SoK: Automatic Deobfuscation of Virtualization-protected Applications

Patrick Kochberger
University of Vienna
Research Group Security and Privacy
Vienna, Austria
St. Pölten University of Applied Sciences
Institute of IT Security Research
St. Pölten, Austria
patrick.kochberger@univie.ac.at

Sebastian Schrittwieser
University of Vienna
Research Group Security and Privacy
Vienna, Austria
sebastian.schrittwieser@univie.ac.at

Stefan Schweighofer
St. Pölten University of Applied Sciences
Institute of IT Security Research
St. Pölten, Austria
is171011@fhstp.ac.at

Peter Kieseberg
St. Pölten University of Applied Sciences
Institute of IT Security Research
St. Pölten, Austria
peter.kieseberg@fhstp.ac.at

Edgar R. Weippl
University of Vienna
Research Group Security and Privacy
Vienna, Austria
edgar.weippl@univie.ac.at

ABSTRACT

Malware authors often rely on code obfuscation to hide the malicious functionality of their software, making detection and analysis more difficult. One of the most advanced techniques for binary obfuscation is virtualization-based obfuscation, which converts the functionality of a program into the bytecode of a randomly generated virtual machine which is embedded into the protected program. To enable the automatic detection and analysis of protected malware, new deobfuscation techniques against virtualization-based obfuscation are constantly being developed and proposed in the literature.

In this work, we systematize existing knowledge of automatic deobfuscation of virtualization-protected programs in a novel classification scheme and evaluate where we stand in the arms race between malware authors and code analysts in regards to virtualization-based obfuscation. In addition to a theoretical discussion of different types of deobfuscation methodologies, we present an in-depth practical evaluation that compares state-of-the-art virtualization-based obfuscators with currently available deobfuscation tools. The results clearly indicate the possibility of automatic deobfuscation of virtualization-based obfuscation in specific scenarios. Furthermore, however, the results highlight limitations of existing deobfuscation methods. Multiple challenges still lie ahead on the way towards reliable and resilient automatic deobfuscation of virtualization-based obfuscation.

ARES 2021, August 17–20, 2021, Vienna, Austria

ACM ISBN 978-1-4503-9051-4/21/08…$15.00
https://doi.org/10.1145/3465481.3465772

CCS CONCEPTS

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

KEYWORDS

Deobfuscation, Virtualiziation-based obfuscation, Application security

ACM Reference Format:
Patrick Kochberger, Sebastian Schrittwieser, Stefan Schweighofer, Peter Kieseberg, and Edgar R. Weippl. 2021. SoK: Automatic Deobfuscation of Virtualization-protected Applications. In *The 16th International Conference on Availability, Reliability and Security (ARES 2021), August 17–20, 2021, Vienna, Austria.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3465481.3465772

1 INTRODUCTION

Code obfuscation is commonly used by malware authors to protect their malicious code from detection and reverse engineering. The goal of obfuscation is to make the analysis process more difficult and time consuming [21]. In the research field of code obfuscation, many different techniques have been proposed in the literature and malware authors have even implemented their own protection schemes for which existing deobfuscation methods are mostly ineffective. Virtualization-based obfuscation is widely considered as one of the strongest techniques for obfuscating an application (or parts of it). Virtualization-based obfuscation describes the process of transforming the functionality of a piece of software into the bytecode of a randomly generated virtual machine which is embedded into the protected program. The random mapping of bytecode instructions of the virtual machine to native machine code functionality is not directly available to the analyst. Without knowing the meaning of each instruction of the bytecode it is not possible to get an understanding of the functionality of the code.

For scalable malware detection and analysis the deobfuscation process has to be highly automated because of the ever increasing number of malicious applications that have to be analyzed each day [21]. Many static and dynamic malware analysis approaches [18, 22, 23, 40] have been proposed in recent years. In the context of virtualization-based obfuscation several automated analysis tools have been published.

In this work, we give a theoretical overview on the different concepts for automatic deobfuscation of virtualization-protected programs, systematize existing knowledge, and provide the results of our practical study with available deobfuscators. For the study, we used four deobfuscation tools, which are freely available as open source software, to evaluate the strength of virtualization-based obfuscation against automatic deobfuscation: Virtual Deobfuscator [34], VMAttack [25], VMHunt [47], and a deobfuscator for the Tigress obfuscator [37, 38] (which we call Tigress DeObf throughout this paper). In our experiments, we set up four different evaluation scenarios with custom obfuscated samples and analyzed how effective the deobfuscated tools are against virtualization-protected samples and if limitations of the deobfuscation approaches on specific samples can be identified. In addition, we evaluated to what extent the approaches can be automated by implementing custom scripts to reduce the required user interaction as much as possible. To the best of our knowledge, this work is the first to provide an in-depth insight into the state of the arms-race between between state-of-the-art virtualization-based obfuscation schemes and current deobfuscation approaches.

The main contributions of this paper are:

- A systematic description of existing deobfuscation methodologies against virtualization-based obfuscation and a novel classification scheme.
- A comprehensive practical analysis of the effectiveness of existing deobfuscators against virtualization-based obfuscation with different sample sets, a highlight of the limitations of current automatic deobfuscation tools, and their impact on real-life malware deobfuscation and analysis.
- A demonstration of how much further existing solutions can be automated, which is crucial for automatic malware analysis.

The remainder of the paper is structured as follows: In section 2 and section 3 we describe the fundamentals of virtualization-based obfuscation and sytematically describe existing deobfuscation strategies. section 4 introduces the methodology of our practical evaluation. The results are presented in subsection 4.4. Finally, section 5 concludes our paper.

2 VIRTUALIZATION-BASED OBFUSCATION

Virtualization-based obfuscation is not a new software protection approach. In 1997, Collberg, Thomborson, and Low [15] described an obfuscation concept called "Table Interpretation": a protected program implements one or more interpreters that run custom bytecode embedded in the program. Modern virtualization-based obfuscation is based on the exact same foundations. Similarily, Hwang and Han [24] and Sharif, Lanzi, Giffin, and Lee [41] use the term *emulation-based obfuscation* to describe this concept.

In a nutshell, virtualization-based obfuscation implements an entire virtual machine inside a program. This usually includes the definition of a custom virtual instruction set, a bytecode interpreter, virtual registers, and a virtual stack. Since the original code is removed from the program and the virtual instruction set is only known to the developer at obfuscation time, but unknown to an attacker, well-known static analysis tools such as IDA Pro are unable to extract any meaningful information from a protected program except from the code of the bytecode interpreter (which usually is also heavily obfuscated). At runtime, the custom bytecode interpreter translates the virtual instructions (bytecode) into native machine code that is then run inside the processor.

Specifically, a virtualization-based obfuscator implements the following components.

Virtual Instruction Set. At obfuscation time, the original machine code of the program is converted to a bytecode which is defined by a custom virtual instruction set. This set is unknown to the attacker and might be different for each copy of a program, thus enabling code diversification. Besides randomizing the mapping of bytecode instructions to native machine opcodes, it is also possible to apply the fundamental obfuscation primitives *split*, *merge*, and *duplicate* [33]. This allows the virtual instruction set to either generate smaller or larger units of functionality, which do not map native machine opcodes one-to-one, or to implement multiple virtual instructions that all map to the same native machine opcode. This effectively prevents simple pattern matching attacks on the mapping of the virtual instruction set as for example opcode frequency analysis [12] is not feasible anymore. At runtime, the bytecode is interpreted with the help of handler functions in the virtual machine interpreter. For each bytecode instruction a handler function exists that is responsible for executing the corresponding native machine instruction.

