conditional transfers specified in the path scheme; the analyzer is then examines the execution trace to identify virtualized conditional transfers. This section elaborates on the identification of virtualized conditional transfers (§3.2) and the reversal of predicate outcomes (§3.3). The implementation details can be found in Appendix D.

| | |
|---|---|
| ⟨Program⟩ | $prog ::= insn$ |
| ⟨Insn⟩ | $insn ::= insn_1; insn_2$ |
| | $\mid\ r \overset{m}{\Leftarrow} [r_{base} + r_{idx} * c_{scale} + c_{disp}]$ |
| | $\mid\ [r_{base} + r_{idx} * c_{scale} + c_{disp}] \overset{m}{\Leftarrow} v \ \mid\ r \Leftarrow v$ |
| | $\mid$ `jmp` $r$ $\mid$ `jmp` $[m]$ $\mid \ldots$ |
| ⟨Register⟩ | $r ::= \{r_{null},\ r_1,\ r_2,\ \ldots\}$ |
| ⟨MemAddr⟩ | $m ::= \{0,1,2,\ldots\}$ |
| ⟨Value⟩ | $v ::= \{0,1,2,\ldots\}$ |

**(a) language**

```
         AbstractVar ::= Register | MemAddr
        ProgramState ::= AbstractVar → Value
      ExecutionTrace ::= [Insn^N_PS]
     VirtualVarValue ::= Insn^N_PS × AbstractVar × Value
                Type ::= AbstractVar | Value
            VpcCands ::= {VirtualVarValue}
          VflagCands ::= {VirtualVarValue}
          VpcUpdVals ::= AbstractVar → [Value]
 ConditionalTransfer ::= VirtualVarValue × VirtualVarValue
```

**(b) abstract types**

```
           PushItem ::= [Type] × Type → [Type]
        GetLastItem ::= [GenericType] → GenericType
         GetValueOf ::= Insn^N_PS × AbstractVar → Value
       IsBitMasking ::= Insn^N_PS → Boolean
    GetBitMaskedOpds ::= Insn^N_PS → {AbstractVar}
    GetAgedVpcCands ::= VpcCands × Insn^N_PS → VpcCands
  GetAgedVflagCands ::= VflagCands × Insn^N_PS → VflagCands
    GetTaintSources ::= AbstractVar → {AbstractVar}
     AddTaintSource ::= AbstractVar → Void
  RemoveTaintSource ::= AbstractVar → Void
     PropagateTaint ::= Insn^N_PS → Void
```

**(c) utility functions**

**Figure 3: Definitions of the language, abstract types and utility functions for understanding semantics.**

### 3.1 Preliminary

**Definitions.** We introduce a simplified low-level language to model binary programs and their execution. The syntax of the language is presented in Figure 3a. A program consists of a sequence of instructions. The instruction $r \overset{m}{\Leftarrow} [r_{base} + r_{idx} * c_{scale} + c_{disp}]$ standardizes memory read operations using different addressing modes [6]. In particular, $r_{base}$ and $r_{idx}$ serve as the base register and the index register, respectively; $c_{scale}$ and $c_{disp}$ the scale and the displacement, respectively; $m$ is the effective address of the memory operation. By disabling $r_{base}$ and/or $r_{idx}$ (setting to $r_{null}$), we can alter the addressing mode. Similarly, the instruction $[r_{base} + r_{idx} * c_{scale} + c_{disp}] \overset{m}{\Leftarrow} v$ standardizes memory write operations using different addressing modes. The instruction $r \Leftarrow v$ moves a value $v$ to a register $r$. The instruction `jmp` $r$ abstracts indirect jump instructions where the target address resides in the register, while `jmp` $[m]$ abstracts those where the target address is in the memory.

Figure 3b introduces a set of abstract types to help define the semantics. An abstract variable (denoted as `AbstractVar`) represents either a register or a memory address. A program state (denoted as `ProgramState`) maps an abstract variable to its runtime value. An execution trace (denoted as `ExecutionTrace`) consists of a sequence of instruction instances. An instruction instance (denoted as $\mathtt{Insn}^{\mathbb{N}}_{\mathbb{PS}}$) is an instruction annotated with a number $\mathbb{N}$ that represents

the index of the instruction instance in the execution trace, and a program state $\mathbb{PS}$ that records the operand values of the instruction instance. A virtual variable value (denoted as `VirtualVarValue`) is in the form of "$\langle \underline{insn}, var, val \rangle$", which indicates that the value of an abstract variable *var* is *val* after the execution of the instruction instance $\underline{insn}$. We define a generic type (denoted as `Type`) to generalize the types of abstract variables and values. We use `VpcCands` and `VflagCands` to denote sets of virtual variable values, which respectively serve as the *vpc* candidates and *vflag* candidates, respectively; and use `VpcUpdVals` to record the updated values for each *vpc* candidate for tracking purposes. A conditional transfer (denoted as `ConditionalTransfer`) is represented by both the virtual variable value of *vpc* and the virtual variable value of its corresponding *vflag*.

Figure 3c introduces some utility functions. `PushItem` appends an item to the end of a sequence. `GetLastItem` retrieves the last item from a sequence. `GetValueOf` returns the value of a specified abstract variable after the execution of a given instruction instance. `IsBitMasking` checks whether an instruction instance performs bit-masking operation on its operands, and `GetBitMaskedOpds` retrieves the bit-masked operands. `GetAgedVpcCands` and `GetAgedVflagCan` get the *vpc* candidates and *vflag* candidates whose corresponding emulating instruction instance is older (exceeding a predefined threshold) than the specified instruction instance. We also define utility functions for taint analysis (explained later).

**Semantic invariant.** As mentioned in §2.1, emulating instructions emulate the semantics of original instructions. According to a previous study [56], a conditional transfer is typically emulated in one of two ways: ⓐ through an indirect transfer whose target depends on the predicate of the original conditional transfer; or ⓑ through a different conditional transfer whose predicate depends on, but differs from, that of the original conditional transfer. Our analysis of popular commercial virtualization obfuscators confirms this conclusion.

Scenario ⓐ: conditional transfer → indirect transfer, as illustrated in Figure 4a. the addresses of the true and the false branches are stored on the stack (Lines 2 and 4). The value of *vflag* undergoes bit manipulations, and the result is used as index to retrieve the target address from the stack (Lines 5-7). The opcode is then fetched from the byetecode based on the target address (Line 8), and the control is finally transferred to the corresponding handler by a native jump instruction (Lines 9-10). VMProtect corresponds to this scenario.

Scenario ⓑ: conditional transfer → conditional transfer′, as illustrated in Figure 4b. Specifically, the value of *vflag* is compared with a constant to determine which branch is taken (Lines 1-2). Different branches lead to different target addresses in the bytecode (Lines 4-5 and Lines 8-9). In particular, the functions $f1$ and $f2$ return target addresses of the false branch and the true branch, respectively. Themida, Enigma, Obsidium correspond to this scenario.

In both scenarios, there exist (direct or transitive) data flows (denoted as $\overset{d}{\rightsquigarrow}$) or control flows (denoted as $\overset{c}{\rightsquigarrow}$) from the bytecode area to *vpc*, from *vflag* to *vpc*, and from *vpc* to the native program counter. Various obfuscation options may complicate the data and control flows, yet the semantics remain unchanged. We consider this to be a semantic invariant.

```
 1 taddr := fetch_bytecode(vpc + α) // bytecode ⇝d taddr
 2 stack[4] := taddr // taddr ⇝d stack
 3 faddr := fetch_bytecode(vpc + β) // bytecode ⇝d faddr
 4 stack[0] := faddr // faddr ⇝d stack
 5 vflag := retrieve_vflag_from_virtual_context()
 6 index := (vflag & 0x40) >> 4 // vflag ⇝d index
 7 vpc := stack[index] // stack ⇝d vpc, index ⇝d vpc
 8 opcode := fetch_bytecode(vpc) // vpc ⇝d opcode
 9 handler := get_handler(opcode) // opcode ⇝d handler
10 jmp handler // handler ⇝d native program counter
```

**(a) conditional transfer → indirect transfer**

```
 1 vflag := retrieve_vflag_from_virtual_context()
 2 cmp vflag, γ
 3 jcc .L1
 4 faddr := fetch_bytecode(vpc + α) // bytecode ⇝d faddr
 5 vpc := f1(vpc,faddr) // faddr ⇝d vpc, vflag ⇝c vpc
 6 jmp .L3
 7 .L1:
 8 taddr := fetch_bytecode(vpc + β) // bytecode ⇝d taddr
 9 vpc := f2(vpc,taddr) // taddr ⇝d vpc, vflag ⇝c vpc
10 .L3:
11 opcode := fetch_bytecode(vpc) // vpc ⇝d opcode
12 handler := get_handler(opcode) // opcode ⇝d handler
13 jmp handler // handler ⇝d native program counter
```

**(b) conditional transfer → conditional transfer′**

**Figure 4: Emulation of conditional transfers.**

One might argue that a force executor designed for native code could easily manipulate obfuscated predicates at the emulation level by exploring native indirect jump table entries (Lines 6-7 of Figure 4a) or by flipping the native conditional jump (Line 3 of Figure 4b). However, this is challenging for two reasons. First, distinguishing the critical instructions responsible for virtualized conditional transfers is nontrivial. Second, a single handler function is responsible for multiple virtualized instructions. Without accurately identifying the invocation instance of the critical instructions responsible for a given virtualized conditional transfer, effective forced execution of virtualized code becomes infeasible.

## 3.2 Identify Virtualized Conditional Transfers

Virtualized conditional transfers in the execution trace are a mixture of those stemming from the logic of the original program and those needed by the interpreter itself. Our analyzer aims to identify those corresponding to the original conditional transfers. A straightforward approach is to specify the registers and memory areas that store *vpc*, *vflag* and bytecode, and then track the data and control flows between these areas. However, such strong and fixed obfuscation-specific assumptions limit the generality of the identification approach.

To minimize assumptions regarding obfuscations, we propose a solution to recognize candidates of critical virtual components (*vpc*, *vflag* and the bytecode) based on the following rules that are

**Algorithm 1:** identify virtualized conditional transfers.

```
1  Input    : trace:         ExecutionTrace
   Output   : cond_xfers:    {ConditionalTransfer}
   Variable : vpc_cands^l:   VpcCands
              upd_vals^l:    VpcUpdVals
              vpc_cands^g:   VpcCands
              vflag_cands^g: VflagCands

   Procedure: HandleCase5()
2  if insn : jmp r or insn : jmp[m] then // Case 5
3      npc ← r(insn : jmp r) | m(insn : jmp[m])
4      handler_end ← false
5      for vpc^l in vpc_cands^l do
6          src_npc ← GetTaintSrcs(npc)
7          sat_rule_c ← (vpc^l.var ∈ src_npc) and (upd_vals^l[vpc^l.var] ≠ ∅)
8          if sat_rule_c then
9              handler_end ← true
10             for vpc^g in vpc_cands^g do
11                 src_vpc^l ← GetTaintSrcs(vpc^l.var)
12                 sat_rule_a ← (vpc^g.var ∈ src_vpc^l)
13                 if sat_rule_a then
14                     for vflag^g in vflag_cands^g do
15                         sat_rule_b ← (vflag^g.var ∈ src_vpc^l)
16                         if sat_rule_b then
17                             last_upd_val ← GetLastItem(upd_vals^l[vpc^l.var])
18                             vpc ← ⟨vpc^l.insn, vpc^l.var, last_upd_val⟩
19                             vflag ← ⟨vflag^g.insn, vflag^g.var, vflag^g.val⟩
20                             new_cond_xfer ← ⟨vpc, vflag⟩
21                             cond_xfers ← cond_xfers ∪ new_cond_xfer
22                             for tmp_vpc^g in vpc_cands^g do
23                                 RemoveTaintSrc(tmp_vpc^g.var)
24                             vpc_cands^g ← ∅; vflag_cands^g ← ∅
25                             break
26                     break
27             break
28     if handler_end then
29         vpc_cands^l ← ∅; upd_vals^l ← ()

   Procedure: Main()
30 for insn in trace do // iterate over the execution trace
31     aged_vpc_cands ← GetAgedCandsOfVPC(vpc_cands^g, insn)
32     aged_vflag_cands ← GetAgedCandsOfVFLAG(vflag_cands^g, insn)
33     vpc_cands^g ← vpc_cands^g \ aged_vpc_cands
34     vflag_cands^g ← vflag_cands^g \ aged_vflag_cands
35     for ⟨*, var, *⟩ in aged_vpc_cands do
36         RemoveTaintSrc(var)
37     if insn : r ⇐(m) [r_base + r_idx * c_scale + c_disp] then // Case 1
38         vpc_cand_1 ← ⟨insn, r_base, GetValueOf(insn, r_base)⟩
39         vpc_cand_2 ← ⟨insn, r_idx, GetValueOf(insn, r_idx)⟩
40         vpc_cand_3 ← ⟨insn, m, GetValueOf(insn, m)⟩
41         vpc_cands^l ← vpc_cands^l ⊎ {vpc_cand_1, vpc_cand_2, vpc_cand_3}
42         vpc_cands^g ← vpc_cands^g ⊎ {vpc_cand_1, vpc_cand_2, vpc_cand_3}
43         AddTaintSrc(r_base); AddTaintSrc(r_idx); AddTaintSrc(m)
44     else if insn : [r_base + r_idx * c_scale + c_disp] ⇐(m) v then // Case 2
45         if ⟨*, r_base, *⟩ ∈ vpc_cands^l then
46             upd_vals^l[r_base] ← PushItem(upd_vals^l[r_base], v)
47         if ⟨*, r_idx, *⟩ ∈ vpc_cands^l then
48             upd_vals^l[r_idx] ← PushItem(upd_vals^l[r_idx], v)
49         if ⟨*, m, *⟩ ∈ vpc_cands^l then
50             upd_vals^l[m] ← PushItem(upd_vals^l[m], v)
51     if insn : r ⇐ v then // Case 3
52         if ⟨*, r, *⟩ ∈ vpc_cands^l then
53             upd_vals^l[r] ← PushItem(upd_vals^l[r], v)
54     if IsBitMasking(insn) then // Case 4
55         bitmasked_operands ← GetBitMaskedOpds(insn)
56         for operand ∈ bitmasked_operands do
57             vflag_cand ← ⟨insn, operand, GetValueOf(insn, operand)⟩
58             vflag_cands^g ← vflag_cands^g ⊎ {vflag_cand}
59             AddTaintSrc(operand)
60     PropagateTaint(insn)
61     HandleCase5()
62 return cond_xfers
```

generalizable across widely used commercial virtualization obfuscators: ⓐ a $vpc$ candidate is used as a memory index for retrieving a value from a potential bytecode area, and the retrieved value subsequently influences the further updated value of the $vpc$ candidate; ⓑ a $vflag$ candidate masks certain bits of a value, and the masked value subsequently influences the updated value of a $vpc$ candidate; ⓒ the updated value of a $vpc$ candidate impacts the native program counter. These candidates serve as the foundation for identifying potential virtualized conditional transfers. Note that no single rule is sufficient on its own to ensure reliable identification. Instead, these rules must be applied in a specific sequence: Rule ⓒ selects potential virtualized instructions, Rule ⓐ narrows them down to potential virtualized transfers, and Rule ⓑ filters out potential virtualized conditional transfers. Appendix B presents an evaluation of the contributions of individual rules.

Algorithm 1 describes how our analyzer identifies virtualized conditional transfers. Given an execution trace that records each executed instruction instance along with the values of its register and memory operands, the algorithm analyzes each instruction instance to collect $vpc$ candidates and $vflag$ candidates, and identify those that satisfy the aforementioned rules. Notably, the algorithm examines every instruction instance appearing in the execution trace, without needing to explicitly distinguish between virtualization-related instructions and original program instructions.

A conditional transfer is emulated by multiple virtual handlers; hence, the analysis should span across virtual handlers. We locally maintain $vpc$ candidates and the values updated for each $vpc$ candidate within individual virtual handlers; and globally maintain $vpc$ candidates and $vflag$ candidates across multiple virtual handlers. Intuitively, the local $vpc$ candidates and their updated values are used to recognize the boundaries of individual virtual handlers (the satisfaction of Rule ⓒ), and the global $vpc$ and $vflag$ candidates are used to recognize the semantic invariant across virtual handlers (the satisfaction of Rule ⓐ and Rule ⓑ). The locally maintained records will reset at the end of each virtual handler (Lines 28-29), while the globally maintained records will reset whenever a virtualized conditional transfer is identified (Line 24). We should clarify that VMForce does not depend on accurately identifying individual virtual handlers. Instead, it detects the potential end of a virtual handler using a heuristic rule. Specifically, if an instruction demonstrates that the updated value of a $vpc$ candidate affects the native program counter, it is considered an indication of handler termination.

