VMAttack: Deobfuscating Virtualization-Based Packed Binaries

Anatoli Kalysch
FAU Erlangen-Nuremberg
anatoli.kalysch@fau.de

Johannes Götzfried
FAU Erlangen-Nuremberg
johannes.goetzfried@cs.fau.de

Tilo Müller
FAU Erlangen-Nuremberg
tilo.mueller@cs.fau.de

ABSTRACT

We present VMAttack, a deobfuscation tool for virtualization-packed binaries based on automated static and dynamic analysis, which offers a simplified view of the disassembly. VMAttack is implemented as a plug-in for IDA Pro and as such, integrates seamlessly with manual reverse engineering. The complexity of the disassembly view is notably reduced by analyzing the inner working principles of the VM layer of protected binaries. Using static analysis, complex bytecode sequences of the VM are mapped to easy-to-read pseudo-code instructions, based on an intermediate representation specifically designed for stack-based virtual machines. Using dynamic analysis, we identify structural components like the interpreter loop and compress instruction sequences by filtering out semantically redundant instructions of the execution trace. The integrated result, which rates both static and dynamic analysis's results, provides the reverse engineer with a deobfuscated disassembly that tolerates weaknesses of a single analysis technique. VMAttack is currently limited to stack-based virtual machines like VMProtect. We evaluated VMAttack using binaries obfuscated with VMProtect and achieved an average execution trace reduction of 89.86% for the dynamic and 96.67% for the combined static and dynamic analysis.

KEYWORDS

Deobfuscation, Virtualization-based Obfuscation, Dynamic Analysis, Static Analysis, Reverse Engineering

ACM Reference format:
Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. 2017. VMAttack: Deobfuscating Virtualization-Based Packed Binaries. In *Proceedings of ARES '17, Reggio Calabria, Italy, August 29-September 01, 2017,* 10 pages.
DOI: 10.1145/3098954.3098995

1 INTRODUCTION

Besides dynamic protection measures, like anti-debugging and hook detection, static code obfuscation is the method of choice to increase the effort that a reverse engineer has to invest to analyze an application. Amongst static code obfuscation techniques, virtualization-based obfuscation has proven to be a particular effective measure. It is used, for example, by professional DRM solutions like the Blu-ray BD+ system [1]. However, the same technique also got the attention of malware authors [16, 20], motivating the demand for unpackers that roll back virtualization-based obfuscation in the AV industry.

Soon after most obfuscation approaches became public, automatic deobfuscation tools were proposed that could recover the original functionality [8, 13, 18]. For virtualization-based obfuscation, however, there is still a lack of reliable deobfuscation tools. Virtualization-based obfuscation works by transforming the original functionality of a program to bytecode for a randomly generated virtual machine. As the VM and the bytecode language are generated randomly, an application can easily be repackaged with a different signature by just re-obfuscating it. Although the original functionality remains unchanged, applications outwardly appear completely different as the protection layer changes [15].

Also internally, virtualization-based obfuscation changes the control flow in a way that known reverse engineering techniques are rendered hard or impossible to apply [14]. Virtualization-packed binaries do not restore their original code at any point in time during execution [9, 15]. Instead, a VM interprets equivalent bytecode instructions leading to an enormous increase in executed instructions, and causing the original representation of a binary to vanish. Additionally, commercial obfuscators like VMProtect [29] and Themida [17] can introduce several layers of VMs, where outer VMs interpret inner VMs, which in turn interpret the payload [10].

1.1 Our Contribution

Deobfuscation approaches for virtualization-packed binaries have long been part of state-of-the-art research, for example, including techniques based on static, dynamic, and concolic execution analysis [14, 19, 30]. Nevertheless, up to today, a universal deobfuscation approach for virtualization-packed binaries has not yet been found. With the design of VMAttack, we contribute to the evolution of deobfuscation tools against VM-based obfuscation. Our implementation allows for automatic and semi-automatic analysis, and provides reverse engineers with a simplified view of the disassembly by seamlessly integrating it as a plug-in into the IDA Pro framework. In detail, our contributions are:

- The static analysis approach of VMAttack transforms virtualized bytecode into pseudo-code instructions of

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.
ARES '17, Reggio Calabria, Italy
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 978-1-4503-5257-4/17/08...$15.00
DOI: 10.1145/3098954.3098995

an intermediate representation specifically targeting stack-based VMs.

- VMAttack's dynamic analysis approach records execution traces of packed binaries and optimizes those by filtering out instructions belonging solely to the VM interpreter logic
- We evaluated VMAttack using binaries obfuscated with VMProtect [29]. We were able to fully recover the functionality with the help of static analysis and could reduce the execution trace by 89.86% on average with the dynamic analysis alone. We achieved an average instruction trace reduction of 96.67% for the combined analysis approach.
- We developed VMAttack as an open source plug-in for the disassembler IDA Pro. VMAttack has been published under the MIT license and is freely available at https://github.com/anatolikalysch/VMAttack.

1.2 Related Work

Traditional approaches for deobfuscation use either static [12, 22] or dynamic [19] analysis, or a combination of both [20], but recently, also techniques based on *symbolic* and *concolic* execution [5, 24, 30] became relevant, like Driller [28] and Angr [25].

A straightforward approach is to deobfuscate the VM interpreter, enabling reverse engineers to convert the VM's bytecode back into x86 assembly instructions by utilizing an intermediate representation [22]. The reversal of the whole interpreter, however, is not always necessary, as Guillot et al. have shown [12] by an approach for reversing a VM and using symbolic execution to simplify the instruction set into a symbolic one. The downside of the approaches from Guillot et al. and Rolles is the immense time consumption and lack of automation.

Dynamic approaches execute the binary at least once and use the resulting execution trace for analysis. For example, the execution trace could be clustered into repeating instructions. Raber [19] applies a post-clustering optimization, but this optimization has to be steadily updated by the reverse engineer to remain precise. *Taint-tracking* and *symbolic execution* are popular techniques to either generate an equivalent but simpler control flow graph [5, 30] or transform the instructions of the trace into an IR and use clustering to generate a mapping of the bytecode to IR instructions.