Bytecode interpreter. For the custom virtual instruction set a matching bytecode interpreter is embedded into the protected program. Salwan, Bardin, and Potet [37] divided the execution flow in the bytecode interpreter into five basic steps. First, the *Fetch* step is responsible for retrieving the next bytecode instruction to be interpreted. Then, the *Decode* step decodes the instruction and its operands. The *Dispatch* step selects the correct handler and prepares the environment of the handler. In the subsequent *Handlers* step the selected handler executes the native machine instruction and proceeds to the final step. The *Terminator* gives back control to *Fetch* if there is code left to be interpreted. Otherwise, it terminates the interpreter. Blazytko, Contag, Aschermann, and Holz's [11] description of the execution flow virtualization-protected programs is similar, with the main difference being the consolidation of the *Handler* and *Terminator* steps.

The Achilles' heel of bytecode interpreters is the translation of opcodes of the virtual instruction set to native machine opcode. Many protection schemes were introduced in the literature.

Besides making the opcode mapping more complex (see paragraph "Virtual Instruction Set") the implementation of the bytecode interpreter can be obfuscated with traditional obfuscation schemes to protect the mapping of the virtual instruction set to the machine code instructions. Matryoshka et al. [19] demonstrated that it is

even possible to obfuscate the bytecode interpreter by transforming it to another virtual machine (*nested virtualization-obfuscation*). Xue, Tang, Ye, Li, Gong, Wangg, Fang, and Wang [46, 48] introduced a technique to mitigate knowledge reuse attacks by creating a custom bytecode that is different in each instance of an obfuscated program. The presented technique randomizes the connection between the virtual machine handlers and the executed virtualized code each time the obfuscation is applied. Kuang, Tang, Gong, Fang, Chen, Xing, Ye, Zhang, and Wang [28] combined several methods against knowledge reuse attacks. One method adds multiple virtual machines to an application with each VM having its own virtualized code and handler. Tang, Li, Ye, Cao, Chen, Gong, Fang, and Wang [43] introduced *VMGuards* [43], which aims at preventing code tampering from interfering with the virtual machine of the obfuscated program. This includes the context of the obfuscated program and the virtual machine code itself. Lee, Suk, and Lee [29] presented *VODKA*, which introduces a dynamic key and other methods that make the analysis of virtualization-protected binaries more difficult. Wang, Fang, Li, Yin, Zhang, and Gu [45] introduced several enhancements for VM-based obfuscation schemes to protect the bytecode code mapping and the context of the virtual machine.

Virtual Registers and Virtual Stack. Computer architectures such as x86 or ARM use general-purpose registers to temporarily store data. Virtualization-based obfuscators usually implement their own (often bigger) set of virtual registers that are then mapped to the hardware registers.

Most virtualization-based obfuscation schemes are stack-based, which means that all data that is exchanged between the memory and the virtual registers goes through a virtual stack similar to the native stack of common computer architectures.

3 DEOBFUSCATION OF VIRTUALIZATION-BASED OBFUSCATION

For over a decade, manual or (semi-)automatic deobfuscation and analysis of virtualization-protected programs as been discussed in the literature and many different approaches have been proposed. In this section, we present the results of our systematic literature review. First, we discuss existing approaches in chronological order. Then, we introduce our novel classification scheme for virtualization-based deobfuscation and present our findings from an in-depth comparision of the proposed methods.

In 2009, Rolles [36] defined six basic steps for deobfuscating virtualization-protected programs. In their paper, the authors used VMProtect to showcase their methodology. First, a reverse engineer has to manually design or choose an intermediate representation (IR) for mapping the virtual machine byte-code. Then it is then necessary to detect the virtual machine. The beginning of the VM is the position in the code, where the program transfers the control flow from the unprotected part to the virtual machine. The third step is to construct a disassembler for the virtual machine. The disassembler can afterwards lift the VM-specific bytecode into the IR. At this point, it is possible to optimize the intermediate representation code to simplify it. The final step generates regular machine code, which in their case is x86 assembly. This early approach requires a knowledgeable reverse-engineer to manually interact with the code at several stages in order to identify the code section which represents the virtual machine.

Sharif, Lanzi, Giffin, and Lee [41] proposed Rotalumé (2009), a framework for the automatic extraction of a bytecode trace to discover the syntax as well as the semantics of the virtual code from virtualization-protected programs. Rotalumé uses QEMU to emulate the sample and generate a trace. The tool lifts the trace to an IR, extracts forward and backward bindings, and clusters memory read instructions. A cluster is a vector of sets, where each set contains the addresses of memory accessed by the same variable. The final step is a behavioral analysis, which taints the clusters and looks for the fundamental execution properties of virtualization. This e.g. includes a main loop fetching bytes, dispatching to the handlers, and changing the VPC. Sharif, Lanzi, Giffin, and Lee analyzed samples protected with Code Virtualizer, Themida, and VMProtect. They were able to demonstrate the extraction of the control flow semantics, a bytecode trace, the bytecode syntax, x86 based semantics, and the CFG of the bytecode.

The binary manipulation framework METASM [20, 21] was first introduced in 2009 and is capable of disassembling virtualization-protected programs as well as other fundamental obfuscation techniques. It uses backtracking, symbolic emulation of instructions, and pattern matching to replace multiple instructions with simpler, unobfuscated ones. For the deobfuscation of virtualization-protection the initial version of METASM required the analyst to manually provide preliminary knowledge such as the type of encoding of the virtual machine, the implementation of the virtual registers, and how the transition between virtual instructions as well as functions (call, return) works. Only then METASM is able to automatically analyze the handler, assign mnemonics to the virtual instructions, and assemble code for the virtual machine. An improved version [21] of METASM was introduced in 2010, which pushes more towards automatic deobfuscation. Besides peephole optimisation (replacing known pattern with simpler ones) the updated METASM is capable of constant propagation (replacing variables with known constant values), constant folding (statically solving arithmetics and storing the value), operation folding (combine operations into single, simpler operations), and stack optimisation (removal of useless push-pop).

In 2011, Coogan, Lu, and Debray [16] introduced a generic deobfuscation approach and demonstrated it on virtualization-protected samples. Their approach uses execution traces as input and identifies system calls contained withit. Then, the analyst has to manually discard non-relevant ones. The following automatic analysis marks relevant instructions using the conditional control flow, control flow of the system calls, and flag instructions. It then cuts out a subtrace based on the marked code. The final product includes only the relevant and unobfuscated code.

In 2012, Kinder [26] extended Jakstab [27], a model-checking based tool for static binary analysis, to include analysis capabilities for virtualization-protected programs. It uses an extended variant of Bounded Address Tracking, which is VPC-sensitive.

Virtual Deobfuscator [34] was introduced by Raber in 2013. Virtual Deobfuscator accepts execution traces of the program to be analyzed as input. The deobfuscation process consists of three basic steps: The aim of the first step is to identify the bytecode instructions that are actually interpreted by the virtual machine. To this

end, the logic of the interpreter of the virtual machine is extracted. Pattern matching is used to form clusters of similar instructions inside the execution trace. A virtual machine inside a virtualization-protected binary then processes one bytecode instruction after the other. As a result, the code of the virtual machine is executed very often and patterns of this code execution can be found repeatedly in the execution traces. For clustering, the Virtual Deobfuscator requires the analyst to manually input a section size, which is used to identify a group of clusters. In order to choose a suitable section size, the location of the interpreter of the VM has to be identified by manually searching for the address of the virtualized function. Clustering is applied recursively until no instructions or existing clusters can be grouped together anymore. The remaining instructions contain the machine code of the actual functionality of the program. In the repackaging step the instructions are used to create binary code again. Finally, further redundancies are removed by a custom IDA Pro script, which also removes other, simple obfuscations from the binary. The authors successfully evaluated Virtual Deobfuscator with malicious and benign samples obfuscated with VMProtect [9] and Code Virtualizer [1].