The analyzer iterates over the execution trace, and takes different actions depending on the type of each instruction instance. Case 1 (Lines 37-43): for a memory read instruction, the registers involved in memory addressing and the resulting effective address are added to the local and global *vpc* candidate sets. Case 2 (Lines 44-50): for a memory write instruction, if the registers involved in memory addressing and the resulting effective address are potential *vpc* candidates, the value that was written is added to the updated value list. Case 3 (Lines 51-53): for a register update instruction, if the destination register is a potential *vpc* candidate, the updated value is added to the updated value list. Case 4 (Lines 54-59): for a bit-masking instruction, its operands being masked are added to the *vflag* candidate set.

Case 5 (Lines 2-29): when encountering an indirect jump instruction, the analyzer sequentially verifies the satisfaction of Rule ⓒ (Line 8), Rule ⓐ (Line 13) and Rule ⓑ (Line 16). In particular, Rule ⓒ is verified for each local *vpc* candidate; Rule ⓐ is verified for the combination of each global *vpc* candidate and each local *vpc* candidate. Rule ⓑ is verified for the combination of each global *vflag* candidate and each local *vpc* candidate. If all rules are satisfied, we consider a virtualized control transfer to be identified. In such a case, the identified conditional transfer is represented by the updated virtual value of *vpc* that satisfies all rules, along with the virtual value of the corresponding *vflag* (Lines 17-20). The identified conditional transfer is then added to the result (Line 21). To avoid excessive overhead, we periodically remove aged candidates from further consideration (Lines 31-36).

During the analysis process, we perform a forward dynamic taint analysis. We maintain taint markings (i.e., taint sources) for each abstract variable *var*, which are a set of abstract variables (i.e., *vpc* candidates or *vflag* candidates), from which the abstract variable receives taints. By invoking the `GetTaintSrcs` function, we can retrieve the taint markings of an abstract variable. For each instruction instance, taint markings are propagated (via the `PropagateTaint` function) from its source operand(s) to its destination operand(s) using taint mapping function that is based on the semantics of the instruction [49]. Both data dependence and control dependence are considered. The details of taint propagation are omitted, as they are standard [57]. Our taint analysis supports on-demand addition and removal of taint sources, allowing for dynamic adjustment of *vpc* candidates. The function `AddTaintSrc` adds a new taint source for propagation. The function `RemoveTaintSrc` removes a specified taint source, disabling its further propagation and deleting it from the taint markings of affected abstract variables. Appendix C uses an example to illustrate the identification of virtualized conditional transfers.

## 3.3 Reverse Predicate Outcomes

Once a virtualized conditional transfer is identified, a straightforward approach to forced execution is to direct *vpc* to follow the alternate branch. This requires obtaining the virtual address of the target branch. However, these target addresses are encrypted within the bytecode, as a security measure implemented by commercial virtualization obfuscators. Different obfuscators, and even different configurations of the same obfuscator, employ distinct decryption methods. It requires substantial reverse-engineering effort to uncover the encrypted target addresses for each obfuscator configuration.

**Algorithm 2:** reversing predicate outcomes.

```
1  Input   : cond_xfer:         ConditionalTransfer
   Output  : vflag2mutation:    VirtualVarValue → Mutatation
   Procedure: Main()
2  vflag2mutation ← ()
3  ⟨vpc, vflag⟩ ← cond_xfer
4  SetBreakpoint(vflag.insn); SetBreakpoint(vpc.insn)
5  for policy in mutation_polices do
6  |  vflag′ ← ApplyPolicy(policy, vflag)
7  |  if Fork() then
8  |  |  ret ← Exe(vpc, vflag′)
9  |  |  if ret = OK then
10 |  |  |  vflag2mutation[⟨vflag.insn, vflag.var, ∗⟩] ← policy
11 |  |  |  break
12 UnsetBreakpoint(vflag.insn); UnsetBreakpoint(vpc.insn)
13 return vflag2mutation
   Procedure: Exe(vpc,vflag)
14 while True do
15 |  ⟨insn, var, val⟩ ← ExecuteUtilBreakpoint()
16 |  if insn = vflag.insn and var = vflag.var then
17 |  |  SetProgramState(vflag.var, vflag.val)
18 |  else if insn = vpc.insn and var = vpc.var and val ≠ vpc.val then
19 |  |  return OK
20 |  else if exception_raised or program_terminated then
21 |  |  return ERROR
```

An alternative approach is to reverse the predicate outcome. In native code, this can be achieved by directly flipping specific bits in the native flag register that control the target conditional jump instruction. However, reversing predicate outcomes of a virtualized conditional transfer is more challenging. This process requires understanding the structure of *vflag*, which is specific to the obfuscation scheme and may differ substantially from the native flag register. Statically recovering the structure of *vflag* and identifying the critical bits that influence branch outcomes is a non-trivial task.

In this paper, we develop a fuzzing-based approach to reverse predicate outcomes. In particular, given a virtualized conditional transfer (represented by the virtual variable values of *vpc* and *vflag*) to be reversed, the fuzzing process continually applies different mutation polices to alter the virtual value of *vflag* until the virtual value of *vpc* changes. We support AFL-like mutation policies [60], include bit/byte flipping, arithmetic operations, etc. Although the search space consists of all possible integer values, since most predicates are simple linear combinations of certain bits of *vflag*, the AFL-like mutation policies are sufficient to reverse these predicates.

Algorithm 2 describes how our executor determines the effective mutation policy for reversing predicate outcomes. Given the virtualized conditional transfer to be reversed, our executor runs the target binary in the monitored mode and sets breakpoints in the emulating instruction instances related to the virtual variable values of the corresponding *vpc* and *vflag* (Line 4). The fuzzing process iterates over predefined mutation policies (Line 5), each of which alters the virtual variable value of *vflag* in different ways (Line 6). To improve performance, the fuzzing process operates in the fork-and-exec mode (Lines 7-11). In this mode, the target binary undergoes loading, linking and initialization only once, and

is then cloned using copy-on-write. In each execution, when the breakpoint related to $vflag$ is triggered, the program state is modified to alter the virtual variable value of $vflag$ (Lines 16-17); when the breakpoint related to *vpc* is triggered, the program state is examined to check whether the virtual value of *vpc* changes (Lines 18-19). The mutation policy on $vflag$ that results in changes to *vpc* is determined as the effective mutation policy for the given conditional transfer.

It is worth noting that the fuzzing process does not need to be performed all the time for every virtualized conditional transfer. Indeed, all virtualized conditional transfers interpreted by the same virtual machine use the same virtual handlers (emulating instructions). The fuzzing process is only necessary for the first virtualized conditional transfer. The determined effective mutation policy will then be used for future virtualized conditional transfers interpreted by the same virtual machine.

# 4 Evaluation

## 4.1 Experimental Setup

**Benchmarks.** Following prior deobfuscation and forced execution approaches [46, 59], we use two types of benchmarks for evaluation. The first one consists of benchmarks with ground truth. It includes the Hash benchmark [46], which contains 10 hash function programs; the SPEC2017 benchmark [23], which consists of real-world programs covering a wide spectrum of computing tasks; and 6 classic non-obfuscated malware samples, representing a broad range of malware families. The Hash and SPEC2017 benchmarks are used to evaluate the accuracy of VMForce, while the classic malware samples are obfuscated using various options of different obfuscators to evaluate the effectiveness and generality of VMForce. The second one consists of real-world virtualization-obfuscated malware samples collected by VirusShare [16] in recent five years. These malware samples are used to evaluate the capability of VMForce in exposing malware behaviors.

**Virtualization-based obfuscators.** The widely used commercial virtualizers include VMProtect [18], Themida [13], Code Virtualizer [1], WinLicense [19], Enigma [4], Obsidium [8]. Themida, Code Virtualizer and WinLicense are distinct products from the same company, all sharing the same virtual machine. Analyzing any one of them can provide insights that are applicable to the others. In our evaluation, we focus on VMProtect, Themida, Enigma, Obsidium, given their popularity and uniqueness. Note that these obfuscators do not support all benchmark items due to their inherent limitations.

**Competitors.** We compare VMForce with four copular deobfuscation tools in terms of identifying virtualized conditional transfers, including FKVMP [5], Oreans UnVirtualizer [10], Generic Deobfuscator [57], and CIA [40]. We do not include a side-by-side comparison with VMAttack [32] and VMHunt [55], as these tools are not designed to identify specific virtualized instructions. Rotalumé [50] is also excluded from the comparison, since it is not open-source. Additionally, we integrate VMForce and PMP into an augmented variant, VMForce$^{++}$, which is capable of forcing both native and virtualized branches. We compare VMForce and VMForce$^{++}$ with Cuckoo [2] and PMP [59] in terms of identifying malware behaviors. Cuckoo is an automated malware analysis system that provides a sandbox environment for executing malware and monitoring its behaviors. PMP is a state-of-the-art forced execution tool for native code.

**Computing resources.** The experiments are conducted on virtual machines to sandbox potentially malicious behaviours. We use VMware as the hypervisor. The host machine has 20-core CPU (Intel® Xeon® Silver 4210 @ 2.20GHz) and 32GB memory. The guest machine has one CPU core and 4GB memory, running the Windows 10 operating system.

## 4.2 Benchmarks with Ground Truth

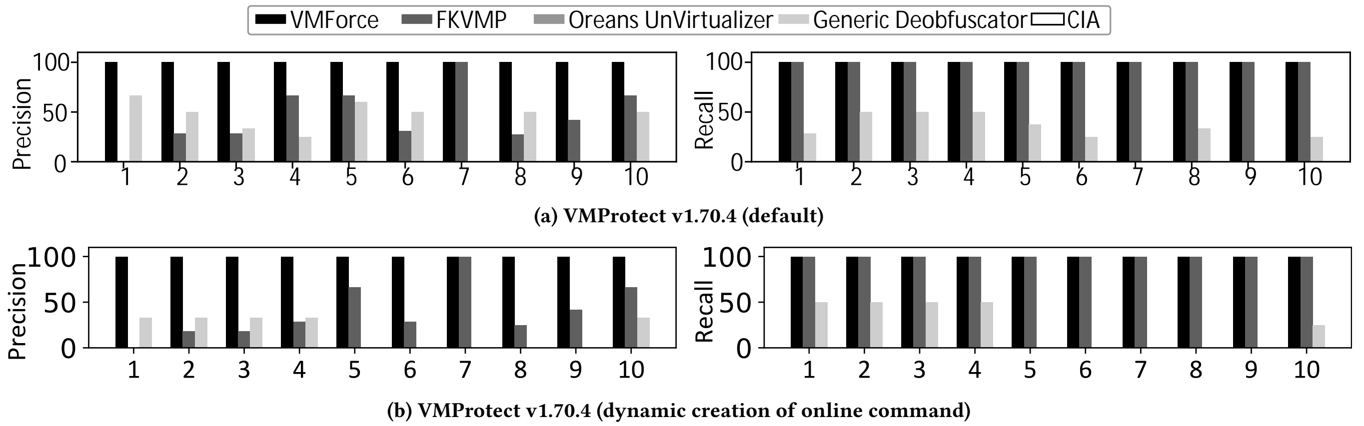
*4.2.1 Accuracy.* We use the Hash and SPEC2017 benchmarks to evaluate the accuracy of VMForce in identifying virtualized conditional transfers. In this evaluation, we do not apply forced execution. Instead, we run the programs with the inputs provided by the benchmarks and check whether the encountered virtualized conditional transfers can be correctly identified. For each benchmark program, we define *CT*, *RCT* and *ORCT* to represent the sets of conditional transfers in the original program, conditional transfers reachable with the benchmark-provided inputs, and reachable conditional transfers selected for obfuscation, respectively. Clearly, we have $ORCT \subseteq RCT \subseteq CT$. For each tool, *ICT* denotes the identified conditional transfers, *CICT* represents the correctly identified conditional transfers, where $CICT = ICT \bigcap ORCT$. Anchor instructions [40] are instrumented during compilation to help align the conditional transfers before and after obfuscation. We then compute precision and recall as in Equation 1 and Equation 2, respectively.

$$precision = \frac{|CICT|}{|ICT|} \tag{1}$$

$$recall = \frac{|CICT|}{|ORCT|} \tag{2}$$

**Results for Hash.** We compare VMForce with FKVMP, Oreans UnVirtualizer, Generic Deobfuscator, and CIA in identifying virtualized conditional transfers obfuscated by the latest versions of VMProtect (v3.8.1), Themida (v3.0.4), Enigma (v6.70), and Obsidium (v1.8.5) with default configurations. For each Hash program, all of its functions are virtually obfuscated. None of the competitors are effective with the latest versions of the obfuscators, while VMForce achieves 100% precision and recall in identifying virtualized conditional transfers.

We also compare VMForce with the competitors using the classic versions of VMProtect (v1.70.4), Themida (v2.1.2), Enigma (v1.6.7), and Obsidium (v1.8.5) with various configurations. The results for VMProtect are presented in Figure 5. We have the following observations: ① FKVMP achieves the same recall as VMForce across all programs, but its precision is generally much lower, since it does not differentiate between conditional and unconditional transfers and reports as many control transfers as possible; ② Oreans UnVirtualizer fails on VMProtect-obfuscated programs, as it relies on bytecode patterns specific to Themida to identify virtualized conditional transfers; ③ CIA also fails to identify virtualized conditional transfers, as the mapping rules for conditional transfers are neither reusable nor learnable; ④ Generic Deobfuscator demonstrates




(a) VMProtect v1.70.4 (default)

(b) VMProtect v1.70.4 (dynamic creation of online command)

**Figure 5: Accuracy of various tools in identifying virtualized conditional transfers on Hash programs obfuscated by VMProtect.**

**Table 1: Generality of VMForce across diverse virtualization-obfuscation configurations in VMProtect.**

| Obfus-cators | Config-uration | Interp-retation | Sample 1 | | | | Sample 2 | | | | Sample 3 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | File Size (KB) | Raw Exe (ms) | Forced Exe (min) | Hidden Syscalls | File Size (KB) | Raw Exe (ms) | Forced Exe (min) | Hidden Syscalls | File Size (KB) | Raw Exe (ms) | Forced Exe (min) | Hidden Syscalls |
| Original | N/A | N/A | 7 | 66 | 4.10 | 67 | 60 | 68 | 3.12 | 55 | 11 | 68 | 4.16 | 66 |
| VMProtect v1.8.0 | DF | DD | 26 | 70 | 5.42 | 67 | 76 | 69 | 3.71 | 55 | 29 | 69 | 5.37 | 66 |
| | CI | DD | 48 | 72 | 5.74 | 67 | 92 | 72 | 3.98 | 55 | 51 | 72 | 5.67 | 66 |
| | HC | DD | 32 | 72 | 5.54 | 67 | 84 | 71 | 3.83 | 55 | 38 | 71 | 5.55 | 66 |
| | CI + HC | DD | 75 | 75 | 8.16 | 67 | 104 | 75 | 4.66 | 55 | 82 | 75 | 8.27 | 66 |
| VMProtect v2.13.8 | DF | DD | 36 | 71 | 6.35 | 67 | 84 | 71 | 6.76 | 55 | 41 | 71 | 5.99 | 66 |
| | CI | DD | 68 | 74 | 11.22 | 67 | 96 | 73 | 8.01 | 55 | 70 | 73 | 10.87 | 66 |
| | HC | DD | 43 | 67 | 9.86 | 67 | 88 | 68 | 7.89 | 55 | 68 | 68 | 9.65 | 66 |
| | CI + HC | DD | 104 | 78 | 15.73 | 67 | 116 | 79 | 9.93 | 55 | 106 | 79 | 16.04 | 66 |
| VMProtect v3.8.1 | CPLX=20 | TH | 174 | 69 | 9.12 | 67 | 196 | 72 | 8.24 | 55 | 184 | 72 | 10.08 | 66 |
| | CPLX=50 | TH | 271 | 80 | 23.68 | 67 | 256 | 83 | 15.67 | 55 | 282 | 83 | 24.64 | 66 |

N/A: not applicable; DF: default; CI: check integrity; HC: hide constants; CPLX: complex; DD: decode-dispatch; TH: threaded.