Taint analysis executes a program and observes which computations are affected by predefined taint sources such as user input. The approaches of Coogan et al. [5] and Yadegari et al. [30] use taint tracking of predefined values to draw conclusions about the inner workings of the binary. Coogan et al. [5] reason that the interaction of the binary with its environment in form of system calls is most important and therefore taint system call values and trace them back throughout the execution. Yadegari et al. [30] taint the input and output values focusing on possible transformations between input and output values. Based on these transformations, they determine which inputs might lead to differing outputs from the virtual machine function.

Concolic solutions have their own drawbacks. Taint tracking-based solution require an IR transformation of the binary instructions and produce a computational overhead by design, due to the tracking and simulation of memory operations. As pointed out by Yadegari et al. and Coogan, a transformation into an IR is required for the code optimizations [6, 30].

An issue of symbolic execution is the handling of path explosions. Depending on the symbolic variables, several additional paths need to be computed subsequently. This results in an increased number of computations [30]. Hence, our static and dynamic analysis approaches present a more feasible fit for the task of deobfuscating VM-based packed binaries, due to the lower requirements towards computational performance and the improved parallelization possibilities. Subsequently, a combination of purely dynamic and static analysis, without symbolic execution or taint tracking, is the focus of this paper.

1.3 Outline

The remainder of this paper is structured as follows: Section 2 provides background information regarding virtualization obfuscation in general. Section 3 presents the design and implementation of VMAttack, detailing the architecture and analysis capabilities, in particular the static and dynamic analysis, and the combined system. Our implementation is then evaluated (Section 4) and open questions are discussed (Section 5). Finally, in Section 6, we draw conclusions about our work.

2 BACKGROUND

This section describes the building blocks necessary to understand the design and implementation of VMAttack. Readers familiar with process virtual machines (Section 2.1) and virtualization obfuscators (Section 2.2) may safely skip this section.

2.1 Process Virtual Machines

The most common virtual machine category used for obfuscation are *Process Virtual Machines* (PVMs), also referred to as application virtual machines [27]. Process virtual machines run on top of the operating system, i.e., on application level and provide a platform-independent programming environment that abstracts away details of the underlying hardware or operating system allowing a program to be executed unmodified on any platform [7].

One subcategory of PVMs, which is often used for obfuscation, are the *Emulating Virtual Machines* (EVMs). EVMs do not require high-level language code being translated into bytecode, but instead are compiled directly into specific machine code for a specific platform [7, 27]. To execute machine code on another platform, for which it was not compiled, the virtual machine interprets this machine code and generates instructions for the executing platform. The virtual machine fetches, decodes and emulates the compiled machine code for the target platform. This often results in a one-to-n instruction relationship between the source platform and the destination platform [27].

2.2 Virtualization Obfuscators

Virtualization obfuscation poses a versatile mechanism, that can be used for static code obfuscation, tamper-proofing and anti-debugging [3, 20, 26]. Virtualization-based obfuscators usually translate selected parts of the binary into bytecode instructions of another language. Consequently, the size of the binary and the execution time increase significantly. Especially the increase in execution time is high, as – depending on the emulated VM architecture – one original instruction can be mapped to tens of VM instructions [16].

At the core of the virtualization-based obfuscation is the *virtual CPU*, an interpreter that must be able to translate instructions from the obfuscator language into instructions of the target architecture. By executing the read bytecode instructions during the execution cycle, the interpreter might simulate another architecture inside the currently running process [4, 23]. The execution cycle of a virtual machine refers to the way a virtual machine interpreter executes the bytecode. From a high-level viewpoint this execution cycle consists of three repeating instruction sets, namely the reading or fetching of bytecode, deciphering or interpreting of the fetched bytecode and lastly executing the instructions [4, 21, 22]. Other than in obfuscation this approach is also used in programming languages to provide platform independence, e.g., Java and its *Java Virtual Machine* (JVM) [7].

To be able to understand the behavior of the obfuscated instructions, both the virtual machine and the obfuscated code must be analyzed. The VM interpreters can be implemented as switch statements, or each instruction must call the next instruction according to the current instruction pointer of the virtual machine [24]. The opcodes for the instructions, which are interpreted by a virtual machine, possibly differ with every binary and compilation. It is not possible to dump the memory to restore the original instructions [26]. Consequently, virtualization-based obfuscation is not only effective against static but also dynamic analysis.

3 DESIGN AND IMPLEMENTATION

This section presents the design and implementation of VMAttack. A high-level design description is provided in Section 3.1, dynamic analysis techniques in Section 3.2, and static analysis techniques in Section 3.3. The automated combination approach is presented in Section 3.4.

3.1 Architecture

The foundation of VMAttack is comprised of an IDA abstraction layer that consists of a collection of support libraries that handle the IDA interaction. Our analysis modules build upon this abstraction layer, namely the dynamic, static, and optimization modules. Concluding the software stack is our automation layer on top of all available modules. A pipes and filters architecture is modeled where the different analysis capabilities of each module serve as filters and can be applied to either execution traces or IDA's disassembly datastructures in the static analysis case. VMAttack was designed around the assumption of an underlying *stack machine-based virtual machine*. The dynamic module can be further subdivided into the dynamic slicing and the dynamic loop detection modules. An additional optimization module provides execution trace filtering capabilities and improvements for the static analysis' IR pseudo-code instructions. The static module, consisting of the virtual translation and disassembly modules, enables the static approaches of VMAttack. On top of these modules, the automation layer allows for an automated combination of all available modules. The overall structure is illustrated in Figure 1. Additionally, the interactive result presentation enables direct interaction with the result in form of removal of single features, resetting and undoing changes, and colorization.