Yadegari, Johannesmeyer, Whitely, and Debray introduced a generic deobfuscation methodology in 2015. They describe the semantics of a program as mappings or transformations of input to output values. Therefore, deobfuscation becomes the task of identifying and simplifying the code used for the transformations. The authors use bit-level taint analysis to identify the involved code as well as dependency analysis to find data dependencies. The simplified code is the result of semantics-preserving transformations of the code responsible for the input-output value flow. In the end, the CFG constructed from the simplified trace is further simplified by graph-based transformations.

Similarly, SEEAD [42] by Tang, Kuang, Wang, Xue, Gong, Chen, Fang, Liu, and Wang is a semantics-based generic deobfuscation approach developed in 2017. Unlike previous solutions, SEEAD dynamically analyzes the sample. To improve code coverage it uses multiple execution path exploration. Taint and control dependency analysis help to only choose branches tied to the input and therefore reduce analysis overhead. SEEAD collects the results of the dynamic analysis (taint analysis, control dependency analysis, and multiple execution path exploration), optimizes this intermediate data and reconstructs the control flow graph and the function call graph. SEEAD was tested against the obfuscation tools CF Obfuscator, MEMP, VMprotect, and Code Virtualizer as well as packed malware.

VMAttack [25] (2017) is a deobfuscation plugin for IDA Pro. It creates and annotates traces to make it easier to find virtualized functions. Kalysch, Götzfried, and Müller successfully evaluated VMAttack against the virtualization-obfuscator VMProtect [9].

In 2017, Blazytko, Contag, Aschermann, and Holz [11] presented Syntia, a deobfuscator based on synthesizing the semantics of obfuscated code. Their generic approach uses the Monte Carlo Tree Search (MCTS) and the Z3 SMT solver for trace simplification. Syntia divides the instruction trace into subtraces, the so-called trace windows. By feeding random inputs to these windows, the tool observes the outputs and generates input-output (I/O) pairs, which describe the semantics of the window. The synthesis step of Syntia uses the MCTS to generate expressions which fit the I/O pairs. Besides the extraction of the semantics of arithmetic VM instruction handlers, Syntia is able to simplify Mixed Boolean Arithmetic (MBA) expressions and Return Oriented Programming (ROP) gadgets.

Liang, Li, Zeng, and Fang [30] presented a method of deobfuscation through the code optimization of a compiler in 2018. They generated an execution trace of an obfuscated application and used symbolic execution on the trace to create symbolic values of the virtual machine handlers. Afterwards, a Miasm translator module translates the symbolic expressions to C code which the compiler then optimizes. Liang, Li, Zeng, and Fang evaluated their approach on samples protected by Code Virtualizer and VMProtect.

In 2018, Xu, Ming, Fu, and Wu [47] introduced the tool VMHunt. Its deobfuscation process consists of three basic steps: The "Virtualized Snippet Boundary Detection" starts by creating an execution trace with PIN [31]. VMHunt then applies normalization and preprocessing algorithms before the actual deobfuscation starts. It is based on instruction clustering to identify the beginning and the end of the actual code inside the protected binary. The "Virtualized Kernel Extraction" step then extracts the program's functionality using the backward slicing technique of BinSim [32]. Finally, the "Multiple Granularity Symbolic Execution" step analyzes the code to obtain its functionality. The authors tested VMHunt against Themida [5], Code Virtualizer [1], VMProtect [9] and Execrypter [2].

In 2018, Salwan, Bardin, and Potet [38] introduced a deobfuscation tool which is based on taint analysis, symbolic execution, and code simplification. It specifically targets the Tigress obfuscator. Their approach consists of 5 steps, one of which requires manual interaction. The first step identifies the input for the program. The input gives a starting point (first seed) for the analysis tool. Using the code and the seed, the dynamic taint analysis isolates the relevant instructions and generates a tainted subtrace. Afterwards, the tool generates a symbolic representation of the tainted subtrace leading to generalized symbolic expressions (AST). The fourth step includes path coverage analysis, which determines how a tainted path can be reached. This might result in new seeds, that can be used as new input in the first step of the algorithm. When no new seeds are discovered, the final step converts the symbolic path tree into LLVM IR and compiles it. The compilation from LLVM incorporates code optimization, building a simplified CFG, and even allows cross-compilation.

In 2019, Cheng, Lin, Gao, and Jia [13] attacked virtualization-based obfuscation by using the frequency distribution of opcodes. Based on their results, the authors introdcued a hardened virtualization-based obfuscation method called DynOpVm. In their deobfuscation approach they compared the frequency of native instructions with virtual instructions and found significant resemblances. Six out of the top 10 virtual instructions could directly be mapped to their native counterparts.

3.1 Classification of Deobfuscation Methodologies

Numerous different methodologies for (semi-)automatic deobfuscation of virtualization-based protections exist in literature. In order to make the various approaches more comparable and to highlight their key differences, we present a novel classification scheme based on four dimensions: (a) extracted artifacts, (b) analysis effort,

(c) degree of automation, and (d) generalizability. Table 1 give a systematic overview of discussed approaches.

Extracted artifacts. Generally speaking, the goal of all methods proposed in the literature is, of course, the de-obfuscation of virtualization-protected programs. However, the specific objectives and outputs of the de-obfuscation methods greatly differ, ranging from simple bytecode mapping to full reconstruction of the original code.

Analysis effort. Schrittwieser, Katzenbeisser, Kinder, Merzdovnik, and Weippl [40] described four categories of code analysis methods: (1) pattern matching, (2) static analysis, (3) dynamic analysis, and (4) human-assisted analysis. No existing approach for de-obfuscation of virtualization-based protections works without human interaction. Thus, according to their classification, all discussed deobfuscation approaches fall into the category of human-assisted analysis. However, we also classified the automatic parts of the analysis into three analysis categories:

- Purely static analysis (e.g. symbolic execution, slicing)
- Purely dynamic analysis (e.g. tracing, taint analysis)
- Combination of static and dynamic analysis (e.g. symbolic execution on an execution trace)

Degree of Automation. A third dimension of classification is the degree of automation of the analysis, i.e. the properties or compoments of a program that can be analyzed automatically for de-obfuscation.

Number of tools tested against. The last category shows against which obfuscation techniques and tools a deobfuscation method has been tested and is thus an indicator of how generic an approach is (e.g. Tigress Deobf expects a specific file structure and the obfuscation has to be applied with Tigress).

Figure 1 compares the 15 evaluated approaches along the 4 dimensions and shows the positive or negative deviation from the mean in each dimension. In all four dimensions a higher value means a better rating (e.g. the less analysis effort the higher the value). It is striking that no temporal trend is seen. The approaches with the highest overall ratings are [Yadegari et al. 2015] and SEEAD from 2017.

4 LAB EXPERIMENTS

The implementations of only a few deobfuscation approaches presented in the literature are publicly available. We performed lab experiments with four tools based on a custom sample set to reproduce the results from the original studies and to evaluate if further automation is feasable.

4.1 Deobfuscation Tools

In this study, we evaluated four deobfuscators against virtualization-based obfuscation: Virtual Deobfuscator [34], VMAttack [25], Tigress DeObf [37] and VMHunt [47]. All four tools were released as open-source software and are available online. A detailed description of the tools can be found in Section 3. Table 2 provides an overview of previous evaluations of the tools conducted by their respective authors.