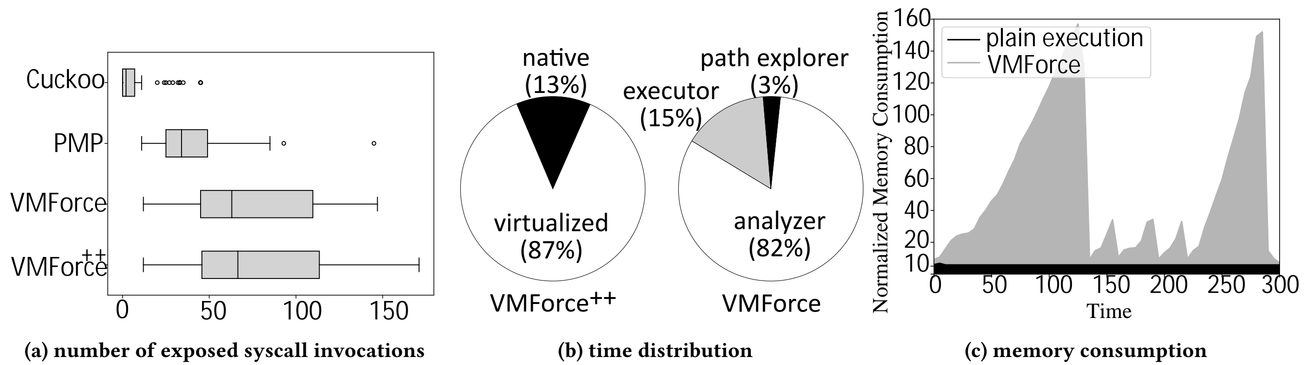
significantly lower precision and recall than VMForce across all programs, and even fails on certain programs with complex obfuscation options. The results for Themida, Enigma and Obsidium are presented in Figure 9 and Figure 10 in Appendix G. We further provide a case study in Appendix E.1 to illustrate why VMForce outperforms the competing approaches.

As discussed in §2.2, Generic Deobfuscator is not designed for identifying individual virtualized instructions. Our evaluation overestimates its potential: we artificially insert anchor instructions before and after each original conditional transfer prior to obfuscation, and manually verify whether the deobfuscated code within these anchor boundaries is semantically equivalent to the original conditional transfer instructions.

**Results for SPEC2017.** We use the latest available versions of the obfuscators to obfuscate these programs. For programs smaller than 1MB (e.g., *mcf*), we attempt to obfuscate all functions that are supported by the individual obfuscators. For larger programs (e.g., *xalancbmk*), obfuscating too many functions significantly increases the overhead and reduces their utility. As a result, we focus on obfuscating only a few functions that contain a substantial number of conditional transfers. The functions are ranked in descending order based on the numbers of conditional transfers, with the top 10% of functions prioritized for obfuscation. The obfuscated programs are then executed using the inputs provided by the benchmark suite, and the set of virtualized conditional transfers is collected to evaluate the accuracy. In this evaluation, we do not compare VMForce with other competitors, as they perform poorly even on small-scale benchmarks like Hash. The results are presented in Table 4 in Appendix G. We observe that for each obfuscator, VMForce achieves 100% recall and nearly 100% precision, with the exception of 5 false positives in the *xalancbmk* program. We show a case study of false positives in Appendix E.2.

*4.2.2 Effectiveness & Generality.* We select 6 classic non-obfuscated malware samples and obfuscate them using different versions and configurations of commercial virtualization obfuscators. These samples represent a broad range of malware families, including downloader, agent, backdoor, ransom, etc. Three of them are in-the-wild distributions to be obfuscated with VMProtect, while the others are compiled from source code and obfuscated with Themida, Enigma and Obsidium. For each original malware sample, we first identify its hidden syscall invocations as the ground truth through a contrastive analysis using the results from PMP and Cuckoo. We then obfuscate the functions involving these hidden syscall invocations. VMForce executes the obfuscated sample and checks whether the obfuscated hidden behaviors can be exposed.

**Results.** Table 1 presents the results for VMProtect. We select three representative versions of VMProtect, including v1.8.0, v2.13.8 and




(a) number of exposed syscall invocations (b) time distribution (c) memory consumption

**Figure 6: Overall results of malware analysis.**

v3.8.1. These versions cover all generations of the product. Regarding obfuscation configurations, we focus on those that influence the identification of conditional transfers, including "*check integrity*" option that enables verification of the integrity of virtualized code, "*hide constants*" option that obscures the addresses of variables, and "*complexity*" option that determines the strength level of obfuscation. The ground truth for each sample is shown in the "`Hidden Syscalls`" cells of the "`Original`" row, while the other "`Hidden Syscalls`" cells display the numbers reported by VMForce.

Based on the results, we have the following observations: ① the stronger the obfuscation, the larger the size of the generated obfuscated program and the slower its execution speed; ② on average, VMForce slows down the execution speed by 115% on the obfuscated programs compared to the force execution on the original programs using PMP; ③ VMForce successfully exposes all hidden syscalls in the obfuscated programs, regardless of the obfuscation tool and configuration, achieving an accuracy of 100%. The results demonstrate the effectiveness and generality of VMForce in exposing hidden behaviors. Similar conclusions hold for other obfuscators, detailed in Table 5 and Table 6 in Appendix G.

## 4.3 Real-World Malware Samples

From the malware samples provided by VirusShare, we randomly select 100, 50, 30, 20 samples protected by VMProtect, Themida, Enigma, and Obsidium, respectively. The number of selected samples for each obfuscator reflects its relative prevalence among the collected malware. The selection is facilitated using Detect-It-Easy [3], a widely used tool for determining file types. The selected samples cover a diverse range of malware families, including command-and-control, botnets, ransomware, etc. We compare VMForce and VMForce$^{++}$ with Cuckoo and PMP on these malware samples to evaluate their ability to expose hidden malware behaviors. The selected samples also cover mainstream versions of each obfuscator, including VMProtect versions 1.X, 2.X, 3.X, and the latest 3.6+, as well as Themida versions 2.X and 3.X. Notably, there are currently no known tools that can reliably and fully determine the specific version or internal architecture of the obfuscator used in an unknown malware sample. Following standard malware analysis practices [29, 45, 59], we capture malware behaviors by monitoring the system library APIs (referred to as *syscalls*) invoked by the malware sample. Intuitively, the greater the number of syscall invocations exposed during execution, the more predicate conditions are triggered, which leads to the exploration of more program paths and the exposure of more malware behaviors.

The analysis environments are standardized across all the compared analysis tools to ensure a fair comparison. Specifically, we configure these analysis tools to monitor the same set of syscalls. The list of monitored syscalls can be found in [7]. Each individual invocation of a syscall is treated as separate behavior, if its arguments exhibit significant difference with the other invocations of the same syscall. Integer arguments are not distinguished, but string arguments are considered if their similarity is above a pre-defined threshold (80% in this study). The dissimilarity of structure-typed arguments is determined by their critical fields, and the pointer arguments are examined based on their dereferenced values. As for execution time, a recent study [39] suggests that two minutes of execution is sufficient for a malware to exhibit its malicious behaviors. We set the analysis time for each malware sample to ten minutes to balance time constraints and test coverage. For fairness, all comparison tools are also given a ten-minute analysis window. We also extend the analysis to twenty minutes per sample and observe no additional hidden behaviors, thereby confirming that a ten-minute window is adequate for our evaluation.

**Results.** Figure 6a presents the number of unique syscall invocations exposed by different tools. We make the following observations: ① PMP and VMForce expose 5.81 times and 10.97 times as many syscall invocations as Cuckoo, respectively. This suggests that many malware samples conceal hidden behaviors safeguarded by specific conditions; ② VMForce exposes 89% more syscall invocations than PMP on average. It implies that the virtualization-based obfuscation often obfuscate trigger conditions for hidden behaviors; ③ PMP exposes more syscall invocations than VMForce in 3.5% of malware samples, primarily those with few obfuscated trigger conditions, where VMForce's virtualization analysis incurs greater time overhead then PMP's native analysis; ④ VMForce$^{++}$ exposes the highest number of syscall invocations, as it forcibly executes both the native and virtualized code. Nonetheless, this count is lower than the combined number exposed by PMP and VMForce with the limited time budget. In Appendix F, we present two case studies.

We also study the time distribution and memory consumption. On average, VMForce$^{++}$ and VMForce take 37 and 29 seconds, respectively, for a round of forced execution. Figure 6b breaks down the time consumption of VMForce$^{++}$ and VMForce. As shown, during a forced execution, the virtualized code consumes 87% of the time, while the native code (under Pin) takes only 13%. Regarding the forced execution of virtualized code, the analyzer (identifying virtualized conditional transfers and fuzzing to reverse predicate outcomes) occupies the most time (82%), followed by the executor (15%) and the path explorer (3%). Figure 6c presents the normalized memory consumption changes during the raw and the forced execution. Plain execution consumes an average of 6.25MB of memory, with peak usage reaching approximately 7.32MB. VMForce introduces 65 times (391.97MB) as much memory consumption as the plain execution on average, with a peak value of 197 times (0.95GB). Such memory consumption is acceptable in malware analysis.

We further investigate the feasibility of virtualized conditional transfers forced by VMForce. Existing deobfuscation tools struggle with understanding virtualized conditional transfers, making it impractical to automatically verify their feasibility. Manually reverse-engineering virtualization-obfuscated samples is considerably more difficult and requires significantly more effort than native samples. Therefore, we conduct a study based on a sampling approach. We first randomly select 20 virtualized conditional transfers forced by VMForce, which expose additional behaviors compared to plain execution. Next, we manually check the selected transfers to examine the conditions required to satisfy the virtualized predicates. Finally, our manual analysis confirms that 17 out of 20 virtualized conditional transfers are feasible. We are unable to determine the remaining 3 cases due to the complexity of the virtualized program logic, and conservatively label them as false positives.We also employed an open-source machine learning project[22] that determines whether a malware sample exhibits malicious behavior by analyzing its syscalls. Detailed experimental results are presented in Table 7, Table 8, Table 9 and Table 10.

## 5 Discussion

Although VMForce is both effective and generalizable to popular commercial virtualization obfuscators, it has limitations and may be susceptible to potential adversarial attacks.

**Scope and Assumptions.** This study focuses solely on malware samples protected by commercial virtualization obfuscators. We exclude those using customized virtualization obfuscators due to their limited prevalence. VMForce assumes that the malware samples are obfuscated using generic virtualization engines involving interactions among $vpc$, $vflag$, and bytecode. Adversaries could potentially circumvent it by customizing the virtualization mechanism, for example, by implementing a non-standard approach to handling branch conditions without introducing explicit data structures like $vflags$. However, such customized virtualization schemes typically increase implementation complexity, reduce the robustness of the obfuscated program, and introduce substantial performance overhead. These drawbacks make such techniques less practical and rarely adopted in real-world malware samples observed in the wild.

**Polymorphism and Nested VM.** The latest versions of VMProtect and Themida introduce polymorphism at the virtual machine level, where multiple virtual machines coexist within a malware sample, responsible for different parts of the code. Obfuscation remains coherent within each virtual machine. VMForce remains robust in the presence of such form of polymorphism. Reverse-engineering analyses [15] further show that Themida's shark, puma, and eagle virtual machines are nested, with inner virtual handlers virtualized by an outer virtual machine. According to the official manual [14], nesting causes severe slowdown and instability. We have not found real-world malware samples protected with nested virtual machines and leave their effective handling as future work.

## 6 Related Work

**Malware analysis.** Malware analysis can be broadly categorized into static and dynamic approaches. Static analysis [17, 37] extracts structural information or patterns from the binary and matches them with known signatures. Dynamic analysis [26, 36, 58] executes the binary in a sandbox and records its behaviors to detect malicious actions, making it immune to obfuscation. However, the quality of analysis depends on the execution environment, as malicious behaviors are only exposed when appropriate conditions are met.

**Forced execution.** There are several forced execution tools developed on different platforms, including desktop binary [29, 45, 59], Android runtime [51] and JavaScript engine [31, 33]. Previous studies put considerable efforts to ensure a crash-free execution. While these tools can expose malware behaviors without setting up environment, they are unable to forcibly execute the virtualization-obfuscated malware due to the virtualized conditional control transfer instructions.

**Deobfuscation.** Early deobfuscation technique [38, 53] utilized static analysis to simplify the obfuscated binary. Unfortunately, they leveraged a static analysis which is not applied to highly obfuscated binary, e.g., virtualization-obfuscated malware. The mainstream studies [32, 50, 55] of deobfuscation techniques rely on the knowledge of the code virtualization mechanism. These methods have limited generalization when their assumptions do not hold. A recent work, CIA [40], constructs a special program to extract reusable knowledge but fails to capture conditional transfer instructions. Achieving comprehensive deobfuscation remains a challenge.

## 7 Conclusion

In this paper, we propose VMForce, a forced execution solution for detecting virtualization-obfuscated malware. It features a lightweight identification of virtualized conditional transfers and fuzzing-based reversal of predicate outcomes. VMForce leverages a critical observation that the virtualization needs to follow specific data- and control-flow in order to preserve the original semantics of control transfers, hence is general to popular obfuscation methods. The evaluation results show that VMForce can accurately identify virtualized conditional transfers and effectively expose hidden malware behaviors obfuscated by virtualization.

## Ethical Considerations

We have carefully considered the ethical implications of developing forced execution techniques for virtualized malware analysis. All experiments were conducted in controlled environments using malware samples exclusively obtained from authorized benchmark datasets (e.g., VirusShare) to prevent any risk of unintended propagation. The research was designed with security community norms in mind, ensuring our methodology contributes primarily to improving defensive capabilities rather than advancing offensive tools. To mitigate potential negative consequences, we have implemented strict experimental safeguards and carefully structured our findings to prioritize responsible disclosure.We have determined that proceeding with this research is ethically justified after evaluating the beneficence principle by weighing the substantial benefits to malware defense against theoretical misuse risks.

## Open Science

In compliance with open science principles, we have made all experimental data publicly available at anonymous repository https://anonymous.4open.science/r/Forced-Execution. We will release the full source code of our analysis tool upon publication to facilitate knowledge sharing and collaborative research within the security community.