3.2 Dynamic Analysis Techniques

Dynamic analysis usually executes a binary to gain insights and information through observations of the binaries behavior. At time of execution, this can include register values or available values on the stack, which can speed up the analysis time considerably [9]. VMAttack executes the binary to save an instruction trace which is then used by all further analysis techniques. The trace can either be generated from within IDA or loaded from a file when the execution and analysis environments are different or a debugger not supported by IDA is used.

An instruction trace consists of four mandatory and two optional values. The thread id, starting address, disassembly, and CPU context after execution are mandatory while the score of a line and stack comments are optional. The stack comment represents additional information gained during the stack address propagation optimization, and the line score is a numeric value given by VMAttack's automation system to each trace line. The line score represents the importance of a trace line and is computed during the automated analysis. The trace is traversed once per selected optimization and once per selected analysis module.

Trace Optimizations. VMAttack's trace optimizations allow for information enrichment and an initial filtering of the trace, depending on the following analysis. Optimizations represent an important foundation, as most other analysis techniques use or even require one or more of these optimizations to be applied to the trace prior to their analysis. Available optimizations can be grouped into propagation (PO) and folding (FO) optimizations.

The focus of PO is to make information that is known at the time of the analysis easier available to the reverse engineer by enriching the trace with additional information. Propagations usually are considered safe to use because no

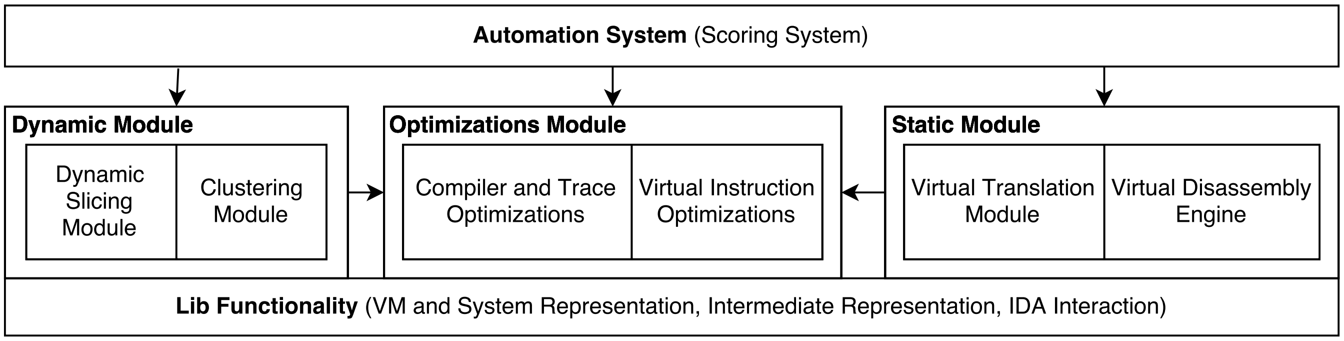
Automation System (Scoring System)
Dynamic Module
Dynamic Slicing Module
Clustering Module
Optimizations Module
Compiler and Trace Optimizations
Virtual Instruction Optimizations
Static Module
Virtual Translation Module
Virtual Disassembly Engine
Lib Functionality (VM and System Representation, Intermediate Representation, IDA Interaction)

Figure 1: Overall structure of VMAttack showing the available modules for static and dynamic analysis.

information is being left out. FO have the focus on reducing the trace in size making it easier to analyze. They can be quite powerful, reducing the trace by a very high percentage but also carry a risk of leaving out crucial information. At present, two propagation optimizations, constant and stack address propagation, and three folding optimizations, unused operand, operation standardization and peephole folding, are supported.

Dynamic Slicing Analysis. Dynamic program slicing refers to techniques able to find all statements that directly or indirectly affect the value of a certain variable [2]. Our *Dynamic Slicing Analysis* (DA) is a virtual machine architecture independent approach and provides separate program slices for every output variable of the VM's output registers. This allows a reverse engineer to view an excerpt of only those instructions that had a direct or indirect role in the VM's output computation. As part of the dynamic module, the DA expects an instruction trace and applies the two POs to the trace. The optimizations are a crucial part of the result presentation as they detail the stack address values used in the analysis output trace. Without the stack comments the trace would be taken out of context and consequently not be comprehensible. Moreover, constant propagation replaces dynamic values, e.g. register names, with their constant values at the time of computation. This enhances the information quality for each trace line this optimization works upon.

After the two optimizations, we proceed with the analysis step. As a first step, the VM input and output parameters are extracted from the trace. To this extent, all registers popped from the stack before function exit are considered VM output parameters.

Assuming an example of an output parameter `0x177DA` that was returned in the EAX register. The input parameters to the virtual machine function were `0xACDC` and `0xCAFE`.

For each output register, the analysis creates the dynamic slices by tracing the origins of the result value. First, the result value is followed on the trace to its last stack address. Let this stack address be `0x60FF50` in our example. Next, the analysis determines how the value saved on this address came to be. The algorithm traces the value to its point of origin which will be either a memory address or a computation. In our example, let the value `0x177DA` be moved to that location. Before being moved, the value was computed as an addition of the input parameters to the function.

During the VM execution, each of these instructions, i.e., moving, adding values, and returning a value, is represented by multiple instructions and furthermore, the offset computations for the next bytecode are added between the relevant instructions. DA removes irrelevant instructions and clarifies the computations that led to the output parameters. If the value's point of origin is a computation, the approach recursively retraces the two values used in the computation. This ensures that composed values are extracted correctly, meaning all trace lines regarding the computation are also traced back to the point of origin for the values used in the computation. If the values point of origin is another memory address, it is traced until no previous memory address can be found. As a result, the output registers are presented with all trace lines that compute or move the corresponding values from and to the stack. The reverse engineer can get the necessary information of how a value was computed and whether this value depends on the input parameters for the virtual machine function or not.

```
EAX:  value from stack address 0x60FF50

Addr  Disassembly               Stack Comment

4822  mov edx, [0x60FFEC]       [0x60FFEC]=0xACDC
4845  mov [0x60FF4C], 0xACDC
4822  mov edx, [0x60FFDC]       [0x60FFDC]=0xCAFE
4845  mov [0x60FF3C], 0xCAFE
421b  mov eax, [0x60FF4C]       [0x60FF4C]=0xACDC
4224  add [0x60FF3C], 0xACDC    [0x60FF3C]=0xCAFE
4328  mov eax, [0x60FF3C]       [0x60FF3C]=0x177DA
4328  mov [0x60FF50], 0x177DA
```

Listing 1: Output for the dynamic slicing analysis example. The result details the elemental values that are part of the eax output register.