4.2 Sample Sets

The sample sets used for this work are based on the work by Salwan, Bardin, and Potet [37]. We use programmatic implementations of different hash algorithms in order to verify, that the deobfuscated code behaves exactly like the original one. This allows for reliable verification of the correctness of the deobfuscated program. The hash algorithm implementations were selected from a sample repository[1] created by Banescu, Collberg, Ganesh, Newsham, and Pretschner [10] as part of their case study. The provided container from the repository is used on the Linux system for compiling and obfuscating the samples. From the repository, only the "simple-hash-functions" were selected for further obfuscation. All of the programs, except the "nohash.c" program, contain exactly two functions, the main function and the hash function. All hash programs are written in the C programming language and were compiled for both Linux and Windows systems and for 32-bit and 64-bit architectures, depending on the requirements of the evaluated deobfuscation tools. The same collection of programs was used as a foundation for each of the four sample sets.

We tested two different virtualization-based obfuscators using a range of different settings: VMProtect and Tigress. While VMProtect is the de facto standard for commercial virtualization-based obfuscators and widely used in commercial programs, Tigress is the most important academic obfuscation framework. One sample set was obfuscated with VMProtect and three sets were protected with different settings in Tigress. The samples were compiled with gcc [3] version 4.8.4 and mingw [4] gcc version 4.8.2.

Sample Set A. *Tigress* [14, 44] is an obfuscator for C code. It supports different obfuscation methods such as virtualization, control flow flattening, opaque predicates, and data encoding. Sample set A employs Tigress to protect the samples with virtualization-based obfuscation only. We used the Tigress command line option `--Transform=Virtualize` without any additional parameters (see Listing 5). Only the hash function within each sample was obfuscated. The generated obfuscated sample set includes a 32-bit and 64-bit version of each program.

Sample Set B. The second sample set was obfuscated with *VMProtect* [9], a commercial software protection solution. It works on binary code and is capable of virtualization-based obfuscation, code substitution ("mutation"), license management, bundling, and watermarking. For our study, we used the trial version of VMProtect 3.4.0. The trial version adds some additional functionality to obfuscated programs (e.g. a nag screen at program launch), thus increases the complexity of the samples. The samples are compiled with gcc version 7.5.0 and mingw gcc version 7.3.0 for both test systems. The programs are compiled with symbols to make the selection of the hash functions in *VMProtect* easier. Furthermore, only "Virtualization" is chosen as protection method while all other possible obfuscations were disabled.

Sample Set C. The samples in set C were again obfuscated with *Tigress* [14] but using the specific protection parameters of the "Tigress Challenge Chall0001" [6]. Apart from that, the process of

[1] https://github.com/tum-i22/obfuscation-benchmarks last access: 2021.03.25

Table 1: Classification of the deobfuscation methodologies. "M" means an analysis needs to be done manually or needs manual input. "✓" means the step is performed automatically or the approach uses the specified method. "-" means the approach does not fall into the specified category or the specified method is not part of the approach.

	Degree of automation					Extracted artifacts	Analysis methods							
	Understand input	Find context switch	Understand fetcher/vpc	Understand dispatcher	Understand bytecode		static	dynamic	Distributions/Statistics	Slicing	Symbolic execution	Taint analysis	Trace	Abstract interpretation
Rolles [36]	M	M	M	M	M	simplified code	✓	-	-	-	-	-	-	-
Rotalumé [41]	M	✓	✓	✓	✓	CFG, trace	-	✓	✓	-	-	✓	✓	-
METASM [20]	M	M	M	M	✓	bytecode mapping, lifting, recompilation	✓	-	-	✓	✓	-	-	-
METASM [21]	M	✓	✓	✓	✓	simplified code, decompiled code	✓	✓	-	✓	✓	-	✓	-
Coogan et al. [16]	M	✓	✓	✓	✓	simplified trace	-	✓	-	✓	-	-	✓	-
Kinder [26]	M	✓	M	✓	✓	CFG, invariants	✓	-	-	-	-	-	-	✓
Virtual Deobfuscator [34]	M	M	✓	✓	✓	simplified code	✓	✓	✓	-	-	-	✓	-
Yadegari et al. [49]	✓	✓	✓	✓	✓	simplified CFG	-	✓	-	-	✓	✓	✓	-
SEEAD [42]	✓	✓	✓	✓	✓	simplified trace, CFG, FCG	-	✓	-	-	-	✓	✓	-
VMAttack [25]	M	✓	✓	✓	✓	graded trace	✓	✓	-	✓	-	-	✓	-
Syntia [11]	M	M	M	M	✓	simplified VM instruction handler (semantics)	-	✓	✓	-	-	-	✓	-
Liang et al. [30]	M	✓	✓	✓	✓	simplified code	-	✓	-	-	✓	-	✓	-
VMHunt [47]	M	✓	✓	✓	✓	symbolic formula	-	✓	✓	✓	✓	-	✓	-
Tigress DeObf [37]	M	✓	✓	✓	✓	simplified code	-	✓	-	-	✓	✓	✓	-
DynOpVM [13]	M	M	M	M	✓	bytecode mapping	✓	-	✓	-	-	-	-	-

generating the samples was exactly the same as for **Sample Set A** and **Sample Set D** (see Listing 6).

Sample Set D. The approach for the generation of sample set D was similar to sample set C. The only difference is that the specific parameters of another "tigress challenge" [7] (Tigress Challenge Chall0003) was used (see Listing 7). This sample set includes more protections than sample sets A and C and therefore should be most difficult to deobfuscate for the evaluated tools.

4.3 Experimental Setup

In our experiments, we used two virtual machines based on VMware Workstation. Both systems were operated on the same hardware and were given the same computing resources. Table 3 provides an overview of the testing environment and shows which tools were evaluated on which machines. The Windows VM was exclusively used to generate the *VMProtect* [9] samples for sample set B as VMProtect is a Windows-only software.

4.4 Results

In the following, we describe the results of the lab experiments for each deobfuscation tool in detail. In summary, it was possible to get positive results for all four tools. However, our study also reveals major challenges for automatic deobfuscation of the analyzed samples. Table 4 gives an overview of the results. It shows how each tool performed on the four sample sets.

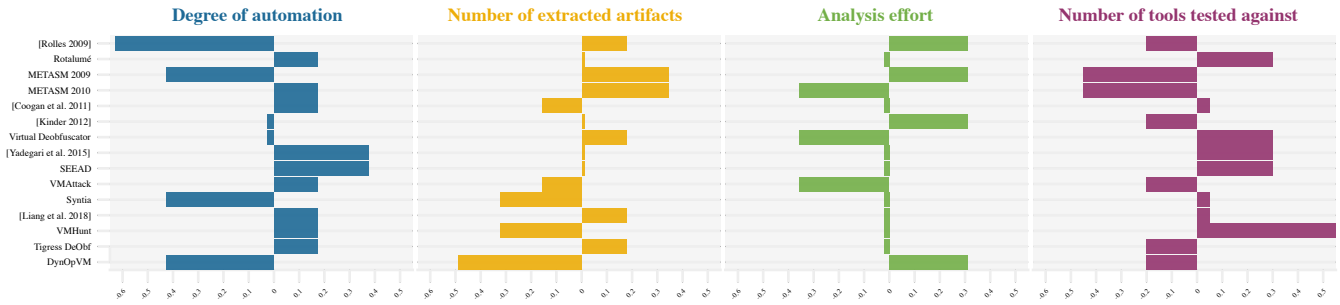

Figure 1: Visualization of the classification. Each dimension (degree of automation, analysis effort,…) shows the deviation of the tool or approach from the mean value. Bars to the left indicate a weakness compared to the average, while deflections to the right represent strong suits.

4.4.1 Virtual Deobfuscator. The evaluation of *Virtual Deobfuscator* was conducted on the Windows VM with the 32-bit binaries of each sample set.