## References


[1] 2024. Code Virtualizer. https://www.oreans.com/CodeVirtualizer.php.
[2] 2024. Cuckoo. https://cuckoosandbox.org/.
[3] 2024. Detect It Easy. https://github.com/horsicq/Detect-It-Easy.
[4] 2024. Enigma. https://www.enigmaprotector.com/.
[5] 2024. FKVMP. https://github.com/lmy375/awesome-vmp.
[6] 2024. Intel 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.
[7] 2024. The List of Monitored Syscalls. https://anonymous.4open.science/r/Forced-Execution/SyscallList.pdf.
[8] 2024. Obsidium. https://www.obsidium.de/home.
[9] 2024. OllyDbg. https://www.ollydbg.de/.
[10] 2024. Oreans UnVirtualizer. https://www.unknowncheats.me/forum/downloads.php?do=file&id=11571.
[11] 2024. Pin - A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.
[12] 2024. Taint Analysis Engine of Generic Deobfuscator. https://www2.cs.arizona.edu/projects/lynx-project/Samples/Obfuscated/.
[13] 2024. Themida. https://oreans.com/themida.php.
[14] 2024. ThemidaHelp. https://www.oreans.com/help/tm/.
[15] 2024. Tuts4you. https://forum.tuts4you.com/topic/37469-devirtualizeme-themida-2350-full/.
[16] 2024. VirusShare. https://virusshare.com/.
[17] 2024. Virustotal. https://www.virustotal.com/gui/home/upload.
[18] 2024. VMProtect. https://vmpsoft.com/.
[19] 2024. WinLicense. https://oreans.com/WinLicense.php.
[20] Acronis. 2024. Acronis End-of-Year Cyberthreats Report. https://www.acronis.com/en-us/pr/2024/acronis-end-of-year-cyberthreats-report- uncovers-222-surge-in-email-attacks-during -2023.
[21] Australian Digital Health Agency. 2022. Cyber Security Report 2022. https://www.digitalhealth.gov.au/sites/default/files/documents/cyber-security-report-2022.pdf.
[22] ArfatKadvekar. [n. d.]. https://github.com/ArfatKadvekar/malware-detection-ML.
[23] ATA. 2024. SPEC2000. https://www.spec.org/cpu2017/.
[24] James R Bell. 1973. Threaded code. *Commun. ACM* 16, 6 (1973), 370–372.
[25] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). 643–659.
[26] Ahmet Salih Buyukkayhan, Alina Oprea, Zhou Li, and William K. Robertson. 2017. Lens on the Endpoint: Hunting for Malicious Software Through Endpoint Data Analysis. In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017*, Marc Dacier, Michael D. Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.). Springer, 73–97.
[27] Kevin Coogan, Gen Lu, and Saumya K. Debray. 2011. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*, Yan Chen, George Danezis, and Vitaly Shmatikov (Eds.). 275–284.
[28] Floris Gorter, Cristiano Giuffrida, and Erik van der Kouwe. 2023. Enviral: Fuzzing the Environment for Evasive Malware Analysis. In *Proceedings of the 16th European Workshop on System Security, EUROSEC 2023*, Jason Polakis and Erik van der Kouwe (Eds.). 8–14.
[29] Dongnan He, Dongchen Xie, Yujie Wang, Wei You, Bin Liang, Jianjun Huang, Wenchang Shi, Zhuo Zhang, and Xiangyu Zhang. 2024. Define-Use Guided Path Exploration for Better Forced Execution. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 287–299.
[30] Mattias Holm. 2015. Emulator Performance Study. *Simulation & EGSE Facilities for Space Programmes* (2015).
[31] Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. 2018. Jsforce: A forced execution engine for malicious javascript detection. In *Security and Privacy in Communication Networks: 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22–25, 2017, Proceedings 13*. Springer, 704–720.
[32] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. 2017. VMAttack: Deobfuscating Virtualization-Based Packed Binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*. ACM, 2:1–2:10.
[33] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-Force: Forced Execution on JavaScript. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017*. 897–906.
[34] Johannes Kinder. 2012. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. IEEE Computer Society, 61–70. doi:10.1109/WCRE.2012.16
[35] Paul Klint. 1981. Interpretation techniques. *Software: Practice and Experience* 11, 9 (1981), 963–973.
[36] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. 2009. Effective and Efficient Malware Detection at the End Host. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, Fabian Monrose (Ed.). USENIX Association, 351–366. http://www.usenix.org/events/sec09/tech/full_papers/kolbitsch.pdf
[37] Christopher Krügel, Engin Kirda, Darren Mutz, William K. Robertson, and Giovanni Vigna. 2005. Polymorphic Worm Detection Using Structural Information of Executables. In *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9, 2005, Revised Papers (Lecture Notes in Computer Science, Vol. 3858)*, Alfonso Valdes and Diego Zamboni (Eds.). Springer, 207–226. doi:10.1007/11663812_11
[38] Christopher Krügel, William K. Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, Matt Blaze (Ed.). USENIX, 255–270. http://www.usenix.org/publications/library/proceedings/sec04/tech/kruegel.html
[39] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. 2021. Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society.
[40] Shijia Li, Chunfu Jia, Pengda Qiu, Qiyuan Chen, Jiang Ming, and Debin Gao. 2022. Chosen-Instruction Attack Against Commercial Code Virtualization Obfuscators. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society.
[41] Zimin Lin, Rui Wang, Xiaoqi Jia, Jingyu Yang, Daojuan Zhang, and Chuankun Wu. 2016. ForceDROID: Extracting Hidden Information in Android Apps by Forced Execution Technique. In *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*. IEEE, 386–393.
[42] Peter S Magnusson and David Samuelsson. 1994. *A compact intermediate format for SimICS*. Swedish Institute of Computer Science.
[43] Andreas Moser, Christopher Krügel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*. IEEE Computer Society, 231–245.
[44] Naveen Namani and Arindam Khan. 2020. Symbolic execution based feature extraction for detection of malware. In *5th International Conference on Computing, Communication and Security, ICCCS 2020, Patna, India, October 14-16, 2020*. IEEE, 1–6.
[45] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium*.

[46] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. 2018. Symbolic Deobfuscation: From Virtualized Code Back to the Original. In *International Conference on Detection of intrusions and malware, and vulnerability assessment.* https://api.semanticscholar.org/CorpusID:49335875

[47] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening Code Obfuscation Against Automated Attacks. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 3055–3073.

[48] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar R. Weippl. 2016. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1 (2016), 4:1–4:37.

[49] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, SP 2010.* 317–331.

[50] Monirul Islam Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2009. Automatic Reverse Engineering of Malware Emulators. In *30th IEEE Symposium on Security and Privacy (SP 2009), 17-20 May 2009, Oakland, California, USA.* 94–109.

[51] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. 2018. Dual-force: understanding WebView malware via cross-language forced execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 714–725. doi:10.1145/3238147.3238221

[52] VirusTotal. 2024. Malware Report. https://www.virustotal.com/gui/file/3110f00c1c48bbba249 31042657a21c55e9a07d2ef315c2eae0a4222346231 94.

[53] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. 2001. Protection of software-based survivability mechanisms. *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]* (2001), 273–282. https://api.semanticscholar.org/CorpusID:15860593

[54] Xiaolei Wang, Yuexiang Yang, and Sencun Zhu. 2019. Automated Hybrid Analysis of Android Malware through Augmenting Fuzzing with Forced Execution. *IEEE Trans. Mob. Comput.* 18, 12 (2019), 2768–2782.

[55] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* ACM, 442–458.

[56] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). 732–744.

[57] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015.* IEEE Computer Society, 674–691.

[58] Heng Yin, Dawn Xiaodong Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson (Eds.). ACM, 116–127. doi:10.1145/1315245.1315261

[59] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. PMP: Cost-effective Forced Execution with Probabilistic Memory Pre-planning. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 1121–1138. doi:10.1109/SP40000.2020.00035

[60] M. Zalewski. 2014. "American Fuzzy Lop". http://lcamtuf.coredump.cx/afl/.

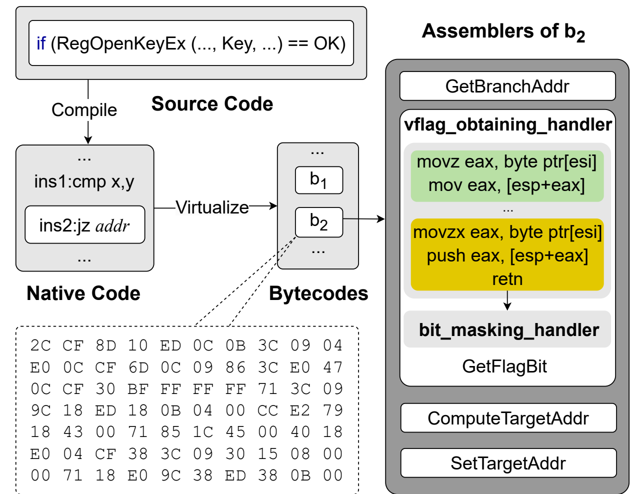




**Figure 7: The virtualization of conditional transfer.**

# A Virtualization of Conditional Transfers

As shown in Figure 7, a native conditional transfer instruction (e.g., `jz`) is decomposed into several assembler units, including `GetFlagBit` that acquires the specific bit value of $vflag$ from the virtual context, `GetBranchAddr` that retrieves the addresses of the true and/or false branches from the bytecode area, `ComputeTargetAddr` that calculates the target address based on the flag bit value and the branch addresses, `SetTargetAddr` that updates $vpc$ with the calculated address. In particular, `GetFlagBit` contains two virtual handlers, including `vflag_obtaining_handler` and `bit_masking_handler`. The execution code (highlighted in green) of `vflag_obtaining_handler` obtains the $vflag$ value from a specific offset in the virtual context. The jump code (highlighted in yellow) fetches the subsequent bytecode instruction and transfers control to `bit_masking_handler`.

# B Contribution of Individual Rules

As discussed in §3.2, the three rules must operate sequentially in the order of Rule ⓒ, Rule ⓐ and Rule ⓑ to reliably identify virtualized conditional transfers. In particular, Rule ⓒ first selects potential virtualized instructions; Rule ⓐ then narrows them down to potential virtualized transfers; and Rule ⓒ finally filters out potential virtualized conditional transfers. To further evaluate the contribution of each rule, we perform an ablation study on the identification of virtualized conditional transfers using the Hash benchmark [46].

In this study, we use VMProtect to obfuscate each hash function and apply different combinations of the three rules to identify virtualized conditional transfers. The evaluation results are presented in Table 2. In the table, $|RCT|$ denotes the number of conditional transfers reachable by running the program using the benchmark-provided inputs; $|ICT|$ denotes the number of identified virtualized conditional transfers. We can observe that the applying Rules ⓒ, ⓐ and ⓑ together correctly identifies all virtualized conditional transfers without any false positives. In contrast, applying only Rules ⓐ and ⓑ results in identifying 5.87 times the correct number, with 82.96% of them being false positives. When only Rule ⓑ is applied,

**Table 2: Contribution of each individual rule evaluated on the Hash benchmark virtualized by VMProtect.**

| Hash Function | \|RCT\| | \|ICT\| | | |
|---|---|---|---|---|
| | | Rules ⓒ+ⓐ+ⓑ | Rule ⓐ+ⓑ | Rule ⓑ |
| #1 Simple MD5 | 7 | 7 | 38 | 621 |
| #2 adler32 | 2 | 2 | 13 | 187 |
| #3 Jenkins's one-at-a-time | 2 | 2 | 11 | 152 |
| #4 SipHash | 2 | 2 | 13 | 174 |
| #5 xxHash | 8 | 8 | 60 | 921 |
| #6 superfasthash | 4 | 4 | 25 | 360 |
| #7 Fowler-Noll-Vo Hash | 1 | 1 | 5 | 78 |
| #8 Spooky Hash | 3 | 3 | 17 | 280 |
| #9 City Hash | 5 | 5 | 20 | 336 |
| #10 Jody Hash | 4 | 4 | 21 | 307 |

the false positive rate can reach as high as 98.89%. Similar results are observed with other commercial virtualization obfuscators, such as Themida, Enigma, and Obsidium.

## C Identification of Virtualized Transfers

Table 3 shows an example of identifying conditional transfers virtualized by VMProtect. The table displays the instruction, the actions, and the taints from left to right. Each row corresponds to the execution of an instruction instance. Newly generated $vpc$ and $vflag$ candidates, updated values and taint sources are highlighted in red. Note that the virtual handlers shown in the table are for illustrative purposes only. Our identification algorithm does not depend on prior knowledge of specific virtual handlers.

The instruction $i_1$ falls into Case 1, where $esi$ is used as the base register for memory addressing, resulting in the effective address $m_1$. The virtual variable values $\langle i_1, esi, v_1\rangle$ and $\langle i_1, m_1, v_1\rangle$ are added to the local and global $vpc$ candidate sets; and the abstract variable $esi$ is marked as taint source. The taint is transitively propagated to the abstract variable $m_{11}$ in the instruction $i_{85}$, which represents the effective address using $ecx$ as the base register for memory addressing. The instruction $i_{53}$ falls into Case 4, which is a bitmasking instruction that masks the $eax$ register. Consequently, $eax$ is added to the global $vflag$ candidate set and marked as taint source. The taint is transitively propagated to $ecx$ in the instruction $i_{85}$, which serves as the base register for memory addressing.

From the above analysis, the $eax$ register in the instruction $i_{85}$ are tainted from both the $esi$ register in the instruction $i_1$ (a $vpc$ candidate) and the $eax$ register in the instruction $i_{53}$. These taints are further transitively propagated to the $esi$ register in $i_{124}$, which eventually taint flow to the native program counter. The instruction $i_{128}$ falls into Case ⑤, where the analyzer tries to verify whether the aforementioned three rules are satisfied. In this example, the $esi$ register is recognized as a $vpc$ candidate in $i_{125}$. This $vpc$ candidate is then updated in $i_{126}$ and is a taint source of the native program counter, thus Rule ⓒ is satisfied. The $esi$ register in the instruction $i_1$ is also a taint source of $vpc$, which makes Rule ⓐ satisfied. Finally, Rule ⓑ is satisfied because the only global $vflag$ candidate (register $eax$ in the instruction $i_{85}$) is a taint source of $vpc$ as well. Therefore, we successfully identify a conditional transfer by the end of this instruction, which demonstrates the feasibility of our approach.

## D Implementation Details

**Path explorer.** VMForce employs the same linear search strategy for path exploration as PMP. In particular, the linear strategy aims to cover all control flow edges. In each round, the linear search selects a new virtualized conditional transfer to enforce, which is the last one whose predicate outcome has not been reversed before.

**Executor.** Forced execution could result in corrupted states, and hence exceptions. To ensure crash-free execution, VMForce adopts a lightweight memory pre-planning approach similar to PMP. It involves pre-allocating a large memory region and filling it with carefully crafted random values before execution. These values are designed so that dereferencing an invalid pointer has a high likelihood of falling into the pre-allocated region, while semantically unrelated invalid pointer dereferences are highly likely to access disjoint pre-allocated regions, avoiding bogus data-flow.

**Analyzer.** The goal of analyzer is to identify virtualized conditional transfers from the execution trace. Unlike a native conditional transfer, which can be represented solely by the address of a conditional jump instruction, a virtualized conditional transfer requires both the value of $vpc$ along with the address of an emulating instruction (serving as the unique identifier), and the value of $vflag$ along with the address of an emulating instruction (for forcibly setting).

**Prototype.** We implement the analyzer and executor of VMForce on top of Pin [11], a well-known dynamic binary instrumentation tool. Pin employs dynamic binary instrumentation rather than debugging. It naturally bypasses many anti-debugging protections that specifically target debugger-based analysis. For the analyzer, we instrument each memory read/write instruction, register update instruction, bit-masking instruction, and indirect jump instruction, taking different actions as explained in Algorithm 1. The taint analysis engine is implemented based on [12]. For the executor, we set breakpoints at the emulating instructions corresponding to the specified conditional transfer and drive the fuzzing process in a fork-and-exec way. To improve performance, the identification of conditional transfers and the reversal of predicate outcomes are performed in a pipeline. We hook system library APIs to capture the program behaviors.

## E Case Studies for Accuracy Evaluation

*E.1 Advantage of* VMForce *over Competitors.* We demonstrate why other competitors perform worse than VMForce. Figure 8a presents the original program logic of two functions, each containing a `jz` instructions corresponding to a loop and a branch statement, respectively. Figure 8b presents snippets of bytecode responsible for the `jz` instructions generated by both the latest and classic versions of Themida. In the classic version, the bytecode contains a fixed prefix. Oreans UnVirtualizer leverages this pattern to recognize virtualized conditional transfers. Unfortunately, the latest version of Themida introduces function-level instruction set randomization, causing identical instruction types in different functions to exhibit different bytecode patterns. Due to the lack of distinctive bytecode patterns, Oreans UnVirtualizer fails to identify virtualized conditional transfers in programs obfuscated by the latest version of Themida.