The outlined example's result can be seen in Listing 1. The result shows the algorithm output in chronological order of trace lines, the algorithm on the other hand traversed the trace backwards.

This example was based upon a test case, where an addition of two values was obfuscated with a stack-based virtual machine. The instruction trace grew from 24 to 586 trace lines due to the obfuscation. Using our DA, we were able to reduce the trace down to 32 trace lines again, recovering the original functionality.

Clustering Analysis. In our clustering approach, we approximate loops on an aquired execution trace into clusters. The looping execution flow of the VM is used against itself by detecting groups of constantly repeating instructions. The execution trace is traversed to detect repetitions of successive addresses which are then declared clusters. Considering a clusters position and its stack interactions, specifically the read and write instructions, it is possible to determine the task for some clusters. The most common clusters correspond to a specific task in the fetch-decode-execute cycle of the VM and will have a more or less fixed interval of instructions between each cluster occurrence. If, for example, the same sequence of instructions is used by the virtual machine to move input values to their corresponding stack addresses, this sequence will be summarized with its own cluster. A possible clustered trace for this example is illustrated in Listing 2.

```
 Addr   Disassembly                Stack  Comment
...
Cluster 4328-4341
 4328   mov eax, [0x60FFEC]       [0x60FFEC]=0xDEAD
 432A   mov [0x60FF30], 0xDEAD
 432E   jmp ds:off4339
 4339   mov al, 0x489A
 433E   add esi, 7
 4341   jmp loc4404
...
Cluster 4328-4341
 4328   mov eax, [0x60FFDC]       [0x60FFDC]=0xBEEF
 432A   mov [0x60FF40], 0xBEEF
 432E   jmp ds:off4339
 4339   mov al, 0x489A
 433E   add esi, 7
 4341   jmp loc4404
...
```

Listing 2: Result output for the clustering analysis without the basic block display. The result shows two clusters that execute the same addresses of the binary.

Loops in the original binary correspond to rather large clusters, that incorporate several runs of the fetch-decode-execute cycle, writes to, and reads from the stack, all in consecutive order. Leftover unique instructions and rarely encountered clusters reveal a lot of information about the pre-virtualization logic of the original program, often even translating to instructions that were part of the original binary. The rarely encountered clusters often represent instructions that were part of the original binary but had to be switched with equivalent instructions because the VM interpreter was incapable of their execution.

The address-based clustering approach consists of consecutive clustering rounds on the instruction trace. In each clustering round, for every address and its neighbor, the occurrence of the exact same sequence of addresses is searched for in the trace. If this same sequence exists a cluster is created.

The clustering loop stops if a clustering round yields no length increase for any cluster. The reverse engineer can remove clusters during result presentation, additionally the default case already removes the most common clusters judged to be part of the VM execution cycle. As the focus of this algorithm is the filtering of non-functionality instructions, and not the decoding of the bytecode as it is in the static module, these parts of execution are filtered out and the results are presented to the reverse engineer.

Additionally, the reverse engineer can remove clusters, basic blocks and single instructions during result presentation, allowing for higher analysis precision.

3.3 Static Analysis Techniques

The static approach analyzes the virtual machine function and the provided bytecode to determine the functionality of the original function. To this extent, our modules work hand in hand to first determine executed instructions corresponding to the bytecode and analyze them to provide a disassembly of the virtual opcodes. This disassembly is translated into the IR language and optimized.

Supported language operators include stack, memory, computational and control flow operators and allow for a reduced familiarization period. Through the stack machine focused approach, our IR instructions allow for the extraction of bytecode to executed functionality mappings while simultaneously preserving the VM's internal value movements. These value movements provide additional meta information into the VM computational processes and can be of assistance in distinguishing between output and input relevant computations. This meta information also supports our decision for a result presentation in our *stack-based IR* (SBIR) language rather than a re-transformation back to assembly which does not allow for a natural way to convey stack-based context. Our SBIR language, being closer to JVM bytecode, allows for a clearer picture of the stack-based VM's used variables, return parameters and control flow dependencies.

Our static analysis approach uses the SBIR language to provide core insights and to simplify the bytecode functionality. In an initial step, the switch-case-structure, especially the jump table of the virtual machine interpreter, is used by the virtual disassembly engine to map the bytecode to corresponding x86/x64 assembly instructions. First, from start to end of the bytecode, every byte is examined towards the path the byte would execute inside the switch statement. This allows to represent bytes as assembly instructions, albeit this representation still includes a lot of overhead, representing the interpreter computations (e.g., the byte pointer increments towards the next byte).

The following algorithmic step removes the overhead from these assembly instructions by translating them into SBIR. At the beginning, a shadow stack is assumed and then, the executed instructions are translated by looking up memory access and interpreter specific patterns, which are needed for the pseudo instruction mapping. This process is completely automated and handled by the virtual translation module which in turn uses the virtual instruction optimizations to achieve best possible results. The virtual instruction optimizations offer improvements on an instruction level and allow for recognized patterns to be optimized during the translation.