Sample Set A. The *trace generation* worked for all tested samples in sample set A. The *clustering* algorithm of Virtual Deobfuscator also yielded promising results. We performed manual analysis of the clustering results to verify the virtual machine functions were indeed detected correctly. For all tested samples in this set, Virtual Deobfuscator was able to cluster the virtual machines. During the selection of the section sizes the boundaries of the virtual machine could not be properly defined for some samples. For two samples it was not possible to detect any boundaries of the virtual machine. For two other samples only one part (either beginning or end) of the boundary was detected. For samples for which automatic boundary detection was not possible, a section size bigger than the largest cluster in the virtual machine was chosen to be able to continue with the evaluation. In the *repackaging* phase of Virtual Deobfuscator *NASM* builds a binary of the assembly file from the previous phase. However, *NASM* was not able to repackage any of the samples in the set and reported various different errors. We manually analyzed the assembly files to identify the problem. Most likely the traces generated by OllyDbg in the first phase of the deobfuscation are not fully compatible with the Virtual Deobfuscator tool. Because the repackaging phase did not work, it was not possible to execute the *peephole optimization* step for sample set A.

Sample Set B. The trace generation of sample set B resulted in rather lengthy trace files. To reduce the size, only the relevant functionality of the samples (i.e. the hash algorithms) was recorded for the evaluation. The clustering algorithm of *Virtual Deobfuscator* worked well for all samples but two. The largest cluster was again chosen as the section size for these two samples. For four samples no assembly output was generated by the Virtual Deobfuscator. For all other samples, similarly to sample set A, the *repackaging* phase failed, because *NASM* was not able to compile the assembly output from the previous phase.

Sample Set C. Trace generation worked for all samples, however, automatic boundery detection failed for all but three samples. As with the previous sample sets the repackaging phase did not work for any sample of the set.

Sample Set D. Trace generation was performed successfully for all samples, however, it was not possible to automatically identify a boundary for two samples. Again, the *Repackaging* step did not yield results due to errors in the assembling step.

Challenges. The section sizes could not be identified reliably. Our workaround of automatically setting a large section size to increase the probability of extracting the correct clusters works, but results in larger than necessary section sizes. For some samples of set B *Virtual Deobfuscator* failed to generated any assembly output, because of a crash of the analysis algorithm. During the *repackaging* step it was not possible to reassemble a valid object file for any of the samples. Thus, the final *peephole optimization* step could not be evaluated.

4.4.2 VMAttack. The evaluation of *VMAttack* was performed on the Windows VM with 32-bit samples.

Sample Set A. The first analysis step, *trace generation*, sucessfully finished for all samples in the set. During the execution of the *grading analysis*, *VMAttack* prompts the user to select the virtualized function. This is a manual task, that cannot be automated. The finished graded trace then gives a human analyst a better understanding of the virtualized function. In particular, the handlers of the virtual machine can be identified more easily in the trace. In our study, we considered the *manual evaluation* phase successful if any additional information about the virtualized function could be gleaned from the code. In sample set A, highlighting of the relevant instructions worked well for all samples.

Sample Set B. Again trace generation and grading analysis finished successfully. However, the grading of the individual instructions did not provide additional insight into better understanding of the original algorithms, compared to the obfuscated binaries.

Sample Set C. Both trace generation and grading analysis finished successfully for all samples of set C. However, similar to sample set B, it was not possible to identify the relevant instructions of the original algorithm representing the hashing algorithm. The grading of the individual instructions did not provide enough

Table 2: Overview of what obfuscation tools have been tested against which deobfuscation tools and approaches.✓= Deobfuscation method uses the obfuscation tool for the evaluation.- = The publication does not use the protection tool for the evaluation.The column Custom is marked for malware or custom virtualizer or packer.

	Custom	Code Virtualizer	Rewolf Virtualizer 64-bit	Tigress	Themida	VMProtect	Number of samples	Type of samples	Evaluation metric
DynOpVM [13]	-	-	✓	-	-	-	128	SPEC CPU2006 benchmark	-
Tigress DeObf [37]	-	-	-	✓	-	-	920	hash functions	size, correctness
VMHunt [47]	✓	✓	-	-	✓	✓	16	OS programs, malware	size, time
Liang et al. [30]	-	✓	-	-	-	✓	4/6	toy	size, simplification score
Syntia [11]	-	-	-	-	✓	✓	1	toy	analysis time, identified handlers
VMAttack [25]	-	-	-	-	-	✓	10	toy	simplification score
SEEAD [42]	✓	✓	-	-	-	✓	16/34	toy, malware	simplification score
Yadegari et al. [49]	-	✓	-	-	✓	✓	44	toy, malware	similarity
Virtual Deobfuscator [34]	✓	✓	-	-	-	✓	1	toy	simplification score
Kinder [26]	✓	-	-	-	-	-	1	toy	known invariants
Coogan et al. [16]	-	✓	-	-	-	✓	12	toy, malware	simplification score
METASM [21]	-	-	-	-	-	-	-	-	-
METASM [20]	-	-	-	-	-	-	2	toy	-
Rotalumé [41]	-	✓	-	-	✓	✓	21/29	toy, OS programs, malware	size
Rolles [36]	-	-	-	-	-	✓	-	-	-

information to correctly highlight instructions that belong to the original programs.

Sample Set D. Trace generation and the *Grading Analysis* phase did finish for all tested samples in the set. By manually evaluating the graded trace it was possible to identify the virtual machine structure, but not the particular instructions that represent the original algorithm. In a real-world scenario, *VMAttack*'s gradings of the samples in set D would definitely support a reverse engineer in the initial analysis phase, because the structure of the virtual machine can be identified more easily. However, it would not be possible to gain deeper understanding of the functionality of the program without additional (manual) analysis steps. Still, since the results are useful for analyzing the given samples, the *Manual Evaluation* was considered to be successful.

Challenges. During the evaluation of VMAttack, we faced two main challenges, which make automatic deobfuscation significantly less effective. Firstly, the analysis of sample set B and sample set D took very long (more than four hours for each sample). Considering that malware analysis labs have to analyze thousands of samples a day, the analysis times measured in this study are definitely too long for practical use. Secondly, the last step of the VMAttack algorithm is the manual analysis of the graded trace that cannot be automated. Moreover, the evaluation of the effectiveness of the grading process is highly subjective and dependent on the human analyst. In general, the degree of automation opportunities for

Table 3: Comparison of the two VM used in the lab experiments.

	Linux System	**Windows System**
OS	Ubuntu Server 18.04 x64	Windows 7 SP1 x64
CPU Cores	2 Cores	2 Cores
RAM	4 GB	4 GB
Virtual Deobfuscator [34]	-	Evaluated
VMAttack [25]	-	Evaluated
VMHunt [47]	Evaluated	Partly (Sample Set B)
Tigress DeObf [37]	Evaluated	-

Table 4: Experiment results of our study. We evaluated Tigress DeObf a second time with modified samples. The results for both versions of the samples are shown in the table. The number of symbols in a field represents the number of analysis steps for a given tool. A check mark shows a successful analysis. A "~" symbol indicates, that not all samples of a set could be analyzed successfully for a given analysis step. A "-" symbol marks analysis step that did not yield positive results. A "*" symbol shows that the results of the specific analysis step need manual interpretation or interaction. † marks the analysis with Tigress DeObf with the patched samples.

	Sample Set A	**Sample Set B**	**Sample Set C**	**Sample Set D**
Virtual Deobfuscator	✓✓- -	✓✓- -	✓✓- -	✓✓- -
VMAttack	✓✓*	✓✓*	✓✓*	✓✓*
VMHunt	✓- - -	✓- - -	✓- - -	✓- - -
Tigress DeObf	- -	- -	- -	- -
Tigress DeObf †	✓~	- -	✓~	✓~

VMAttack is low compared to other tools. Still, we implemented a script in Listing 3 to assist with the *manual evaluation* by identifying patterns in the graded traces.