**Table 3: An example of identifying conditional transfers virtualized by VMProtect.**

| Instruction | | Actions | | | | Taints |
|---|---|---|---|---|---|---|
| | | vpc_cands$^l$ | vpc_cands$^g$ | vflag_cands$^g$ | upd_vals$^l$ | |
| GetBranchAddr | | | | | | |
| $i_1$ | mov eax, [esi] | $\{\langle i_1, esi, v_1\rangle, \langle i_1, m_1, v_1\rangle\}$ | $\{\langle i_1, esi, v_1\rangle, \langle i_1, m_1, v_1\rangle\}$ | {} | {} | $esi \longrightarrow eax$ |
| $i_2$ | lea esi, [esi+4] | $\{\langle i_1, esi, v_1\rangle, \langle i_1, m_1, v_1\rangle\}$ | $\{\langle i_1, esi, v_1\rangle, \langle i_1, m_1, v_1\rangle\}$ | {} | $\{esi : [v_2]\}$ | |
| $i_3$ | mov [ebp+0], eax | $\{\langle i_1, esi, v_1\rangle, \langle i_1, m_1, v_1\rangle\}$ | $\{\langle i_1, esi, v_1\rangle, \langle i_1, m_1, v_1\rangle\}$ | {} | $\{esi : [v_2]\}$ | $eax \longrightarrow m_2$ |
| ... | | | | | | |
| vflag_obtaining_handler | | | | | | |
| $i_{33}$ | movzx eax, byte ptr [esi] | $\{\langle i_{33}, esi, v_3\rangle, \langle i_{33}, m_3, v_3\rangle\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | {} | {} | $esi \longrightarrow eax$ |
| $i_{34}$ | mov eax, [esp+eax] | $\{\langle i_{33}, esi, v_3\rangle, \langle i_{33}, m_3, v_3\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | {} | $\{eax : [v_4]\}$ | $esp \longrightarrow eax$ |
| $i_{35}$ | mov [ebp+0], eax | $\{\langle i_{33}, esi, v_3\rangle, \langle i_{33}, m_3, v_3\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | {} | $\{eax : [v_4]\}$ | $eax \longrightarrow m_4$ |
| ... | | | | | | |
| bit_masking_handler | | | | | | |
| $i_{49}$ | mov eax, [ebp+0] | $\{\langle i_{49}, ebp, v_5\rangle, \langle i_{49}, m_4, v_5\rangle\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | {} | {} | $m_4 \longrightarrow eax$ |
| $i_{50}$ | mov ecx, [ebp+4] | $\{\langle i_{49}, ebp, v_5\rangle, \langle i_{49}, m_4, v_5\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | {} | {} | $m_5 \longrightarrow ecx$ |
| $i_{51}$ | not eax | $\{\langle i_{49}, ebp, v_5\rangle, \{\langle i_{49}, m_4, v_5\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | {} | {} | |
| $i_{52}$ | not ecx | $\{\langle i_{49}, ebp, v_5\rangle, \langle i_{49}, m_4, v_5\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | {} | {} | |
| $i_{53}$ | and eax, ecx | $\{\langle i_{49}, ebp, v_5\rangle, \langle i_{49}, m_4, v_5\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | {} | $ecx \longrightarrow eax$ |
| $i_{54}$ | mov [ebp+0], eax | $\{\langle i_{49}, ebp, v_5\rangle, \langle i_{49}, m_4, v_5\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | {} | $eax \longrightarrow m_6$ |
| ... | | | | | | |
| ComputeTargetAddr | | | | | | |
| $i_{76}$ | mov eax, [ebp+0] | $\{\langle i_{76}, ebp, v_7\rangle, \langle i_{76}, m_6, v_7\rangle\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | {} | $m_6 \longrightarrow eax$ |
| $i_{77}$ | mov ecx, [ebp+4] | $\{\langle i_{76}, ebp, v_7\rangle, \langle i_{76}, m_6, v_7\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | {} | $m_7 \longrightarrow ecx$ |
| $i_{78}$ | add eax, ecx | $\{\langle i_{76}, ebp, v_7\rangle, \langle i_{76}, m_6, v_7\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | {} | $ecx \longrightarrow eax$ |
| $i_{79}$ | mov [ebp+0], eax | $\{\langle i_{76}, ebp, v_7\rangle, \langle i_{76}, m_6, v_7\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | {} | $eax \longrightarrow m_8$ |
| ... | | | | | | |
| $i_{84}$ | mov ecx, [ebp+0] | $\{\langle i_{76}, ebp, v_7\rangle, \langle i_{76}, m_6, v_7\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | {} | $m_8 \longrightarrow ecx$ |
| $i_{85}$ | mov eax, ss:[ecx] | $\{\langle i_{85}, ecx, v_{10}\rangle, \langle i_{76}, m_{11}, v_{10}\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | {} | $ecx \longrightarrow eax, m_2 \longrightarrow eax$ |
| $i_{86}$ | mov [ebp+0], eax | $\{\langle i_{85}, ecx, v_{10}\rangle, \langle i_{76}, m_{11}, v_{10}\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | {} | $eax \longrightarrow m_9$ |
| ... | | | | | | |
| SetTargetAddr | | | | | | |
| $i_{124}$ | mov esi, [ebp+0] | $\{\langle i_{124}, ebp, v_8\rangle, \langle i_{84}, m_9, v_8\rangle\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | {} | $m_9 \longrightarrow esi$ |
| $i_{125}$ | movzx eax, byte ptr [esi] | $\{\langle i_{125}, esi, v_9\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | | $esi \longrightarrow eax$ |
| $i_{126}$ | add esi, 1 | $\{\langle i_{125}, esi, v_9\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | $\{esi : [v_{10}]\}$ | |
| $i_{127}$ | push [0xAC1504 + eax*4] | $\{\langle i_{125}, esi, v_9\rangle, ...\}$ | $\{\langle i_1, esi, v_1\rangle, ...\}$ | $\{\langle i_{53}, ecx, v_6\rangle\}$ | $\{esi : [v_{10}]\}$ | $eax \longrightarrow m_{10}$ |
| $i_{128}$ | retn | {} | {} | {} | {} | $m_{10} \longrightarrow npc$ |

```
void secret(...){
 for(idx!=len) jz0x401593①
 ...
}
int main(...){
 if(argc!=2)... jz0x401625②
}
```

(a) original program logic

**classic version**

```
jz ①:55 01 56 01 00 0C 00 00
jz ②:55 01 56 01 00 0C 00 00
```

**latest version**

```
jz ①:24 48 8C 82 DC 70 FB 00
jz ②:81 E9 3A BD BF 1B DD 14
```

**(b) bytecode of jz in Themida**

**classic version**

```
VM entry point:
  push imm
  call addr
jz handler:
  // esi as fixed vpc
  mov  esi,[ebp]
  ...
  retn
```

**latest version**

```
VM entry point:
// no push              // no push
call addr               jmp addr
jz handler:
// ebp as vpc           // ecx as vpc
mov  eax,[edi]          mov edx,[esi]
mov  ebp,eax            mov ecx,edx
...                     ...
jmp  reg                retn
```

**(c) VM entry point and jz handler in VMProtect**

**Figure 8: Case study of accuracy evaluation.**

Figure 8c presents snippets of related interpreter code from both the latest and classic versions of VMProtect. In the classic version, VMProtect leaves distinctive signatures for fingerprinting. In particular, the entry point of a virtual machine begins with the combination of "push *imm*" and "call *addr*" instructions. The handler for a virtualized transfer ends with a "retn" instruction and contains a "mov *esi*, *mem*" instruction, where *esi* is consistently used as the *vpc*. FKVMP leverages these signatures to identify the entry point of virtual machines and subsequently recognize the handlers for virtualized transfers. Unfortunately, the latest version of VMProtect introduces polymorphism to both the virtual machine entry point and the handlers for virtualized transfers. Different virtualized instances of the same program exhibit different forms, eliminating the distinctive signatures previously used for fingerprinting. Hence, FKVMP fails to identify virtualized transfers in programs obfuscated by the latest version of VMProtect.

```
1   // src code: switch (*(_DWORD *)(a1 + 12) & 0xF)
2   // memory [ebp-0x20] has been read before Line 3
3     mov     eax, [eax-0xc]
4     and     eax, 0xf
5     mov     [ebp-0x20], eax
6     sub     eax, 5        // switch 6 cases
7     ja      0x53dda9      // jump table default case
8     mov     eax, [ebp-0x20]
9     mov     eax, ds:0x53dda9[eax*4]
10    jmp     eax           // switch jump
```

**Listing 2: Assembly code of a false positive case.**

```
1  #define MAX_PATH 260
2
3  int main() {
4    void *ExecuteCode = UnPackPayload();
5    TransferControl(Payload);
6    return 0;
7  }
8  /* Virtualizatoin-Based Obfuscation - Begin */
9  void Payload() {
10   if(!IsVboxVM() && !IsVMwareVM()){
11     TCHAR folder[MAX_PATH];
12     SHGetFolderPath(..., CSIDL_MYDOCS,..., folder);
13     encrypt(folder);
14     ...
15   }
16 }
17   /* Virtualizatoin-Based Obfuscation - End */
```

**Listing 3: A ransom malware virtualized by Themida.**

*E.2 False Positives.* Our experiments show that VMForce accurately identify virtualized conditional transfers in the majority of virtualization obfuscated samples. However, we also found that it produces false positives in some very specific cases. List 2 shows one such case from SPEC2017. The source code for this case is a special switch statement, where the expression for the switch is derived from a bitwise operation applied to the value obtained by dereferencing a pointer parameter. Although the switch statement is not obfuscated, it is mistakenly identified as a virtualized conditional transfer by VMForce.

We explain why this happens. First the analyzer adds [$eax$-0xc] to $vpc_cands^g$ and includes it as a taint source (Line 1). Then, it will adds $eax$ to $vflag_cands^g$ and includes it as a taint source (Line 2). Since [$ebp$-0x20] has been read previously, it is added to $vpc_cands^l$, and the analyzer also adds it to $upd_vals^l$ (Line 3). Subsequently, the analyzer adds [$ebp$-0x20] to $vpc_cands^l$ and includes it as a taint source (Line 6). Eventually, the analyzer checks whether this constitutes a virtualized conditional transfer (Line 8). The taint source of $npc$ is [$ebp$-0x20], which is present in both $vpc_cands^l$ and $upd_vals^l$, satisfying Rule ⓒ. Additionly, [$eax$-0xc] in $vpc_cands^g$ is the taint source of [$ebp$-0x20], satisfying Rule ⓐ. Furthermore, $eax$ in $vflag_cands^g$ is also the taint source of [$ebp$-0x20], satisfying Rule ⓑ. Therefore, this switch statement leads to a false positive. Even though VMForce produces false positives in extreme cases, their impact on the results is limited, as only one round of unnecessary forced execution occurs.

# F Case Studies for Malware Samples

**Case study 1: d70516aa97f6d7d77adb33c2615165a0.** It is a ransom malware that encapsulates the malicious payload to evade static analysis and is obfuscated by Themida (Lines 9-16). Listing 3 depicts a simplified code snippet of the processing logic. Upon execution, the malware unpacks itself (Line 4) and transfers control to the unpacked payload (Line 5). It checks whether it is running in a virtual machine environment (Line 10), which potentially indicates that it is under analysis. The malicious file encryption behaviours (Lines 11-13) are only triggered when it is running on a real machine. The logic payload is very simple, but obfuscated by virtualization.

```
1  #define password "892jf1jed"
2  #define SVREXE "http://184.27.29.150/3o902jd.exe"
3  #define MAX_PATH 260
4
5  /* Virtualizatoin-Based Obfuscation - Begin */
6  int __stdcall WinMain(..., LPSTR lpCmdLine, ...) {
7    char* command = (char*)calloc(60, 1);
8    if (IsDebuggerPresent()) ExitProcess(0);
9    if (!strcmp(lpCmdLine, password)) {
10     HINTERNET hNet = InternetOpen(...);
11     if (hInternet) {
12       HINTERNET hConn = InternetOpenUrl(hNet, ...);
13       if (!hConn) return;
14       InternetReadFile(hConn, command, ...);
15       InternetCloseHandle(hConn);
16       InternetCloseHandle(hNet);
17     }
18     if (!strncmp(command, "je30c", 5)) {
19       system(command + 7);
20     } else if (!strncmp(command, "bs01m", 5)) {
21       URLDownloadToFile(..., SVREXE, "word.exe");
22       ShellExecute(NULL, "open", "word.exe", ...);
23     } else if (!strncmp(command, "pl77r", 5)) {
24       char fileName[MAX_PATH];
25       GetModuleFileName(NULL, fileName, MAX_PATH);
26       DeleteFile(fileName);
27     } else ......
28   }
29   ExitProcess(0);
30 }
31   /* Virtualizatoin-Based Obfuscation - End */
```

**Listing 4: A trojan malware virtualized by VMProtect.**

PMP cannot identify virtualized control transfers or reverse their predicate outcomes, and therefore fails to expose the hidden behaviors. The comparison of different tools on this malware sample is presented in the row with ID 107 in Table 9 of Appendix G. As we can observe, PMP does not expose any hidden behaviors, while VMForce exposes 17. Our manual analysis confirms the feasibility of these hidden behaviors exposed by VMForce. Furthermore, we found that this malware sample appears to have enabled many protection options provided by Themida, such as anti-debugger detection, compression and encryption, entry point virtualization, etc. Although these options have significantly increased the difficulty of analysis, VMForce remains effective.

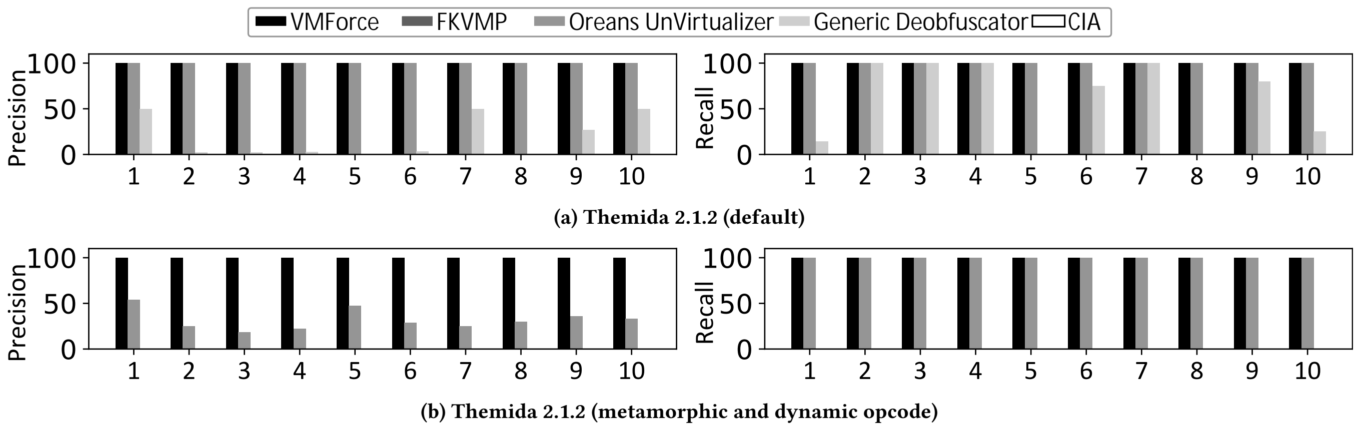
**Case study 2: f0bf41fb52b4ee8515033cdeb2dc1881.** This is a Windows trojan malware. Listing 4 depicts the simplified code snippet. It first checks for the presence of a debugger to decide whether to continue execution or exit (Line 8). Then, it examines whether the running instance is configured with a specific command line (Line 9), which triggers the malware to receive command strings for execution from a remote server (Lines 10-17). Different command strings trigger different actions, such as executing remote command lines (Line 19), downloading and executing remote executable files (Lines 21-22), deleting itself from the file system (Lines 24-26), and so forth. The core logic of the malware is obfuscated by VMProtect in the shadow lines. The result on this malware is presented in the row with ID 66 in Table 8 of Appendix G. Cuckoo captures only two syscalls, i.e., `IsDebuggerPresent` and `ExitProcess`. The obfuscated hidden behaviors can only be detected by VMForce.

## G Evaluation Details

Due to space limitations, we move certain evaluation details that are not essential for main content to the appendix. Figure 9 and Figure 10 present the accuracy results of various tools in identifying virtualized conditional transfers on Hash programs obfuscated by Themida, Enigma and Obsidium, respectively. Table 4 presents the accuracy results of VMForce in identifying virtualized condiitonal transfers on the SPEC2017 benchmark. Table 5 and Table 6 present the generality evaluation of VMForce across different configurations in Themida, Enigma and Obsidium.

## H Prevalence of Virtualized Malware Samples

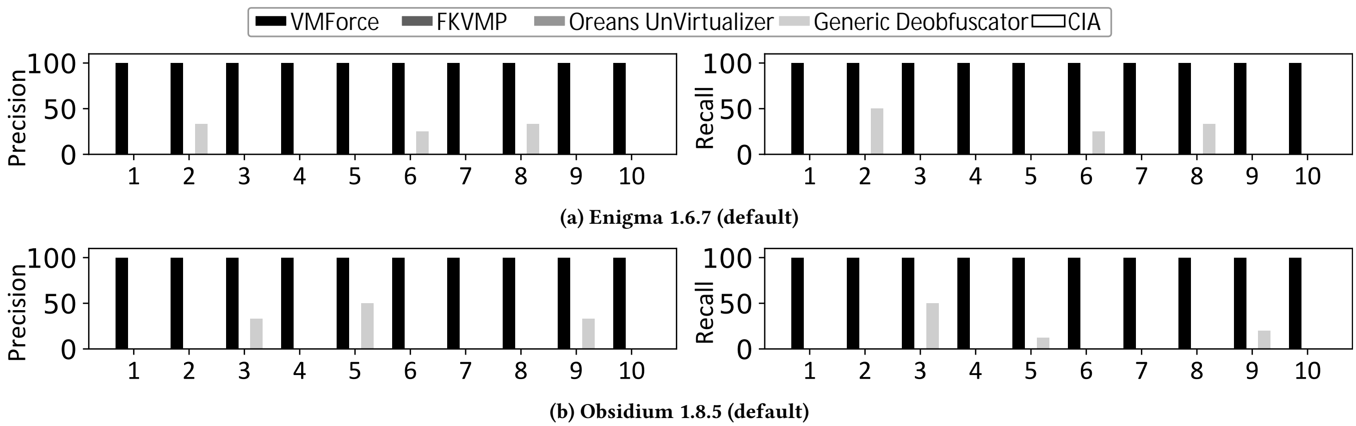
To investigate the prevalence of virtualization-based obfuscation in modern malware, we obtained a large-scale sample set from the VirusShare repository. comprising 1310720 samples from 2015 to 2025. An initial screening with Detect It Easy reveals that 48497 samples (approximately 3.7%) exhibits signatures of virtualized obfuscation. Although such samples still constitute a minority of overall malware, their proportion has grown significantly over the past decade from 0.56% to 5.8%. This highlights the increasing adoption of virtualized obfuscation by malware authors and warrants dedicated research into this challenging evasion technique.