3.4 Automation

The current approach at automation has the form of a scoring model for trace lines. Basically, an automated combination of VMAttack's available analysis modules with additional stack machine pattern matching mechanisms is employed to strip the trace of VM instructions and preserve functionality instructions. This algorithm currently consists of eight steps and can be extended with additional implementations for analysis modules. The procedure requires an instruction trace to be available and assigns a seed value to each trace line, called initial score. This value is computed according to the uniqueness of the trace line. The score is then raised or lowered according to the outcome of each analysis. A graphical overview of the algorithm is given in Figure 2.

Initialization. When dynamic analysis is enabled, an instruction trace is obtained in the first step. After its acquisition the trace lines are annotated with their initial score. This value is determined by computing a line's frequency inside the trace, where the more unique a line is, the higher the score will be.

First, the frequency of all lines is computed to determine the different frequency levels. The lowest frequency level will always be one, as a line needs to be executed at least once to be part of a trace. The highest level will vary from trace to trace. Next, the initialization step computes the initial score, where the frequency levels are used as score in reverse order, that is unique lines have the highest score, lines encountered twice with the second highest, and so forth. The initialization step concludes with the most frequent lines being graded with the worst possible score of one.

This approach of the initialization system makes use of the virtual machine execution cycle. The more unique instructions are, the more likely they correlate to payload functionality. For example, they will start from a better position in the grading system than the read byte and write byte instructions of the VM.

Propagations and Register Mappings. After the initialization step, the trace is still in its unoptimized state. Before the two PO required by the analysis are applied to the trace, the register mappings are computed. The trace is traversed once for the virtual machine's bytecode interaction, especially read and write instructions. If the registers interacting with the bytecode in these read and write instructions are only used for the addressing mechanism of the VM, and not the functionality computation they are flagged as addressing-only registers. Addressing-only registers are used to compute the offset for the next byte or the stack location where a value is currently located or will be stored at, resulting in addressing-only registers being irrelevant for the original functionality and more of a way to bloat the trace and thereby increase obfuscation. Furthermore, there can be functionality-only registers that compute directly on input or output values and intermediary results, and mixed registers where no conclusion can be drawn. Removing mixed registers poses the danger of leaving out crucial instructions so we decided to include instructions with mixed registers in the output of this step. Simultaneously, the propagations are applied to the trace in this step.

Scoring Steps. With the prerequisites out of the way, the actual scoring takes place. With every algorithmic step another analysis module is applied to the trace and the score is reevaluated according to the analysis findings. The scoring changes correlate directly to the (default or user defined) automation parameters for each analysis module.

- ***Dynamic Slicing Analysis:*** Modeled after the DA of VMAttack. The algorithm assumes the default case, that all output registers, except the addressing-only registers, are considered important and obtains the output backtraces for those output register. These backtrace lines experience a raise in their score.
- ***Register and Memory Usage:*** During the prerequisite step, the register mappings were computed, which detail whether registers exist, that are only used for the bytecode offset computation process. The score of trace lines containing those registers is reduced while the mixed and functionality register trace lines score is increased. In addition, trace lines with VM read instructions from the stack are raised if they use a functionality register.
- ***Clustering Analysis:*** Relies on the clustering analysis results to increase scores for unique lines and decreases the score of repeating lines. Not all repeating instructions are decreased. Instead, to reduce the score for fetching, reading and decoding instructions, the most frequent occurring lines experience a decrease.
- ***Folding Optimizations:*** Uses FO to determine important instructions. The application of all available optimizations results in a highly reduced trace. Moreover, register-based filtering is used on the trace to filter registers only used for the virtual machine execution cycle. For every remaining line in the reduced trace, the score for the corresponding line in the original trace is raised.
- ***Static Analysis:*** At the end of the analysis, the result from the static deobfuscation is integrated into the score. The score for the corresponding line for a SBIR instruction is raised.

Lastly, the trace lines with the highest score are presented to the reverse engineer. These trace lines represent the best candidates for the original binaries functionality instructions. However, the reverse engineer can also choose to view not only the best but also lower grades. This design results in

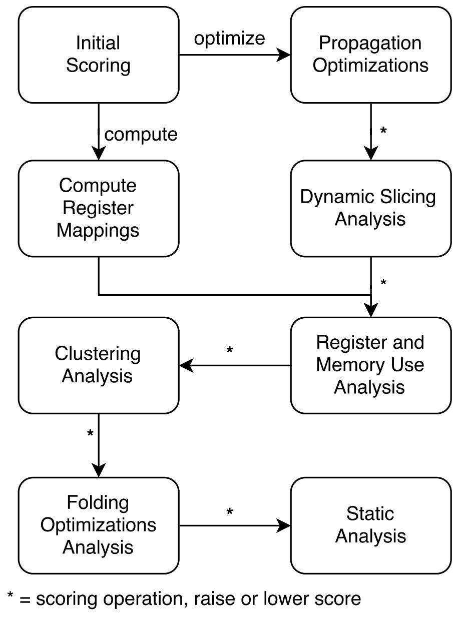
Initial Scoring
optimize
Propagation Optimizations
compute
*
Compute Register Mappings
Dynamic Slicing Analysis
*
Clustering Analysis
*
Register and Memory Use Analysis
*
Folding Optimizations Analysis
*
Static Analysis
* = scoring operation, raise or lower score

Figure 2: Scoring algorithm overview.

high robustness and speed of the combined automated analysis, as the overall score of a trace line consists of several grading steps, resulting in one failing analysis not significantly impacting the score. All the used algorithms are automatable ensuring low levels of required user interaction. Lastly, the stand-alone nature of each approach results in high *parallelization.*

4 EVALUATION

In this section, we evaluate VMAttack regarding the correctness of its automated analysis (Section 4.1) and the manual analysis capabilities. The static module (Section 4.2) is evaluated separately from the dynamic and optimizations modules (Section 4.2) because it does not require an instruction trace for the analysis.