4.4.3 VMHunt. VMHunt supports execution traces generated from 32-bit binaries only. It was evaluated on the Linux VM with the sample sets A, C, and D and on the Windows VM with sample set B (*PIN* failed to correctly instrument the samples on the Linux system).

Sample Sets A, C, and D. It was possible to generate the traces for all samples in sample set A, sample set C and sample set D. However, the tool was not able to extract the virtual machine from any of those traces. *VMHunt* did finish the analysis extraction algorithm correctly, but it produced no output.

Sample Set B. Sample set B consists of the most complex samples. During trace generation, we observed, that the size of the generated trace files grew fast. In order to keep the trace size small enough for deobfuscation, the tracer component was modified to only record a specific range of instructions. In the Windows VM, *Boundary detection* failed in a very early phase of the analysis. We continued *boundary detection* with the recorded traces in the Linux VM where it finished successfully. However, the result set was again empty for all samples.

Challenges. The boundary detection algorithm of VMHunt did not return a valid results for any sample. Despite a thorough analysis of the problem, we were unable to identify the cause.

RAM usage of the boundary detection step was a challenge during the analysis of sample set B as it exceeded the 4GB of available memory on the testing system. To finish the analysis, we had to increase the RAM to 12GB. Moreover, *boundary detection* crashed the VMHunt algorithm during the analysis of one of the samples.

4.4.4 Tigress DeObf. Tigress DeObf [37] was evaluated on the Linux VM with 64-bit binaries of the sample sets A, C, and D. For sample set B, we additionally used the 32-bit binaries of the samples.

Sample Sets A, C, D. *Tigress DeObf* was not able to analyze any of the samples in the sample sets A, C, and D, because Tigress DeObf requires a very specific structure of the application under scrutiny and cannot be used universally for any program. Specifically, the binary must call both "strtoul" and "printf". These two calls enable Tigress DeObf to identify the start and end points of the protected payload of the virtual machine. The Tigress challenges as well as the samples provided in the repository of *Tigress DeObf* adhere to these specific properties. Our samples, however, do not call "strtoul", but directly retrieve the input from "argv[1]". We modified all samples to include the required function calls. *Tigress DeObf* was then able to successfully deobfuscate most of them. Four samples from the sample sets A, C, and D produced incorrect results (i.e. hash calculations).

Sample Set B. *Tigress DeObf* reproducibly crashed when analyzing the 64-bit samples from sample set B. The analysis of the 32-bit samples was successful, however, RAM requirements exceeded the initial 4GB of the VM. We gradually increased RAM until we reached the physical limits of the host machine at 24GB. However, Tigress DeObf still filled up the entire available memory without finishing the analysis.

Challenges. Aside from the required structure of the binaries, two major challenges were the memory consumption when analyzing samples from the most complex sample set B and the crashes during the analysis of 64-bit samples. Furthermore, we identified some incorrect functionality in the deobfuscated binaries.

5 CONCLUSION

In this paper, we presented a novel classification of deobfuscation methodologies against virtualization-based protections with a focus on workflow automation. We further performed an empirical study on available deobfuscation tools, in which we evaluated the effectiveness and resilience of four tools against virtualization-based obfuscation. Our paper shows where we currently stand in the arms race between protection and analysis and indicates specific challenges of the deobfuscation tools, in particular when used in an automatic large-scale analysis of malware samples.

The results clearly indicate the difficulties in the automation of the deobfuscation of virtualization-protected programs with current techniques and tools. Either only very specific samples work or the process needs manual interaction to produce valid results. Further research is needed to make deobfuscation more robust in the future in order to enable large-scale analysis.

ACKNOWLEDGMENTS

This work was funded by the Austrian Science Fund (FWF) under grant I 3646-N31.

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

REFERENCES

[1] 2020. Code Virtualizer. https://www.oreans.com/CodeVirtualizer.php Last Accessed 2020.03.12.
[2] 2020. EXECrypter. https://web.archive.org/web/20180520123330/http://www.strongbit.com/execryptor.asp Last Accessed 2020.04.24.
[3] 2020. GCC, the GNU Compiler Collection. https://gcc.gnu.org/ Last Accessed 2020.07.11.
[4] 2020. MinGW - Minimalist GNU for Windows. http://www.mingw.org/ Last Accessed 2020.07.11.
[5] 2020. Themida. https://www.oreans.com/Themida.php Last Accessed 2020.04.24.
[6] 2020. Tigress Challenge Script 0001. http://tigress.cs.arizona.edu/scripts_txt/0001.sh.txt Last Accessed 2020.05.28.
[7] 2020. Tigress Challenge Script 0003. http://tigress.cs.arizona.edu/scripts_txt/0003.sh.txt Last Accessed 2020.05.28.
[8] 2020. VMHunt. https://github.com/s3team/VMHunt Last Accessed 2020.02.27.
[9] 2020. VMProtect. https://vmpsoft.com/ Last Accessed 2020.01.31.
[10] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code Obfuscation against Symbolic Execution Attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC '16)*. ACM, 189–200.
[11] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC)*. USENIX, 643–659.
[12] Xiaoyang Cheng, Yan Lin, Debin Gao, and Chunfu Jia. 2019. DynOpVm: VM-based software obfuscation with dynamic opcode mapping. In *International Conference on Applied Cryptography and Network Security*. Springer, 155–174.
[13] Xiaoyang Cheng, Yan Lin, Debin Gao, and Chunfu Jia. 2019. DynOpVm: VM-Based Software Obfuscation with Dynamic Opcode Mapping. In *Applied Cryptography and Network Security*. Springer International Publishing, 155–174.
[14] Christian Collberg. 2020. The Tigress C diversifier/obfuscator. https://tigress.wtf/introduction.html Last Accessed 2020.03.01.
[15] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations*. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.
[16] 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*. ACM, 275–284.
[17] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 337–340.
[18] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM Comput. Surv.* 44, 2 (02 2012).
[19] Sudeep Ghosh, Jason D Hiser, and Jack W Davidson. 2015. Matryoshka: Strengthening Software Protection via Nested Virtual Machines. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 10–16.
[20] Yoann Guillot and Alexandre Gazet. 2009. Semi-automatic binary protection tampering. *Journal in Computer Virology* 5 (05 2009), 119–149.
[21] Yoann Guillot and Alexandre Gazet. 2010. Automatic binary deobfuscation. *Journal in Computer Virology* 6, 3 (2010).
[22] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. 2016. A Survey on Aims and Environments of Diversification and Obfuscation in Software Security. In *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016 (CompSysTech '16)*. ACM, 113–120.
[23] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. 2018. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology* 104 (12 2018).
[24] Joonhyung Hwang and Taisook Han. 2018. Identifying Input-Dependent Jumps from Obfuscated Execution using Dynamic Data Flow Graphs. In *Proc. of the 8th Software Security, Protection, and Reverse Engineering Workshop*. ACM.
[25] 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 (ARES '17)*. ACM.
[26] Johannes Kinder. 2012. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 61–70.
[27] Johannes Kinder and Helmut Veith. 2008. Jakstab: A Static Analysis Platform for Binaries. In *Computer Aided Verification*. Springer Berlin Heidelberg, 423–427.
[28] Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen, Tianzhang Xing, Guixin Ye, Jie Zhang, and Zheng Wang. 2016. Exploiting Dynamic Scheduling for VM-Based Code Obfuscation. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 489–496.
[29] Jae-Yung Lee, Jae Hyuk Suk, and Dong Hoon Lee. 2019. VODKA: Virtualization Obfuscation Using Dynamic Key Approach. In *Information Security Applications*. Springer International Publishing, 131–145.
[30] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. 2018. Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization. In *Information and Communications Security*. Springer International Publishing, 313–324.
[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, 190–200.
[32] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 253–270.
[33] Jasvir Nagra and Christian Collberg. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education.
[34] J Raber. 2013. Virtual deobfuscator-a darpa cyber fast track funded effort. *Proc. of the 16th Black Hat USA* (2013).
[35] Jeffrey Racine. 2000. The Cygwin tools: a GNU toolkit for Windows. *Journal of Applied Econometrics* 15 (2000), 331–341.
[36] Rolf Rolles. 2009. Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies (WOOT'09)*. USENIX Association.
[37] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. 2018. Symbolic Deobfuscation: From Virtualized Code Back to the Original. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 372–392.
[38] Jonathan Salwan, Sebastien Bardin, and Marie-Laure Potet. 2020. Tigress_Protection. https://github.com/JonathanSalwan/Tigress_protection Last Accessed 2020.02.27.
[39] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 31–54.
[40] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Computing Surveys (CSUR)* 49, 1 (04 2016), 4:1–4:37.
[41] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic Reverse Engineering of Malware Emulators. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 94–109.
[42] Zhanyong Tang, Kaiyuan Kuang, Lei Wang, Chao Xue, Xiaoqing Gong, Xiaojiang Chen, Dingyi Fang, Jie Liu, and Zheng Wang. 2017. SEEAD: A Semantic-Based Approach for Automatic Binary Code De-obfuscation. In *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 261–268.