(a) Themida 2.1.2 (default)

(b) Themida 2.1.2 (metamorphic and dynamic opcode)

**Figure 9: Accuracy of various tools in identifying virtualized conditional transfers on Hash programs obfuscated by Themida.**




(a) Enigma 1.6.7 (default)

(b) Obsidium 1.8.5 (default)

**Figure 10: Accuracy of various tools in identifying virtualized conditional transfers on Hash programs obfuscated by Enigma and Obsidium.**

**Table 4: Accuracy of VMForce in identifying virtualized conditional transfers on the SPEC2017 benchmark.**

<table>
<tr><th rowspan="2">Program</th><th rowspan="2">File Size (MB)</th><th colspan="5">Virtualized by VMProtect</th><th colspan="5">Virtualized by Themida</th><th colspan="5">Virtualized by Enigma</th><th colspan="5">Virtualized by Obsidium</th></tr>
<tr><th>|ORCT|</th><th>|ICT|</th><th>|CICT|</th><th>Prec</th><th>Rec</th><th>|ORCT|</th><th>|ICT|</th><th>|CICT|</th><th>Prec</th><th>Rec</th><th>|ORCT|</th><th>|ICT|</th><th>|CICT|</th><th>Prec</th><th>Rec</th><th>|ORCT|</th><th>|ICT|</th><th>|CICT|</th><th>Prec</th><th>Rec</th></tr>
<tr><td>mcf</td><td>0.41</td><td>253</td><td>253</td><td>253</td><td>100%</td><td>100%</td><td>253</td><td>253</td><td>253</td><td>100%</td><td>100%</td><td>253</td><td>253</td><td>253</td><td>100%</td><td>100%</td><td>253</td><td>253</td><td>253</td><td>100%</td><td>100%</td></tr>
<tr><td>omnetpp</td><td>4.73</td><td>1079</td><td>1079</td><td>1079</td><td>100%</td><td>100%</td><td>1147</td><td>1147</td><td>1147</td><td>100%</td><td>100%</td><td>1181</td><td>1181</td><td>1181</td><td>100%</td><td>100%</td><td>1005</td><td>1005</td><td>1005</td><td>100%</td><td>100%</td></tr>
<tr><td>xalancbmk</td><td>14.53</td><td>1294</td><td>1299</td><td>1294</td><td>99.62%</td><td>100%</td><td>1160</td><td>1162</td><td>1160</td><td>99.83%</td><td>100%</td><td>1311</td><td>1312</td><td>1311</td><td>99.92%</td><td>100%</td><td>1108</td><td>1110</td><td>1108</td><td>99.82%</td><td>100%</td></tr>
<tr><td>x264</td><td>1.08</td><td>1309</td><td>1309</td><td>1309</td><td>100%</td><td>100%</td><td>1212</td><td>1212</td><td>1212</td><td>100%</td><td>100%</td><td>1028</td><td>1028</td><td>1028</td><td>100%</td><td>100%</td><td>1052</td><td>1052</td><td>1052</td><td>100%</td><td>100%</td></tr>
<tr><td>deepsjeng</td><td>0.52</td><td>1122</td><td>1122</td><td>1122</td><td>100%</td><td>100%</td><td>1122</td><td>1122</td><td>1122</td><td>100%</td><td>100%</td><td>1122</td><td>1122</td><td>1122</td><td>100%</td><td>100%</td><td>1122</td><td>1122</td><td>1122</td><td>100%</td><td>100%</td></tr>
<tr><td>leela</td><td>0.81</td><td>579</td><td>579</td><td>579</td><td>100%</td><td>100%</td><td>579</td><td>579</td><td>579</td><td>100%</td><td>100%</td><td>579</td><td>579</td><td>579</td><td>100%</td><td>100%</td><td>579</td><td>579</td><td>579</td><td>100%</td><td>100%</td></tr>
<tr><td>xz</td><td>0.49</td><td>1028</td><td>1028</td><td>1028</td><td>100%</td><td>100%</td><td>1028</td><td>1028</td><td>1028</td><td>100%</td><td>100%</td><td>1028</td><td>1028</td><td>1028</td><td>100%</td><td>100%</td><td>1028</td><td>1028</td><td>1028</td><td>100%</td><td>100%</td></tr>
</table>

**Table 5: Generality of VMForce across diverse virtualization-obfuscation configurations in Themida.**

| Obfus-cators | Config-uration | Interp-retation | Sample 4 | | | | Sample 5 | | | | Sample 6 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | File Size (KB) | Raw Exe (ms) | Forced Exe (min) | Hidden Syscalls | File Size (KB) | Raw Exe (ms) | Forced Exe (min) | Hidden Syscalls | File Size (KB) | Raw Exe (ms) | Forced Exe (min) | Hidden Syscalls |
| Original | N/A | N/A | 125 | 72 | 4.05 | 28 | 10 | 240 | 5.54 | 41 | 11 | 40 | 3.01 | 30 |
| Themida v2.4.6 | TW | TH | 1387 | 77 | 5.23 | 28 | 1396 | 251 | 6.38 | 41 | 1475 | 45 | 3.73 | 30 |
| | TR | TH | 2078 | 77 | 5.42 | 28 | 2123 | 253 | 6.40 | 41 | 2210 | 48 | 3.87 | 30 |
| | TB | TH | 2700 | 79 | 6.07 | 28 | 2672 | 257 | 6.66 | 41 | 2815 | 53 | 4.15 | 30 |
| | FW | TH | 967 | 82 | 8.11 | 28 | 986 | 265 | 8.23 | 41 | 1055 | 56 | 6.34 | 30 |
| | DW | TH | 1057 | 83 | 9.32 | 28 | 1048 | 269 | 10.98 | 41 | 1093 | 60 | 8.00 | 30 |
| | FR | TH | 1217 | 86 | 11.85 | 28 | 1160 | 274 | 12.17 | 41 | 1301 | 72 | 10.78 | 30 |
| Themida v3.0.4 | TW | TH | 2811 | 78 | 5.46 | 28 | 2811 | 254 | 6.57 | 41 | 2812 | 51 | 4.01 | 30 |
| | TR | TH | 3611 | 84 | 5.47 | 28 | 3611 | 256 | 6.62 | 41 | 3812 | 54 | 4.15 | 30 |
| | TB | TH | 4411 | 85 | 6.19 | 28 | 4411 | 263 | 7.19 | 41 | 4412 | 60 | 5.37 | 30 |
| | FW | TH | 2411 | 86 | 8.22 | 28 | 2411 | 273 | 8.54 | 41 | 2412 | 69 | 7.92 | 30 |
| | DW | TH | 3611 | 96 | 10.85 | 28 | 3611 | 278 | 11.37 | 41 | 3612 | 80 | 10.78 | 30 |
| | FR | TH | 2811 | 99 | 13.04 | 28 | 2811 | 292 | 13.01 | 41 | 2812 | 93 | 13.86 | 30 |

N/A: not applicable; DD: decode-dispatch; TH: threaded;
TW: tiger white; TR: tiger red; TB: tiger black; FW: fish white; FR: fish red; DW: dalphin white.

**Table 6: Generality of VMForce across diverse virtualization-obfuscation configurations in Enigma and Obsidium.**

| Obfus-cators | Config-uration | Interp-retation | Sample 4 | | | | Sample 5 | | | | Sample 6 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | File Size (KB) | Raw Exe (ms) | Forced Exe (min) | Hidden Syscalls | File Size (KB) | Raw Exe (ms) | Forced Exe (min) | Hidden Syscalls | File Size (KB) | Raw Exe (ms) | Forced Exe (min) | Hidden Syscalls |
| Original | N/A | N/A | 125 | 72 | 4.05 | 28 | 10 | 240 | 5.54 | 41 | 11 | 40 | 3.01 | 30 |
| Enigma v3.10 | DF | TH | 1035 | 78 | 5.11 | 28 | 1042 | 245 | 6.38 | 41 | 1038 | 53 | 4.12 | 30 |
| Enigma v6.70 | DF | TH | 2177 | 80 | 5.49 | 28 | 2260 | 258 | 6.12 | 41 | 2294 | 56 | 4.28 | 30 |
| Obsidium v1.6.7 | DF | TH | 231 | 75 | 4.52 | 28 | 210 | 244 | 5.91 | 41 | 211 | 42 | 3.31 | 30 |
| Obsidium v1.8.5 | DF | TH | 215 | 75 | 4.43 | 28 | 215 | 243 | 5.93 | 41 | 217 | 43 | 3.35 | 30 |

N/A: not applicable; DF: default; TH: threaded.

**Table 7: Details of malware analysis results (ID: 001-050).**

| ID | MD5 | Obfuscator | Version | Cuckoo | | PMP | | VMForce | | VMForce$^{++}$ | | PMP $\cap$ VMForce | PMP $\cup$ VMForce |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 001 | 007a6a5ce873532ecdbfdf17f0b55c30 | VMProtect | 3.2.0-3.5.0 | 2 | × | 32 | ✓ | 33 | ✓ | 33 | ✓ | 32 | 33 |
| 002 | 016ea611120272bc4c954d10cf032a25 | VMProtect | 1.70 | 3 | × | 32 | ✓ | 33 | ✓ | 33 | ✓ | 32 | 33 |
| 003 | 01bb5a323da5fe3a189db37551440c0f | VMProtect | 3.0.0 | 0 | × | 30 | ✓ | 61 | ✓ | 61 | ✓ | 30 | 61 |
| 004 | 01bd9360dcd186a7fbbe985772b1c679 | VMProtect | 2.0.3-2.13 | 0 | × | 15 | ✓ | 23 | ✓ | 23 | ✓ | 15 | 23 |
| 005 | 01fd7eb5249bb24eb1a0b2d2e95a4f69 | VMProtect | 3.2.0-3.5.0 | 1 | × | 18 | ✓ | 19 | ✓ | 21 | ✓ | 12 | 25 |
| 006 | 08e3d3bb8a048e8bd8c5c5d4bfc1c898 | VMProtect | 1.70 | 2 | × | 23 | ✓ | 22 | ✓ | 23 | ✓ | 22 | 23 |
| 007 | 0b501ad94e8d810523a06325d68b6888 | VMProtect | 1.70 | 4 | × | 62 | ✓ | 64 | ✓ | 73 | ✓ | 47 | 79 |
| 008 | 0d09ef2b65110497eef73350b5c2cd81 | VMProtect | 3.0.0 | 0 | × | 11 | ✓ | 12 | ✓ | 12 | ✓ | 11 | 12 |
| 009 | 0e194d35b6a8b8b90452d326d2e05602 | VMProtect | 2.0.3-2.13 | 0 | × | 43 | ✓ | 61 | ✓ | 61 | ✓ | 42 | 62 |
| 010 | 10359ab59cb43715f7ec0043b2c14ab8 | VMProtect | 3.0.0 | 2 | × | 32 | × | 40 | ✓ | 40 | ✓ | 32 | 40 |
| 011 | 10722f53fe48e0f5b977e7f562613055 | VMProtect | 1.70 | 2 | × | 34 | ✓ | 61 | ✓ | 68 | ✓ | 22 | 73 |
| 012 | 13903a7e289cc092e67b748dfea5389c | VMProtect | 3.0.0 | 8 | × | 15 | ✓ | 47 | ✓ | 47 | ✓ | 15 | 47 |
| 013 | 150ec5c8a3120bd4ac9d7bbfa71f006b | VMProtect | 3.2.0-3.5.0 | 0 | × | 33 | × | 61 | ✓ | 69 | ✓ | 22 | 72 |
| 014 | 194430b9ea4e553fe8c01f81ac89f65c | VMProtect | 3.2.0-3.5.0 | 9 | × | 49 | × | 78 | ✓ | 82 | ✓ | 40 | 87 |
| 015 | 1b6fc925045eafaf7b17b02e7f837fac | VMProtect | 3.6+ | 0 | × | 52 | × | 78 | ✓ | 78 | ✓ | 39 | 91 |
| 016 | 1b82ba5b7f783ddd43fe541b99f3b24c | VMProtect | 3.6+ | 4 | × | 46 | ✓ | 64 | ✓ | 66 | ✓ | 39 | 71 |
| 017 | 1c41576f89d8a4c6089080584e97f67a | VMProtect | 3.2.0-3.5.0 | 6 | × | 41 | ✓ | 52 | ✓ | 54 | ✓ | 33 | 60 |
| 018 | 1ce0609b4eeff8f222097f30681083ca | VMProtect | 3.2.0-3.5.0 | 4 | × | 28 | ✓ | 32 | ✓ | 32 | ✓ | 28 | 32 |
| 019 | 1ceaea53a6183669e74848bcc73993bc | VMProtect | 1.70 | 1 | × | 19 | ✓ | 18 | ✓ | 19 | ✓ | 18 | 19 |
| 020 | 1e5f17bd63f56e432440b983358ce988 | VMProtect | 3.2.0-3.5.0 | 0 | × | 15 | ✓ | 22 | ✓ | 22 | ✓ | 15 | 22 |
| 021 | 1fb805dad79925544ffdc23ce5d630aa | VMProtect | 1.70 | 7 | × | 51 | ✓ | 50 | ✓ | 56 | ✓ | 43 | 58 |
| 022 | 23f75f55d35fe425f5bb5b31f435ae95 | VMProtect | 3.2.0-3.5.0 | 0 | × | 41 | × | 51 | ✓ | 51 | ✓ | 39 | 53 |
| 023 | 25692f89edbeb95f2100fc96e9fda94d | VMProtect | 1.70 | 0 | × | 47 | ✓ | 59 | ✓ | 66 | ✓ | 36 | 70 |
| 024 | 26d46ee46e6d8c7f06e735c013515b94 | VMProtect | 2.0.3-2.13 | 1 | × | 39 | ✓ | 46 | ✓ | 46 | ✓ | 36 | 49 |
| 025 | 2886c08535b2f45e200a491914549842 | VMProtect | 2.0.3-2.13 | 2 | × | 29 | ✓ | 33 | ✓ | 33 | ✓ | 29 | 33 |
| 026 | 28ad2ec5fbba6ad51ce3bf7d41dd1ff7 | VMProtect | 3.2.0-3.5.0 | 0 | × | 50 | ✓ | 52 | ✓ | 53 | ✓ | 48 | 54 |
| 027 | 29231a7766201e3d69aa6717fef02bfa | VMProtect | 3.0.0 | 0 | × | 50 | ✓ | 52 | ✓ | 53 | ✓ | 48 | 54 |
| 028 | 2946070c046804062b5a5ffcc8501f3a | VMProtect | 3.0.0 | 0 | × | 15 | × | 23 | ✓ | 23 | ✓ | 15 | 23 |
| 029 | 2abf574f2d557da458c26c201d8f569b | VMProtect | 3.2.0-3.5.0 | 4 | × | 30 | × | 67 | ✓ | 67 | ✓ | 30 | 67 |
| 030 | 2ae29b997cdc82275b2a668cf8028f66 | VMProtect | 3.2.0-3.5.0 | 5 | × | 55 | ✓ | 79 | ✓ | 92 | ✓ | 40 | 94 |
| 031 | 2b4891e9aa1323ce9f4482a3da1845fa | VMProtect | 2.0.3-2.13 | 0 | × | 59 | ✓ | 61 | ✓ | 61 | ✓ | 56 | 64 |
| 032 | 2bbe151302d02a94dcc72a0ba0eb7e7f | VMProtect | 3.2.0-3.5.0 | 4 | × | 50 | × | 53 | ✓ | 53 | ✓ | 48 | 55 |
| 033 | 2c5b07edf14d2ffd8abb8987afb5f90e | VMProtect | 3.2.0-3.5.0 | 1 | × | 11 | × | 19 | ✓ | 19 | ✓ | 11 | 19 |
| 034 | 2c8d6c8b9880036c934d7276de54ef7e | VMProtect | 3.0.0 | 0 | × | 31 | × | 84 | ✓ | 95 | ✓ | 17 | 98 |
| 035 | 31ccb3fc5930455686deff041e7f6085 | VMProtect | 3.2.0-3.5.0 | 1 | × | 49 | ✓ | 50 | ✓ | 51 | ✓ | 44 | 55 |
| 036 | 34e37e3190b66f2553cca83005c9987d | VMProtect | 2.0.3-2.13 | 1 | × | 145 | ✓ | 147 | ✓ | 171 | ✓ | 117 | 175 |
| 037 | 35ee32848a15b517a28b1d765b71a235 | VMProtect | 2.0.3-2.13 | 0 | × | 15 | ✓ | 67 | ✓ | 67 | ✓ | 15 | 67 |
| 038 | 3646ff166750748808dab0c7522eef1b | VMProtect | 2.0.3-2.13 | 4 | × | 32 | ✓ | 33 | ✓ | 33 | ✓ | 32 | 33 |
| 039 | 3656d35d9aa9deaae18208f008ed3b00 | VMProtect | 3.6+ | 1 | × | 51 | ✓ | 50 | ✓ | 51 | ✓ | 48 | 53 |
| 040 | 3776ec0496f11552e5c9766d9eadd0cd | VMProtect | 2.0.3-2.13 | 0 | × | 69 | ✓ | 89 | ✓ | 102 | ✓ | 52 | 106 |
| 041 | 37aa82e8565bb2dbdc5d8f7e77d108a7 | VMProtect | 3.6+ | 5 | × | 50 | × | 64 | ✓ | 67 | ✓ | 38 | 76 |
| 042 | 38eba38191fc023acccffe3e3d2ed5b1 | VMProtect | 3.0.0 | 0 | × | 15 | ✓ | 26 | ✓ | 26 | ✓ | 15 | 26 |
| 043 | 3a23725eee0cc53a809dde4025e823d5 | VMProtect | 1.70 | 0 | × | 54 | × | 73 | ✓ | 79 | ✓ | 40 | 87 |
| 044 | 3af2d3bbce1abcbf89d43f61ceaec7b8 | VMProtect | 3.2.0-3.5.0 | 2 | × | 28 | ✓ | 53 | ✓ | 53 | ✓ | 28 | 53 |
| 045 | 77b4bf2cda957a75c64894a3bd0629fd | VMProtect | 1.70 | 0 | × | 52 | ✓ | 63 | ✓ | 67 | ✓ | 39 | 76 |
| 046 | 7aadc78e849bb393ad3eb6451949bcdf | VMProtect | 3.6+ | 1 | × | 11 | × | 58 | ✓ | 58 | ✓ | 11 | 58 |
| 047 | 7bfd94b73b5879508c7b271d1b53f646 | VMProtect | 3.6+ | 3 | × | 17 | ✓ | 18 | ✓ | 18 | ✓ | 17 | 18 |
| 048 | 7d670cd42d1fb4e6b9057e89ad93fee9 | VMProtect | 2.0.3-2.13 | 0 | × | 11 | ✓ | 19 | ✓ | 19 | ✓ | 11 | 19 |
| 049 | 7f7ecc65cedc20c8bf91e94c9925951a | VMProtect | 3.6+ | 0 | × | 55 | × | 88 | ✓ | 97 | ✓ | 39 | 104 |
| 050 | 815c1431ad60ca15dcfe6a883a448010 | VMProtect | 3.2.0-3.5.0 | 5 | × | 57 | ✓ | 63 | ✓ | 76 | ✓ | 39 | 81 |