To represent different viewpoints and challenges for the analysis and deobfuscation methods, we compiled functions containing common programming constructs, obfuscated them with VMProtect and then deobfuscated them using our IDA plug-in. VMProtect was chosen due to the use of stack machine-based virtualization [29] which is the main focus of our analysis approaches. The following constructs were chosen to represent the foundation of Turing-complete programming languages:

- arithmetic instructions: add, mul, sub, div
- logic instructions: and, or, xor
- control flow instructions: conditional and unconditional jumps, calls and recursion

The main objective of the evaluation was the *recovery of equivalent instructions from the virtualized binary.* Due to the significant increase of the instruction trace after obfuscation[1], we decided to use the overall instruction trace reduction as quantitative metric if the remaining trace still contained all of the instructions equivalent to the original functionality. The idea behind trace line reduction is that it results in a less time consuming analysis for a reverse engineer as long as no functionality preserving trace lines are left out. To evaluate our static analysis, the obfuscated binaries were deobfuscated by means of our static analysis module. The main objective remained the recovery of original functionality, but as a trace reduction metric can not be applied to our static approach, we decided to evaluate the *Control Flow Graph* (CFG) similarity instead, as was suggested by previous work [30]. In our static approach we generate the CFG directly from our SBIR language by using our optimized static analysis result. This was used in a similarity metric comparing the reconstructed CFG with the original CFG. To achieve high similarity the recovered CFG needs to display the same amount of basic blocks with the same number of incoming and outgoing edges. Furthermore, all recovered instructions need to be the SBIR equivalent instructions for the original binary.

All evaluations were performed on a desktop computer with an Intel Core i3 running at 1.8 GHz and four gigabytes of RAM. From the software side, we used Windows 10 with IDA Pro 6.9 and Python 2.7.12. The test binaries were compiled for the x86 or x64 architecture with VMProtect 2.9.

4.1 Automated Analysis Evaluation

The automated recovery of the original functionality worked for all test cases, two thirds of the test cases could be solved with default automation parameters. Automation parameters decrease or increase the importance of an analysis and should be changed if a reverse engineer favors a specific approach over another. Every approach has a specific focus during analysis, for example, the dynamic slicing analysis focuses on the output parameters while the clustering analysis focuses on unique instructions. Changing the importance weights allows to increase or decrease the focus on such an important area. Good examples are the branch binaries where the focus lies less on the output parameters and more on unique instructions and as such the automation parameters had to be adapted towards a less output centered approach. Table 1 presents the results of the automated analysis evaluation. Aside from the binary name, the table shows whether the default importance weights of the automated analysis were a good fit or had to be changed manually to successfully recover instructions. The last two columns detail the achieved instruction trace reduction and the specific reduction of the obfuscation layer, namely, the percentage of stripped VM instructions.

The achieved trace reduction removed at least 90% of the virtualization layer. For five out of ten binaries the original instructions could be recovered, the other cases allowed only

[1]On average, virtualization resulted in 191 times as many executed instructions for our test cases.

Binary	Automation Parameters	Overall Trace Reduction	VM Instructions Stripped
addition	default	99.65 %	98.58 %
multiplication	default	98.85 %	98.03 %
subtraction	default	99.53 %	98.54 %
division	increased DA value	97.94 %	94.28 %
bitwise and	default	99.53 %	96.83 %
bitwise or	default	98.14 %	96.16 %
bitwise xor	default	97.43 %	96.48 %
simple branch	increased CA value	96.92 %	96.48 %
looped branch	increased CA value	87.31 %	87.07 %
recursive fibonacci	increased DA value	91.46 %	91.20 %

Table 1: Results for the combined analysis with ten different test binaries obfuscated by VMProtect.

for a recovery of equivalent instructions which can be attributed to the VM interpreters limited instruction set. If a VM interpreter is not able to execute an instruction it will usually be substituted with available equivalent instructions. The loop binary displayed the least trace reduction of only 87.31%, which is still notable compared to the initial trace. Contrary to the previous test cases, the results of the combined binaries contained unnecessary conversion instructions and movement operations for stack values. The latter are helpful but not crucial for the understanding of the functionality and are an artifact of the virtual machine's stack architecture.

4.2 Manual Analysis Evaluation

VMAttack also supports manual analysis with each analysis technique offering one or multiple result presentations and interactive result interaction. The reverse engineer can directly highlight, remove or apply another analysis techniques to the result of the last analysis.

Static Only Analysis. For the evaluation of the static analysis the obfuscated binaries were deobfuscated by means of the static analysis module. After the execution of the static analysis a control flow graph was created from the deobfuscated virtual instructions and compared to the original binarie's CFG. The deobfuscation was judged a success, if both graphs had high similarity to each other and additionally the recovered virtual instructions represented the exact or equivalent computations to the original binary. The evaluation was judged a failure if the recovered computations were not equivalent to the original binaries computations or were falsely distributed amongst the basic blocks of the CFG. Similarity was computed according to the number of basic blocks and edges connecting them.

The result of the static analysis was a complete deobfuscation of all test cases. For the first eight binaries, the original CFGs matched the structure of the deobfuscated CFGs and the pseudo code instructions recovered were equivalent to those of the original binary.

Overall, the recovered virtual instructions resemble the original functionality in every test case and the generated CFG structures match, except for the recursive fibonacci, where the CFGs of the functions slightly differed. This difference, however, can be attributed to the equivalent instructions used by the VM interpreter, which resulted in a semantically equivalent but visually different CFG.

Dynamic Only Analysis. As the dynamic approaches require the presence of an execution trace, the main focus in this section lies on the trace reduction provided by VMAttack. The three supported dynamic approaches for the ten evaluated test cases are contrasted in Figure 3 . Each binary was evaluated through optimizations with subsequent register filtering, dynamic slicing analysis, and clustering analysis.

The dynamic slicing analysis shows potential to greatly reduce the trace. Only output relevant values, computations and their elemental parts are extracted. For the first eight binaries, the trace reduction was enormous, yet the analysis preserved all the relevant trace lines equivalent to the original binaries. Compared to the combined analysis, however, more artifacts from the virtual machine remain, as some move instructions for the corresponding values are not filtered from the trace.