[43] Zhanyong Tang, Meng Li, Guixin Ye, Shuai Cao, Meiling Chen, Xiaoqing Gong, Dingyi Fang, and Zheng Wang. 2018. VMGuards: A Novel Virtual Machine Based Code Protection System with VM Security as the First Class Design Concern. *Applied Sciences* 8, 5 (2018).

[44] Clark Taylor and Christian Colberg. 2016. A Tool for Teaching Reverse Engineering. In *2016 USENIX Workshop on Advances in Security Education (ASE 16)*. USENIX.

[45] Huaijun Wang, Dingyi Fang, Guanghui Li, Xiaoyan Yin, Bo Zhang, and Yuanxiang Gu. 2013. NISLVMP: Improved Virtual Machine-Based Software Protection. In *Proceedings of the 9th International Conference on Computational Intelligence and Security (CIS '13)*. IEEE, 479–483.

[46] Wei Wang, Meng Li, Zhanyong Tang, Huanting Wang, Guixin Ye, Fuwei Wang, Jie Ren, Xiaoqing Gong, Dingyi Fang, and Zheng Wang. 2019. Invalidating Analysis Knowledge for Code Virtualization Protection Through Partition Diversity. *IEEE Access* 7 (2019), 169160–169173.

[47] 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 '18)*. ACM, 442–458.

[48] Chao Xue, Zhanyong Tang, Guixin Ye, Guanghui Li, Xiaoqing Gong, Wei Wangg, Dingyi Fang, and Zheng Wang. 2018. Exploiting Code Diversity to Enhance Code Virtualization Protection. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 620–627.

[49] 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*. IEEE, 674–691.

A APPENDIX

```
void getctx(ADDRINT addr, CONTEXT *fromctx,
    ADDRINT raddr, ADDRINT waddr) {
if(addr >= 0x00400000 && addr <= 0x00600000) {
  fprintf(fp, "%x;%s;%x,%x,%x,%x,%x,%x,%x,%x,%x
      ,%x,\n", addr, opcmap[addr].c_str(),
    PIN_GetContextReg(fromctx, REG_EAX),
    PIN_GetContextReg(fromctx, REG_EBX),
    PIN_GetContextReg(fromctx, REG_ECX),
    PIN_GetContextReg(fromctx, REG_EDX),
    PIN_GetContextReg(fromctx, REG_ESI),
    PIN_GetContextReg(fromctx, REG_EDI),
    PIN_GetContextReg(fromctx, REG_ESP),
    PIN_GetContextReg(fromctx, REG_EBP),
    raddr, waddr);
}
}
```

Listing 1: Modified code of the *VMHunt* [47] "tracer" [8]. The modifications reduce the size of the generated traces. They are used for the evaluation of sample set B and restrict the address range.

```
#!/bin/bash

for file in ./x32_linux*; do
    pin -t ./instracelog.so -- $file 0123456789
    ./vmextract instrace.txt
done
```

Listing 2: Script for the automation of *VMHunt* [47].

```
#!/usr/bin/env python3

# ./script <JSON> <SIZE> <START_VM> <END_VM>
# Example: ./filterJson.py bkdr.json 5 401603
    401BDD

# The size parameter is used to output
    instructions,
# which are close to the highest graded
    instruction in the search space

import sys
import json
from collections import OrderedDict
from itertools import islice

with open(sys.argv[1]) as f:
    trace = json.load(f, object_pairs_hook=
        OrderedDict)

if len(sys.argv) >= 5:
    size = int(sys.argv[2]) # used to output
        MAX_GRADE - size

    start = sys.argv[3].lower()
    end = sys.argv[4].lower()

    startVM = 0
    endVM = 0

    for instruction in trace:
        if trace[instruction][1] == start:
            startVM = int(instruction)

        if trace[instruction][1] == end:
            endVM = int(instruction)

    result = OrderedDict(islice(trace.items(),
        startVM, endVM))
    print(f'StartVM: {startVM}')
    print(f'EndVM: {endVM}')
else:
    # Size, Start & End of VM are not given
    # check the complete trace with a default
        grade size range of 10
    size = 10
    result = trace

maxGrade = max([result[instruction][-1] for
    instruction in result])
print(f'Maxgrade: {maxGrade}')

filterdTrace = [result[instruction] for
    instruction in result if result[instruction
    ][-1] >= (maxGrade-size)]

for instruction in filterdTrace:
    print('G:{}\tAddr:0x{}\tINST:{} '.format(
        instruction[-1], instruction[1],
        instruction[2]))
```

Listing 3: Script to support the analysis of the graded trace JSON output from VMAttack [25]. The script filters the graded trace and therefore highlights patterns.

```
#!/bin/bash

SET='s1'
```

```
4
5  for file in ~/tests/samples/testing_samples/
       tp_modified_samples/$SET/x64_linux_*; do
6      ~/tests/deobf_tools/Tigress_protection/solve
           -vm.py $file
7
8      out1=$(~/tests/deobf_tools/
           Tigress_protection/
           deobfuscated_binaries/$(basename $file)
           .deobfuscated 123456789 | xargs -I{}
           python -c "print str(hex({}))[2:]")
9      out2=$($file 123456789)
10
11     if [ "$out1" == "$out2" ]; then
12             echo "$out1"
13             echo "$out2"
14             mv ~/tests/deobf_tools/
                   Tigress_protection/
                   deobfuscated_binaries/$(
                   basename $file).deobfuscated ~/
                   tests/results/tp/$SET/
15     else
16             echo "Deobfuscation failed"
17             mv ~/tests/deobf_tools/
                   Tigress_protection/
                   deobfuscated_binaries/$(
                   basename $file).deobfuscated ~/
                   tests/results/tp/$SET/$(
                   basename $file).
                   deobfuscated_failed
18     fi
19 done
```

Listing 4: Script to automate the analysis of multiple samples with Tigress DeObf [37].

```
1 TIGRESS_SETTINGS="--Transform=Virtualize"
2 ...
3 tigress $TIGRESS_SETTINGS --Functions=$function
      --out=$virt_file $file
```

Listing 5: Tigress settings for Sample Set A

```
1  TIGRESS_SETTINGS="\
2  --Transform=InitEntropy \
3    --Functions=main \
4  --Transform=InitOpaque \
5    --Functions=main \
6    --InitOpaqueCount=1 \
7    --InitOpaqueStructs=list,array \
8  --Transform=Virtualize \
9    --VirtualizeMaxDuplicateOps=2 \
10   --VirtualizeAddOpaqueToVPC=true \
11   --VirtualizeDispatch=direct \
12   --VirtualizeOperands=stack,registers \
13   --VirtualizeMaxMergeLength=5 \
14   --VirtualizeSuperOpsRatio=2.0 \
15 --VirtualizeInstructionHandlerSplitCount=2"
16 ...
17 tigress $TIGRESS_SETTINGS --Functions=$function
       --out=$virt_file $file
```