**Table 8: Details of malware analysis results (ID: 051-100).**

| ID | MD5 | Obfuscator | Version | Cuckoo | | PMP | | VMForce | | $\text{VMForce}^{++}$ | | PMP ⋂ VMForce | PMP ⋃ VMForce |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 051 | 850885ce04b00306df47f30237f80c4b | VMProtect | 3.2.0-3.5.0 | 7 | × | 42 | × | 61 | ✓ | 61 | ✓ | 41 | 62 |
| 052 | 8f048197675214a5816d55b79d833f71 | VMProtect | 3.6+ | 3 | × | 12 | × | 92 | ✓ | 92 | ✓ | 12 | 92 |
| 053 | 911ef74abf3854db44acaf9493b9662d | VMProtect | 3.2.0-3.5.0 | 1 | × | 16 | × | 24 | ✓ | 24 | ✓ | 16 | 24 |
| 054 | 925d49044708f1321a827b5192a1cfd2 | VMProtect | 2.0.3-2.13 | 5 | × | 33 | × | 63 | ✓ | 63 | ✓ | 33 | 63 |
| 055 | 925e4e855273e65f6dceaabb6b60c95c | VMProtect | 3.0.0 | 1 | × | 11 | ✓ | 19 | ✓ | 19 | ✓ | 11 | 19 |
| 056 | 92b703d08ea9e694fb8d1107b5514cfc | VMProtect | 2.0.3-2.13 | 1 | × | 15 | × | 23 | ✓ | 23 | ✓ | 15 | 23 |
| 057 | 92fa5c32f00645c4291ba4041ad034c5 | VMProtect | 1.70 | 1 | × | 11 | ✓ | 23 | ✓ | 23 | ✓ | 11 | 23 |
| 058 | 9495e0aa55950cfa58ea1145bbacade7 | VMProtect | 2.0.3-2.13 | 0 | × | 25 | × | 36 | ✓ | 36 | ✓ | 23 | 38 |
| 059 | 98a46b97ba8c347b7a1966c2e70e4872 | VMProtect | 2.0.3-2.13 | 0 | × | 22 | ✓ | 21 | ✓ | 22 | ✓ | 21 | 22 |
| 060 | 99501cd0ca0a50a7ba03624fd54b3bc3 | VMProtect | 1.70 | 5 | × | 93 | × | 126 | ✓ | 149 | ✓ | 67 | 152 |
| 061 | 9b309a2cfb89ba712acb1d7091786e8b | VMProtect | 1.70 | 3 | × | 44 | × | 50 | ✓ | 50 | ✓ | 36 | 58 |
| 062 | 9dcea5793a3a1d6643f6c69a81c0026f | VMProtect | 3.0.0 | 3 | × | 12 | × | 45 | ✓ | 45 | ✓ | 12 | 45 |
| 063 | aa47250ec4a8a95c07aa55456af9172d | VMProtect | 3.2.0-3.5.0 | 3 | × | 55 | × | 74 | ✓ | 76 | ✓ | 39 | 90 |
| 064 | ab24e0d36827095689462634470ebb92 | VMProtect | 1.70 | 3 | × | 50 | ✓ | 55 | ✓ | 61 | ✓ | 39 | 66 |
| 065 | ab4d8d321b135bcd4f0a1aa7b099b253 | VMProtect | 3.0.0 | 1 | × | 19 | ✓ | 18 | ✓ | 19 | ✓ | 18 | 19 |
| 066 | f0bf41fb52b4ee8515033cdeb2dc1881 | VMProtect | 2.0.3-2.13 | 2 | × | 52 | × | 78 | ✓ | 78 | ✓ | 52 | 78 |
| 067 | ac02551f3fc6e6e4b648d9696ea96083 | VMProtect | 3.0.0 | 5 | × | 51 | ✓ | 76 | ✓ | 84 | ✓ | 39 | 88 |
| 068 | ac09aa6fd52ab4414e8a43e4288a93f8 | VMProtect | 3.2.0-3.5.0 | 0 | × | 15 | ✓ | 26 | ✓ | 26 | ✓ | 15 | 26 |
| 069 | ac3237a732299e68388afc42c0244660 | VMProtect | 3.0.0 | 1 | × | 43 | × | 52 | ✓ | 53 | ✓ | 41 | 54 |
| 070 | ad49118907c1cd509f5b2f6457e736af | VMProtect | 3.2.0-3.5.0 | 0 | × | 30 | ✓ | 69 | ✓ | 69 | ✓ | 30 | 69 |
| 071 | ae691bcac6ec9acce2aefaf2e3046e57 | VMProtect | 3.0.0 | 0 | × | 54 | ✓ | 70 | ✓ | 73 | ✓ | 38 | 86 |
| 072 | af5c4941d0cda259866ecca382c7e4c7 | VMProtect | 3.6+ | 3 | × | 32 | ✓ | 33 | ✓ | 33 | ✓ | 32 | 33 |
| 073 | af72055eb2030e50a4c47936766d4d95 | VMProtect | 3.2.0-3.5.0 | 7 | × | 11 | × | 19 | ✓ | 19 | ✓ | 11 | 19 |
| 074 | afbac493b4a447273008483d740b8ef1 | VMProtect | 3.0.0 | 5 | × | 53 | × | 69 | ✓ | 78 | ✓ | 39 | 83 |
| 075 | afd06ba223b75090ae2d964ddf550b36 | VMProtect | 3.0.0 | 0 | × | 44 | × | 76 | ✓ | 77 | ✓ | 41 | 79 |
| 076 | b12bfbcd7225c1fc858b16340ddc603a | VMProtect | 3.2.0-3.5.0 | 4 | × | 11 | ✓ | 25 | ✓ | 25 | ✓ | 11 | 25 |
| 077 | b18b92f505839c475432d14049c4f0f8 | VMProtect | 1.70 | 3 | × | 35 | ✓ | 36 | ✓ | 36 | ✓ | 34 | 37 |
| 078 | b1fd758c899da19405908b6c91580a55 | VMProtect | 1.70 | 0 | × | 60 | × | 145 | ✓ | 148 | ✓ | 56 | 149 |
| 079 | b286a1df6ee124a2d9e50f00d194abe0 | VMProtect | 3.2.0-3.5.0 | 0 | × | 55 | × | 70 | ✓ | 75 | ✓ | 38 | 87 |
| 080 | b33c923fabf63524d89a61d3d0bd0502 | VMProtect | 2.0.3-2.13 | 4 | × | 16 | ✓ | 58 | ✓ | 58 | ✓ | 16 | 58 |
| 081 | b471fb4c82d29daefdf9cf75194f69d5 | VMProtect | 3.0.0 | 4 | × | 35 | ✓ | 63 | ✓ | 64 | ✓ | 33 | 65 |
| 082 | b48c1fcd849816cc10355a0ada672d74 | VMProtect | 3.0.0 | 0 | × | 16 | × | 106 | ✓ | 106 | ✓ | 16 | 106 |
| 083 | b4dc166e2664f6274f5c3c00fa30be20 | VMProtect | 3.2.0-3.5.0 | 7 | × | 42 | × | 53 | ✓ | 53 | ✓ | 41 | 54 |
| 084 | b6d1db345f9d12522bf12f82be3a5da7 | VMProtect | 2.0.3-2.13 | 0 | × | 28 | ✓ | 32 | ✓ | 32 | ✓ | 28 | 32 |
| 085 | b82c5d273cd0da5ebc30150ed1ede68b | VMProtect | 3.0.0 | 4 | × | 34 | × | 63 | ✓ | 63 | ✓ | 33 | 64 |
| 086 | b8be805a5d7a62e23fd5833d828e4295 | VMProtect | 3.0.0 | 4 | × | 31 | ✓ | 32 | ✓ | 32 | ✓ | 31 | 32 |
| 087 | b90ef94ac29b2b575a6959f2f694e317 | VMProtect | 1.70 | 0 | × | 42 | ✓ | 53 | ✓ | 53 | ✓ | 41 | 54 |
| 088 | b9276c9b566f6a04ca6724c9b7ea214c | VMProtect | 3.0.0 | 0 | × | 15 | × | 20 | ✓ | 20 | ✓ | 15 | 20 |
| 089 | bd5e34c4c35580a839cef03a4f799b18 | VMProtect | 3.2.0-3.5.0 | 5 | × | 53 | × | 80 | ✓ | 91 | ✓ | 39 | 94 |
| 090 | bedc464e0d43051e2f200e087e196720 | VMProtect | 3.2.0-3.5.0 | 9 | × | 50 | × | 79 | ✓ | 84 | ✓ | 40 | 89 |
| 091 | bf73501ca53ed763080c167a94ace97c | VMProtect | 1.70 | 0 | × | 85 | × | 88 | ✓ | 112 | ✓ | 44 | 129 |
| 092 | c106f9bd9dd44c44fb9cfb6bc04928ad | VMProtect | 1.70 | 3 | × | 37 | ✓ | 38 | ✓ | 38 | ✓ | 36 | 39 |
| 093 | c18678648e0a4f96bbb7d507e95c3c9e | VMProtect | 3.2.0-3.5.0 | 5 | × | 42 | × | 53 | ✓ | 53 | ✓ | 41 | 54 |
| 094 | c2c4a781871afd02f2aff735390be9f4 | VMProtect | 3.2.0-3.5.0 | 0 | × | 32 | ✓ | 33 | ✓ | 33 | ✓ | 32 | 33 |
| 095 | c52152b43b5df1876c76dc513a2bf6c9 | VMProtect | 2.0.3-2.13 | 0 | × | 34 | × | 63 | ✓ | 63 | ✓ | 33 | 64 |
| 096 | c675d14923701cee53e63319f101bfa0 | VMProtect | 3.0.0 | 0 | × | 36 | × | 71 | ✓ | 72 | ✓ | 33 | 74 |
| 097 | c6a6c52b45d6306e244a6242906df7f2 | VMProtect | 3.2.0-3.5.0 | 9 | × | 59 | ✓ | 74 | ✓ | 93 | ✓ | 39 | 94 |
| 098 | c858c16f574d5a3cc670f36bf59d5b7b | VMProtect | 2.0.3-2.13 | 6 | × | 41 | ✓ | 40 | ✓ | 41 | ✓ | 39 | 42 |
| 099 | c97ba4424299730db2909d9d402dec59 | VMProtect | 3.0.0 | 4 | × | 56 | × | 68 | ✓ | 68 | ✓ | 39 | 85 |
| 100 | cb2ae2f72ec3cdc867888214ef53fd28 | VMProtect | 3.6+ | 4 | × | 44 | ✓ | 64 | ✓ | 65 | ✓ | 38 | 70 |