Clustering analysis removes the most frequent clusters, but it lacks in the initial trace reduction compared to the dynamic slicing analysis and the optimization and filtering approach. Original computations are often part of a cluster. The best results were achieved by removing additional cluster groups one after another, as soon as analysis determined the cluster has nothing to do with the obfuscated functions. This provides additional information about the virtual machines structure and patterns of operation. This approach is more stable as it does not assume dependencies between input and output values, or dependencies towards the VM infrastructure. Additionally, this technique has proven to be suitable for loop and recursion detection. Through VMAttack's stack changes view, additional relevant information about important and less important stack addresses can be extracted.

FOs coupled with selective register folding help greatly at removing unwanted parts of the execution. Additionally, the VM artifacts, such as next byte address computations,

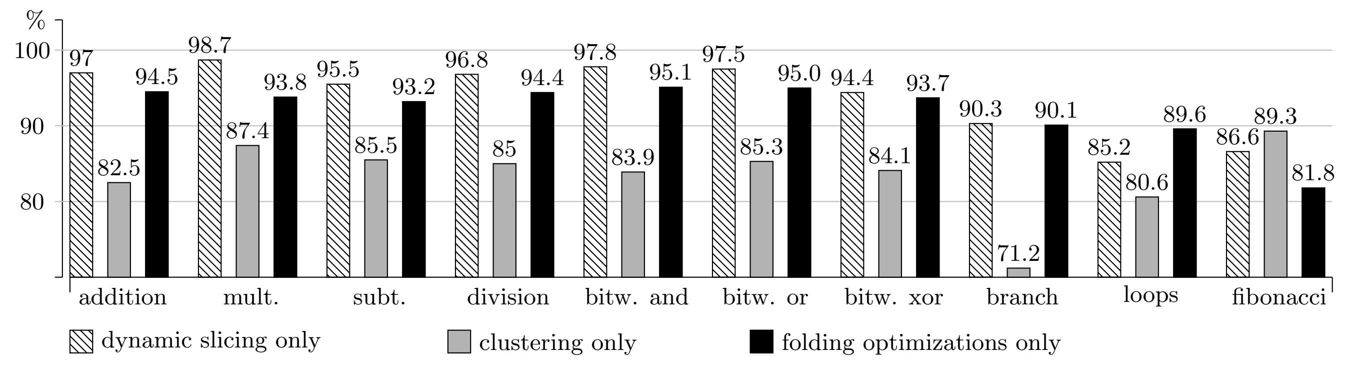

Figure 3: Overall trace reduction in percent achieved by dynamic analysis for ten test binaries.

can be excluded more reliably if their structure is analyzed beforehand.

5 DISCUSSION

VMAttack provides an interface to manual static and dynamic analysis techniques and bundles all available analysis modules as an automated approach. The central assumption, however, is that the underlying VM architecture used for obfuscation is a stack-based VM.

That being said, the presented techniques can be adapted for other VM architectures in the future. Dynamic slicing, for example, does not rely on architectural aspects but rather on the output variables the VM returns and the clustering analysis uses loop detection and pattern matching. While not having the same execution cycle of a stack-based VM, repeating patterns can also be observed on other architectures. The static module would need an extension in form of an architecture detection routine and a bytecode to SBIR extension for non-stack-based VMs. As our propagation optimizations have no architectural dependencies, additional analysis and implementation overhead would be needed for the adaptation of our FO.

Compared to prior solutions on the deobfuscation of stack-based VMs our approach provides a combination of previously untested approaches, namely dynamic slicing and optimizations, coupled with improvements upon prior work, e.g., our static module [22] and our clustering analysis [19]. Unlike previous semi-automated approaches by Rolles or Guillot et al. our approach can be completely automated [12, 22]. Our dynamic analysis techniques can arguably provide a computational advantage, as not the complete VM interpreter needs to be reverse engineered but rather the executed instructions are deobfuscated. Compared to Raber‘s automated greedy clustering approach our analysis provides a higher deobfuscation rate due to increased trace reduction, increased number of available analysis modules, and more efficient clustering and optimization algorithms. This increased precision, however, also results in a higher analysis runtime compared to Raber‘s clustering-only approach.

We consider the *specialization towards stack-based VMs* currently a scope defining factor for our approach. As other architectures are not supported yet, general approaches without architectural restrains offer a better alternative for non-stack machine VMs. Concolic analysis currently seems to hold promise for the general deobfuscation scenario which is why the general approaches rely either on symbolic execution, as in the case of Rotalumè [24], or taint tracking combined with execution trace analysis as suggested by Yadegari et. al [30]. For the specialized case of stack-based VMs our tool provides comparable results to Rotalumè with regard to trace reduction and comparable results with the approach by Yadegari et. al in terms of CFG reconstruction.

In light of general approaches favoring concolic analysis over static and dynamic techniques, the runtime should play a role in future evaluations of approaches as well. However, a direct comparison currently seems impossible due to the closed source character of Rotalumè as well as the proof of concept from Yadegari et al., and their approach evaluation does not factor in analysis runtime. Our open-sourced approach can be used for a direct comparison by future work. A runtime comparison between static, dynamic, and concolic approaches for virtualization-based packed binaries has not yet been conducted and would provide valuable insights for the reverse engineering community.

Moreover, current approaches, with exception of Rolles, seem to include trace generation at some point during the analysis which prompts the question about achieved code coverage. A solution similar to Driller [28] could be employed to generate an instruction trace with the desired code coverage.

6 CONCLUSION

Virtualization-based obfuscation resists conventional static and dynamic approaches by translating functionality into a random language and architecture. Prior approaches at deobfuscation either try to reverse engineer the VM interpreter or try to determine the semantics behind the executed instructions to peek at the functionality. We presented VMAttack, an analysis tool that aims at deobfuscating virtualization-based obfuscated binaries supporting a wide array of versatile strategies, combining both static and dynamic analysis.