Listing 6: Tigress settings for Sample Set C

```
1  TIGRESS_SETTINGS=" \
2    --Transform=InitEntropy \
3      --Functions=main \
4    --Transform=InitOpaque \
5      --Functions=main \
6      --InitOpaqueCount=1 \
7      --InitOpaqueStructs=list,array \
8    --Transform=Virtualize \
9      --VirtualizeMaxDuplicateOps=2"
10 TIGRESS_SETTINGS2="--Transform=EncodeArithmetic"
11 TIGRESS_SETTINGS3="--Transform=Split"
12 TIGRESS_SETTINGS4="\
13     --SplitCount=50 \
14     --SplitName=SPLIT \
15   --Transform=Merge \
16     --Functions=%30  --Exclude=main \
17     --MergeName=MERGE1\
18     --MergeFlatten=true --MergeFlattenDispatch=
           switch \
19   --Transform=Merge \
20     --Functions=%30  --Exclude=main \
21     --MergeName=MERGE2 \
22     --MergeFlatten=true --MergeFlattenDispatch=
           goto \
23   --Transform=Merge \
24     --Functions=%30  --Exclude=main \
25     --MergeName=MERGE3 \
26     --MergeFlatten=true --MergeFlattenDispatch=
           indirect \
27   --Transform=AntiAliasAnalysis \
28     --Functions=*"
29 ...
30 tigress $TIGRESS_SETTINGS --Functions=$function
       $TIGRESS_SETTINGS2 --Functions=$function
       $TIGRESS_SETTINGS3 --Functions=$function
       $TIGRESS_SETTINGS4 --out=$virt_file $file
```

Listing 7: Tigress settings for Sample Set D

Table 5: The software and its dependencies used during the evaluation. The table also shows on which test system the software was installed. If no specific version can be given, the beginning of the git commit hash is listed.

Software	Source	Linux	Windows
Virtual Deobfuscator	[34]	-	de1131b
VMAttack	[25]	-	67dcce6
VMHunt	[47]	fcdadb9	fcdadb9
Tigress_Protection	[37]	79c6969	-
VMProtect	[9]	3.4.0	3.4.0
Tigress	[14]	2.2	-
IDA		-	6.8
Radare2		433b106	-
OllyDbg		-	2.01
NASM		-	2.14.02
PIN	[31]	3.11-97998	3.11-97998
gcc		4.8.4 / 7.5.0	7.4.0
mingw gcc		4.8.2 / 7.3.0	-
Cygwin	[35]	-	3.1.2
Visual Studio		-	16.4.3
Python		2.7 / 3.6	2.7
Z3	[17]	4.5.0	-
Capstone		4.0.1	-
Triton	[39]	fb3241e9	-
diStorm3		-	3.3.4
Cute		-	1.0.1
lxml		-	4.4.2
Arybo		1.0.0	-
LIEF		0.9.0	-
llvmlite		0.31.0	-

Table 6: The file size of the graded traces generated with VMAttack. The reported size is the result of "ls -la –block-size=MB" on the Linux machine.

	Sample Set A	Sample Set B	Sample Set C	Sample Set D
BKDR	4 MB	19 MB	4 MB	28 MB
BP	3 MB	18 MB	4 MB	54 MB
DEK	4 MB	20 MB	4 MB	20 MB
DJB	4 MB	17 MB	4 MB	87 MB
ELF	5 MB	25 MB	6 MB	97 MB
FNV	4 MB	20 MB	4 MB	30 MB
JS	4 MB	21 MB	4 MB	44 MB
PJW	5 MB	26 MB	6 MB	60 MB
RS	5 MB	20 MB	4 MB	54 MB
SDBM	4 MB	21 MB	5 MB	56 MB

Table 7: The analysis steps of Virtual Deobfuscator per sample set. Check marks indicate the given analysis step was successfully. A "-" symbol is shown if an error occurred.

	Trace Generation	Clustering	Repackaging	Peephole Optimization
Sample Set A	✓	✓	-	-
Sample Set B	✓	✓	-	-
Sample Set C	✓	✓	-	-
Sample Set D	✓	✓	-	-

Table 8: The analysis results of the modified sample sets. The numbers represent the number of samples successfully analyzed during the given analysis step. The maximum value is ten.

	Binary Reconstruction	Binary Comparison
Modified Sample Set A	10	9
Modified Sample Set B	0	-
Modified Sample Set C	10	9
Modified Sample Set D	10	8

Table 9: The files sizes of the traces generated by VMHunt. The reported file size results from the command "ls -la –block-size=MB" on the Linux machine.

	Sample Set A	Sample Set B	Sample Set C	Sample Set D
BKDR	18 MB	787 MB	18 MB	28 MB
BP	18 MB	754 MB	18 MB	40 MB
DEK	18 MB	800 MB	19 MB	23 MB
DJB	18 MB	683 MB	18 MB	55 MB
ELF	19 MB	728 MB	19 MB	62 MB
FNV	18 MB	796 MB	19 MB	28 MB
JS	18 MB	797 MB	19 MB	34 MB
PJW	19 MB	761 MB	19 MB	42 MB
RS	19 MB	781 MB	19 MB	39 MB
SDBM	18 MB	820 MB	19 MB	40 MB

Table 10: File size of the traces recorded with OllyDbg. The reported file size is the result of the command "ls -la –block-size=MB" on the Linux operating system.

	Sample Set A	Sample Set B	Sample Set C	Sample Set D
BKDR	4 MB	16 MB	4 MB	9 MB
BP	4 MB	13 MB	4 MB	10 MB
DEK	4 MB	14 MB	4 MB	8 MB
DJB	4 MB	14 MB	4 MB	11 MB
ELF	4 MB	18 MB	5 MB	11 MB
FNV	4 MB	14 MB	4 MB	8 MB
JS	4 MB	14 MB	4 MB	10 MB
PJW	4 MB	17 MB	5 MB	10 MB
RS	4 MB	14 MB	4 MB	10 MB
SDBM	4 MB	14 MB	4 MB	10 MB

Table 11: The analysis steps of VMAttack per sample set. A check mark represents a successful analysis step. A "-" symbol means the analysis of this step did not yield positive results. A "*" symbol indicates the result of the analysis step depends on the manual interpretation of the analysis results.

	Trace Generation	Grading Analysis	Manual Evaluation
Sample Set A	✓	✓	✓*
Sample Set B	✓	✓	- *
Sample Set C	✓	✓	- *
Sample Set D	✓	✓	✓*

Table 12: Section sizes used for the analysis during the test of Virtual Deobfuscator.

	Sample Set A	Sample Set B	Sample Set C	Sample Set D
BKDR	110	5000	800	4000
BP	900	300	700	5000
DEK	1080	5000	1300	4000
DJB	120	600	1000	500
ELF	1300	9000	2000	5000
FNV	990	500	1300	4000
JS	1080	6000	1300	400
PJW	1300	5000	300	7000
RS	1300	200	400	5000
SDBM	1080	6000	1000	400

Table 13: The analysis steps of VMHunt per sample set. A check mark means the analysis step was finished successfully, while a "-" symbol represents unsuccessful analysis steps.

	Trace Generation	Boundary Detection	Kernel Extraction	Symbolic Execution
Sample Set A	✓	-	-	-
Sample Set B	✓	-	-	-
Sample Set C	✓	-	-	-
Sample Set D	✓	-	-	-