**Table 9: Details of malware analysis results (ID: 101-150).**

| ID | MD5 | Obfuscator | Version | Cuckoo | | PMP | | VMForce | | VMForce$^{++}$ | | PMP ⋂ VMForce | PMP ⋃ VMForce |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 101 | 0d3189772138415a64187e43356de58f | Themida | 3.XX | 0 | × | 34 | × | 120 | ✓ | 130 | ✓ | 17 | 137 |
| 102 | 9f8deaea62bfbe19feba37b4603b56b5 | Themida | 3.XX | 0 | × | 30 | × | 117 | ✓ | 128 | ✓ | 14 | 133 |
| 103 | c80c47aabfee0cf388b986b7fc1d5754 | Themida | 1.XX-2.XX | 0 | × | 21 | × | 124 | ✓ | 127 | ✓ | 13 | 132 |
| 104 | b981e93c1a9009de944755e8d8fec27a | Themida | 1.XX-2.XX | 0 | × | 14 | ✓ | 118 | ✓ | 118 | ✓ | 10 | 122 |
| 105 | 4119e6d0b20031d861fda3e16a765acf | Themida | 3.XX | 33 | ✓ | 34 | ✓ | 121 | ✓ | 126 | ✓ | 18 | 137 |
| 106 | 60cc75f9b4c061ade6b0d72deb56efb4 | Themida | 3.XX | 0 | × | 31 | ✓ | 116 | ✓ | 129 | ✓ | 15 | 132 |
| 107 | d70516aa97f6d7d77adb33c2615165a0 | Themida | 1.XX-2.XX | 24 | × | 24 | × | 41 | ✓ | 43 | ✓ | 20 | 45 |
| 108 | 7d6be7fb3620160997ccec96672d5a0e | Themida | 3.XX | 33 | ✓ | 34 | ✓ | 118 | ✓ | 132 | ✓ | 18 | 134 |
| 109 | d6d21d241af79e6bf41b497e072822a9 | Themida | 3.XX | 0 | × | 31 | ✓ | 120 | ✓ | 134 | ✓ | 15 | 136 |
| 110 | e4d300c2db92bf3c69273b952342c0c1 | Themida | 3.XX | 33 | × | 34 | × | 123 | ✓ | 125 | ✓ | 18 | 139 |
| 111 | bf901ee58a26cbdeeee469012944558b | Themida | 3.XX | 0 | × | 31 | ✓ | 120 | ✓ | 120 | ✓ | 15 | 136 |
| 112 | d7eb0914792b467a115accf63a58d8be | Themida | 3.XX | 33 | × | 34 | × | 118 | ✓ | 124 | ✓ | 18 | 134 |
| 113 | 05d0fe05506a86ba4c8c5d44a82ed372 | Themida | 3.XX | 33 | × | 34 | × | 120 | ✓ | 121 | ✓ | 18 | 136 |
| 114 | 06c572150fad97087926d2927e8c999a | Themida | 3.XX | 29 | × | 38 | × | 119 | ✓ | 125 | ✓ | 18 | 139 |
| 115 | 086bb5bfb09825c3b0f1822a04c1384e | Themida | 3.XX | 0 | × | 31 | × | 118 | ✓ | 120 | ✓ | 15 | 134 |
| 116 | 102322223922116321c49df10ea50978 | Themida | 3.XX | 10 | × | 29 | ✓ | 119 | ✓ | 133 | ✓ | 14 | 134 |
| 117 | 10a3bc719f4991dd390597b84c7c1883 | Themida | 3.XX | 45 | × | 48 | × | 120 | ✓ | 124 | ✓ | 20 | 148 |
| 118 | 1454fcbbfaf2cc099c6de24c68de79b9 | Themida | 3.XX | 0 | × | 11 | × | 121 | ✓ | 121 | ✓ | 10 | 122 |
| 119 | 259ba80f7c7037230b07879101b44b8f | Themida | 3.XX | 25 | × | 28 | × | 119 | ✓ | 122 | ✓ | 14 | 133 |
| 120 | 25d9e5af9d94a123747eacefe9d0fcfe | Themida | 3.XX | 0 | × | 33 | × | 116 | ✓ | 124 | ✓ | 16 | 133 |
| 121 | 2627a0f277095c747318bd068af03149 | Themida | 3.XX | 0 | × | 30 | × | 108 | ✓ | 115 | ✓ | 14 | 124 |
| 122 | 2851781e019ae7be34311f783aafc385 | Themida | 3.XX | 0 | × | 28 | × | 112 | ✓ | 124 | ✓ | 14 | 126 |
| 123 | 2c35ee0ffe366f82afb19bed2e9e22e7 | Themida | 3.XX | 45 | ✓ | 48 | ✓ | 117 | ✓ | 123 | ✓ | 20 | 145 |
| 124 | 352542db3dafc44c8b8f30adfde7c0e7 | Themida | 3.XX | 0 | × | 11 | × | 113 | ✓ | 113 | ✓ | 10 | 114 |
| 125 | 3aeae604ef7949fc20b3b62d9662a86b | Themida | 3.XX | 0 | × | 31 | × | 117 | ✓ | 123 | ✓ | 15 | 133 |
| 126 | 4938d606cbef03a47a8b5994d8cc3855 | Themida | 3.XX | 45 | ✓ | 48 | ✓ | 119 | ✓ | 125 | ✓ | 20 | 147 |
| 127 | 49420255ebba47e2e2cc77b549229d4c | Themida | 3.XX | 0 | × | 33 | × | 117 | ✓ | 120 | ✓ | 18 | 132 |
| 128 | 4e8662569e17c873a532fb94c543fdbc | Themida | 3.XX | 0 | × | 31 | × | 118 | ✓ | 118 | ✓ | 16 | 133 |
| 129 | 4fe855e2505311325f987ffd25df4cb0 | Themida | 3.XX | 2 | × | 49 | × | 112 | ✓ | 115 | ✓ | 21 | 140 |
| 130 | 503720ef3f7ff131ccd8d1e1f1f3e086 | Themida | 1.XX-2.XX | 1 | × | 25 | × | 121 | ✓ | 125 | ✓ | 14 | 132 |
| 131 | 5d702f1c807b600a88c7fc9e956cf1f6 | Themida | 3.XX | 0 | × | 31 | × | 116 | ✓ | 118 | ✓ | 15 | 132 |
| 132 | 63fa78dcd8f4c96f97f37810a780e4ec | Themida | 3.XX | 0 | × | 11 | × | 109 | ✓ | 109 | ✓ | 10 | 110 |
| 133 | 64c923df50d20a648f3c8682171d13fa | Themida | 3.XX | 0 | × | 12 | × | 119 | ✓ | 119 | ✓ | 11 | 120 |
| 134 | 665515a1e8ea6c66a12bec7b82540e2d | Themida | 3.XX | 0 | × | 29 | × | 120 | ✓ | 126 | ✓ | 15 | 134 |
| 135 | 6adbf865a38262ca2e253ae76a45585a | Themida | 3.XX | 33 | ✓ | 34 | ✓ | 120 | ✓ | 134 | ✓ | 18 | 136 |
| 136 | 6c9b7ea4eea5a309c2792174f00bcda0 | Themida | 3.XX | 27 | × | 38 | × | 132 | ✓ | 148 | ✓ | 19 | 151 |
| 137 | 7b4f25a2b6604ee4bf429f469e213621 | Themida | 3.XX | 45 | ✓ | 49 | ✓ | 115 | ✓ | 130 | ✓ | 21 | 143 |
| 138 | 7b77e8f23f9764bff3a2bfd3e59253ca | Themida | 3.XX | 0 | × | 31 | ✓ | 116 | ✓ | 125 | ✓ | 15 | 132 |
| 139 | 84feb1d8fb46542749220f7f9e80adcf | Themida | 3.XX | 0 | × | 29 | × | 119 | ✓ | 124 | ✓ | 15 | 133 |
| 140 | 989f720056d4f507004a41c4a6ad5247 | Themida | 3.XX | 0 | × | 12 | × | 118 | ✓ | 118 | ✓ | 11 | 119 |
| 141 | abca382dd1a0e8eb510262cb937fca1f | Themida | 1.XX-2.XX | 20 | × | 24 | × | 114 | ✓ | 121 | ✓ | 13 | 125 |
| 142 | b0795cd046781763d65a8839fc8f4eef | Themida | 3.XX | 33 | ✓ | 34 | ✓ | 119 | ✓ | 121 | ✓ | 18 | 135 |
| 143 | b2789284ea3e11d1fb340d901cb65d41 | Themida | 3.XX | 33 | ✓ | 34 | ✓ | 119 | ✓ | 126 | ✓ | 18 | 135 |
| 144 | b6281009ddc4cbf4f69bcde4f4f87b46 | Themida | 3.XX | 0 | × | 31 | × | 115 | ✓ | 124 | ✓ | 15 | 131 |
| 145 | b9f4e75acf2105de00dda971c9fffcf6 | Themida | 3.XX | 45 | ✓ | 49 | ✓ | 121 | ✓ | 133 | ✓ | 21 | 149 |
| 146 | bac90a44deaf1d41492803b7363fc9ab | Themida | 3.XX | 35 | ✓ | 37 | ✓ | 122 | ✓ | 125 | ✓ | 20 | 139 |
| 147 | be91e1a0fecfcf43b874c0f408c2378c | Themida | 3.XX | 33 | ✓ | 34 | ✓ | 125 | ✓ | 135 | ✓ | 18 | 141 |
| 148 | c08dcf94bd60086c9f0377f1caeefc3b | Themida | 3.XX | 32 | × | 42 | ✓ | 121 | ✓ | 142 | ✓ | 18 | 145 |
| 149 | c0d087a6d0d7d40f37077751a21c3513 | Themida | 3.XX | 45 | ✓ | 48 | ✓ | 118 | ✓ | 138 | ✓ | 20 | 146 |
| 150 | 6663b47ea4cab4b13eaf4f10c11c4267 | Themida | 3.XX | 0 | × | 31 | × | 118 | ✓ | 125 | ✓ | 15 | 134 |

**Table 10: Details of malware analysis results (ID: 151-200).**

| ID | MD5 | Obfuscator | Version | Cuckoo | | PMP | | VMForce | | VMForce$^{++}$ | | PMP $\cap$ VMForce | PMP $\cup$ VMForce |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 151 | 00d35409e99b0f59730af9527d9bd998 | Enigma | 5.X | 10 | × | 70 | ✓ | 82 | ✓ | 88 | ✓ | 64 | 88 |
| 152 | 0ebc3c08d0e3b7f973466d55394119ac | Enigma | 5.X | 5 | × | 44 | ✓ | 46 | ✓ | 47 | ✓ | 43 | 47 |
| 153 | 19bb3daa7a8a0511547e957b3a68745d | Enigma | 4.0 | 0 | × | 23 | ✓ | 27 | ✓ | 34 | ✓ | 16 | 34 |
| 154 | 1f13ec3d0b944946d8eadd1cc661cf6e | Enigma | Virtual Box | 6 | × | 41 | × | 54 | ✓ | 54 | ✓ | 41 | 54 |
| 155 | 2ac67685348947f76ee39fe813f780f1 | Enigma | 4.0 | 4 | × | 47 | ✓ | 48 | ✓ | 48 | ✓ | 46 | 49 |
| 156 | 3a9e9651f7ba85642cfcb04b1a8b018f | Enigma | 5.X | 8 | × | 39 | ✓ | 55 | ✓ | 55 | ✓ | 36 | 58 |
| 157 | 3c62ff7a257c922ba29d491b1813d4cc | Enigma | 5.X | 9 | × | 51 | × | 75 | ✓ | 76 | ✓ | 49 | 77 |
| 158 | 3ca398e27b93a3af695885eccd329239 | Enigma | 4.0 | 11 | × | 57 | × | 76 | ✓ | 79 | ✓ | 53 | 80 |
| 159 | 3cd3b98273c5895a80bf040acbb36fa6 | Enigma | 5.X | 5 | × | 66 | × | 71 | ✓ | 72 | ✓ | 65 | 72 |
| 160 | 3ceb847f1e5fc375e898104e278cebec | Enigma | 5.X | 5 | × | 39 | × | 60 | ✓ | 61 | ✓ | 37 | 62 |
| 161 | 45316370c4acf18728aa5363cb7ca861 | Enigma | 5.X | 11 | × | 15 | ✓ | 26 | ✓ | 26 | ✓ | 11 | 30 |
| 162 | 4553b8cd3e1cdf2254a40a03e045ea9d | Enigma | 5.X | 11 | × | 53 | × | 59 | ✓ | 59 | ✓ | 48 | 64 |
| 163 | 4569edbd7c219bf06c4e9b4f14a58fd7 | Enigma | 4.0 | 2 | × | 26 | ✓ | 34 | ✓ | 38 | ✓ | 19 | 41 |
| 164 | 589e3d6bf6a906c4717795dcf4e17ec0 | Enigma | 5.X | 2 | × | 49 | × | 60 | ✓ | 65 | ✓ | 44 | 65 |
| 165 | 689df0c5835f2939153b330d312658ed | Enigma | 4.0 | 0 | × | 22 | × | 41 | ✓ | 42 | ✓ | 15 | 48 |
| 166 | 874dad172d7d4af0bab54a7a5b7dc6a7 | Enigma | 4.0 | 0 | × | 22 | × | 52 | ✓ | 57 | ✓ | 15 | 59 |
| 167 | 993fa24ed6cd9a76f7356bad0e87fbef | Enigma | 4.0 | 10 | × | 38 | ✓ | 64 | ✓ | 64 | ✓ | 34 | 68 |
| 168 | 9b2b2b8bff06f1758f397e52e979a79b | Enigma | 4.0 | 8 | × | 26 | × | 45 | ✓ | 46 | ✓ | 19 | 52 |
| 169 | 9f589dc2fb40c7b30cc45163ef68cef7 | Enigma | 5.X | 0 | × | 44 | × | 53 | ✓ | 54 | ✓ | 43 | 54 |
| 170 | a637f8dcd7b014468e294f31782b3f34 | Enigma | 5.X | 6 | × | 25 | ✓ | 41 | ✓ | 42 | ✓ | 21 | 45 |
| 171 | b11ea086b15fdd2697ed30209ff95f33 | Enigma | 5.X | 0 | × | 24 | × | 36 | ✓ | 36 | ✓ | 24 | 36 |
| 172 | b56487475b5d59ffbf868cd3a45e8c68 | Enigma | 4.0 | 8 | × | 62 | ✓ | 63 | ✓ | 64 | ✓ | 61 | 64 |
| 173 | ba506d12334610298fcae906b1231b12 | Enigma | 5.X | 6 | × | 60 | ✓ | 73 | ✓ | 76 | ✓ | 54 | 79 |
| 174 | bd1b3b496b17b6ebbc19483dc8fd2b38 | Enigma | 4.0 | 0 | × | 29 | ✓ | 31 | ✓ | 31 | ✓ | 25 | 35 |
| 175 | c108a99104488f06bd03b9f18c58b4cf | Enigma | 4.0 | 10 | × | 45 | ✓ | 66 | ✓ | 67 | ✓ | 44 | 67 |
| 176 | c7f172516b62568cb53cf150372fb5fd | Enigma | 4.0 | 11 | × | 60 | ✓ | 75 | ✓ | 78 | ✓ | 57 | 78 |
| 177 | dffd3b096c51703ba0507a6205927073 | Enigma | 5.X | 6 | × | 29 | ✓ | 48 | ✓ | 48 | ✓ | 25 | 52 |
| 178 | e4c3f4417160312aa250e48270aaeb1a | Enigma | 4.0 | 8 | × | 20 | × | 38 | ✓ | 39 | ✓ | 19 | 39 |
| 179 | fe42412daa37113977a4da70f837b601 | Enigma | 2.0 | 10 | × | 63 | ✓ | 80 | ✓ | 80 | ✓ | 58 | 85 |
| 180 | feb75eba686cf3be01048adf58acde9b | Enigma | 5.X | 10 | × | 50 | ✓ | 53 | ✓ | 55 | ✓ | 48 | 55 |
| 181 | 0723a4223cb36fd9140ac5740421e143 | Obsidium | v1.4.X.X | 9 | × | 20 | × | 50 | ✓ | 51 | ✓ | 19 | 51 |
| 182 | 0c7d6b7702dc64daf64dad4bd952e4d4 | Obsidium | v1.4.X.X | 2 | × | 60 | ✓ | 86 | ✓ | 86 | ✓ | 53 | 93 |
| 183 | 1ea855fa125549b8ab1b9a2af86db88d | Obsidium | unknown 2 | 0 | × | 17 | × | 24 | ✓ | 24 | ✓ | 17 | 24 |
| 184 | 209eefe2b664041baa0295e342f2e85a | Obsidium | unknown 2 | 0 | × | 60 | ✓ | 63 | ✓ | 66 | ✓ | 57 | 66 |
| 185 | 2e10e8c4d30a08273420da7674b9418b | Obsidium | 1.4.2.0 | 5 | × | 38 | × | 63 | ✓ | 63 | ✓ | 36 | 65 |
| 186 | 40e45651b4aaf5f206c836868450a0ff | Obsidium | unknown 3 | 11 | × | 21 | ✓ | 22 | ✓ | 23 | ✓ | 17 | 26 |
| 187 | 46addeb1ae0884580955efc7998bbb7d | Obsidium | 1.4.2.0 | 4 | × | 61 | × | 70 | ✓ | 75 | ✓ | 56 | 75 |
| 188 | 6ff6d0e450ecd38bd82e6da4e3a6305d | Obsidium | v1.4.X.X | 10 | × | 62 | ✓ | 67 | ✓ | 69 | ✓ | 55 | 74 |
| 189 | 7d041e9fc3b011856ff240c2ee5649ae | Obsidium | unknown 3 | 6 | × | 70 | ✓ | 74 | ✓ | 75 | ✓ | 69 | 75 |
| 190 | 7e58dde124aa4ad5df544d56b2bdcc6d | Obsidium | unknown 2 | 7 | × | 56 | ✓ | 73 | ✓ | 75 | ✓ | 51 | 78 |
| 191 | 85abf44b821263cbf5cfcf8624333490 | Obsidium | v1.4.X.X | 0 | × | 62 | × | 71 | ✓ | 71 | ✓ | 62 | 71 |
| 192 | 8647f0f58e6d152be16b810a094b820b | Obsidium | unknown 2 | 0 | × | 63 | × | 72 | ✓ | 72 | ✓ | 59 | 76 |
| 193 | 89de83408953a608c6ca1dbcf3f749de | Obsidium | v1.4.X.X | 0 | × | 42 | ✓ | 44 | ✓ | 45 | ✓ | 41 | 45 |
| 194 | 937ca4be079785d747cf51a99c0d6c69 | Obsidium | unknown 3 | 9 | × | 36 | × | 63 | ✓ | 66 | ✓ | 32 | 67 |
| 195 | 939d53583c7690c008a8e95551b8d22b | Obsidium | 1.4.2.0 | 7 | × | 33 | ✓ | 61 | ✓ | 62 | ✓ | 32 | 62 |
| 196 | ab0c32b1492c5497a901d36b16057ebd | Obsidium | v1.4.X.X | 10 | × | 49 | ✓ | 76 | ✓ | 76 | ✓ | 47 | 78 |
| 197 | afc36babf59c7ba4d118c6d9dc05c0a7 | Obsidium | v1.4.X.X | 0 | × | 18 | ✓ | 45 | ✓ | 45 | ✓ | 18 | 45 |
| 198 | f8dc241cca1cb976e3d5c72e04b9d391 | Obsidium | unknown 3 | 11 | × | 63 | × | 89 | ✓ | 89 | ✓ | 63 | 89 |
| 199 | fd714d4760bef0dacb54cf3ce0718fdf | Obsidium | v1.4.X.X | 5 | × | 42 | ✓ | 49 | ✓ | 49 | ✓ | 37 | 54 |
| 200 | fef7d367ec927cfeb2a244595bc5ef18 | Obsidium | 1.4.2.0 | 0 | × | 15 | × | 29 | ✓ | 30 | ✓ | 9 | 35 |