The static approach focuses on the deobfuscation of the virtual machine's bytecode through automated analysis of the VM into SBIR. The dynamic approach focuses on the recovery of the original functionality through dynamic slicing and clustering-centered reduction techniques. The analysis approaches represent either improvements based upon previous work in the case of the static and clustering analysis or previously disregarded approaches as in the case for the dynamic slicing analysis. An optimization module features techniques to enrich the information available to the reverse engineer and the extraction of the relevant instructions out of an execution trace. The automation system combines all analysis techniques to provide an automated binary analysis. The combination of these analysis techniques and their weight in the overall analysis can be controlled as needed and the presentation of the result allows for dynamic interaction.

On average we are able to reduce the instruction trace by 89.86% through dynamic analysis and by 96.67% through combined analysis, eliminating the VM artifact instructions while retaining all instructions equivalent to the original functionality.

Lastly, VMAttacks placed second during the IDA Plug-In Contest in 2016 [11]. The open-sourced implementation can be found at https://github.com/anatolikalysch/VMAttack.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89). We also want to thank Prof. Dr.-Ing. Felix Freiling for his helpful comments on this paper.

REFERENCES

[1] Rambus Inc. 2009. About Self-Protecting Digital Content. (2009). https://www.rambus.com/about-spdc/, accessed on 06. March 2017.

[2] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. In *ACM SIGPlan Notices*, Vol. 25. ACM, 246–256.

[3] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. 2006. Proteus: virtualization for diversified tamper-resistance. In *Proceedings of the ACM workshop on Digital rights management*. ACM, 47–58.

[4] Samuel Chevet. 2015. Inside VMProtect. (2015). http://lille1tv.univ-lille1.fr/telecharge.aspx?id=d5b2487e-cacc-4596-ab37-dab2b362cb9e, accessed on 10. March 2017.

[5] Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of Virtualization-obfuscated Software: A Semantics-based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. ACM, New York, NY, USA, 275–284. DOI:http://dx.doi.org/10.1145/2046707.2046739

[6] Kevin Patrick Coogan. 2011. *Deobfuscation of Packed and Virtualization-obfuscation Protected Binaries*. Ph.D. Dissertation. Tucson, AZ, USA. Advisor(s) Debray, Saumya. AAI3468656.

[7] Iain D. Craig. 2006. *Virtual Machines*. Springer-Verlag.

[8] Yoni De Mulder, Brecht Wyseur, and Bart Preneel. 2010. Cryptanalysis of a perturbated white-box AES implementation. In *International Conference on Cryptology in India*. Springer.

[9] E. Eilam and E. J. Chikofsky. 2005. *Reversing: secrets of reverse engineering*. Wiley.

[10] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. 2011. Multi-stage Binary Code Obfuscation Using Improved Virtual Machine. In *Information Security*, Xuejia Lai, Jianying Zhou, and Hui Li (Eds.). Lecture Notes in Computer Science, Vol. 7001. Springer Berlin Heidelberg, 168–181. DOI:http://dx.doi.org/10.1007/978-3-642-24861-0_12

[11] Ilfak Guilfanov. 2016. IDA Pro Plug-in Contest 2016. (2016). https://www.hex-rays.com/contests/2016/index.shtml, accessed on 23. March 2017.

[12] Yoann Guillot and Alexandre Gazet. 2010. Automatic binary deobfuscation. *Journal in Computer Virology* (2010). DOI:http://dx.doi.org/10.1007/s11416-009-0126-4

[13] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*. ACM, 46–53.

[14] Johannes Kinder. 2012. Towards static analysis of virtualization-obfuscated binaries. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 61–70.

[15] Jasvir Nagra and Christian Collberg. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education.

[16] Eric Chien Nicolas Falliere, Patrick Fitzgerald. 2009. Inside the Jaws of Trojan.Clampi. (2009). https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/inside_trojan_clampi.pdf, accessed on 17. October 2016.

[17] Oreans Technologies. 2016. Themida. (2016). http://www.oreans.com, accessed on 10. March 2017.

[18] Frederic Perriot. 2009. Countering polymorphic malicious computer code through code optimization. (Nov. 24 2009). US Patent 7,624,449.

[19] Jason Raber. 2013. Virtual Deobfuscator – a DARPA Cyber Fast Track funded effort. (2013). http://www.cerosecurity.com/blackhat-usa-2013-presentaciones-y-diapositivas/, accessed on 10. March 2017.

[20] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. 2012. Camouflage in malware: from encryption to metamorphism. In *International Journal of Computer Science and Network Security*. 74–83.

[21] Rolf Rolles. 2007. Defeating HyperUnpackMe2. (2007). http://www.openrce.org/articles/full_view/28, accessed on 10. March 2017.

[22] Rolf Rolles. 2009. Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies (WOOT'09)*. USENIX Association, Berkeley, CA, USA.

[23] Shared Encyclopedia. 2016. VMProtect Logical instruction. (2016). http://et97.com/view/1281031.htm, accessed on 10. March 2017.

[24] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2009. Automatic Reverse Engineering of Malware Emulators. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. 94–109. DOI:http://dx.doi.org/10.1109/SP.2009.27

[25] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. (2016).

[26] Craig Smith. 2008. Creating Code Obfuscation Virtual Machines. In *Proceedings of the RECON 2008, Reverse Engineering Conference*. Neohapsis, Inc.

[27] Jim Smith and Ravi Nair. 2005. *Virtual machines: versatile platforms for systems and processes*. Elsevier.

[28] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. (2016).

[29] VMPSoft. 2016. VMProtect. (2016). http://www.vmpsoft.com, accessed on 10. March 2017.

[30] 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*. 674–691. DOI:http://dx.doi.org/10.1109/SP.2015.47