2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-5278-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/SANER56733.2023.00082

xVMP: An LLVM-based Code Virtualization Obfuscator

Xuangan Xiao, Yizhuo Wang, Yikun Hu*, and Dawu Gu*
Shanghai Jiao Tong University, Shanghai, China
{xgxiao, mr.wang-yz, yikunh, dwgu}@sjtu.edu.cn

***Abstract*—Obfuscation techniques are widely used to protect the digital copyright and intellectual property rights of software. Among them, code virtualization is one of the most powerful obfuscation techniques, which hides both the control flow and the data flow of the code, thereby preventing code from being decompiled. However, existing code virtualization solutions are not well-resistant to de-obfuscation techniques (e.g., symbolic execution and frequency analysis), and only target limited program languages and architectures, which are challenging to integrate into the process of software development and maintenance.**

In this paper, We propose an LLVM-based code virtualization tool, namely xVMP to fulfill a scalable and virtualized instruction-hardened obfuscation. To mask the effects of multiple program languages and architectures, xVMP incorporates the obfuscation process of code virtualization into the compilation, and generates virtualized code based on LLVM intermediate representation (IR). After virtualization, it embeds the interpreter of virtualized code into the IR and compiles to an executable. To enhance the security, xVMP encrypts virtualized instructions in each basic block and decrypts them at runtime to enhance the security of obfuscation. In addition, it supports specified function obfuscation. xVMP identifies the function annotations marked by the developer in the source code to locate the function to protect. We implement the prototype of xVMP, and evaluate it with a microbenchmark and three real-world programs. The experimental results show that xVMP can be more difficult to crack than the state-of-the-art obfuscators, and it can support more source code types and architectures, and can be applied to real-world software. Source Code: https://github.com/GANGE666/xVMP.

***Index Terms*—Code Virtualization, Obfuscation, Reverse Engineering**

I. Introduction

Obfuscation techniques are widely applied to protect valuable assets from reverse engineering, such as business logic, encryption keys and secret algorithms. They usually perform well-known methods[1], such as control flow flattening, opaque predicates and white-box encryption to harden the control flow or dataflow of the target code. To defeat these obfuscations, a lot of work leverages symbolic execution, program synthesis, etc., to recover control and data flow.

To strengthen obfuscation, virtualization is used for obfuscation [1], which runs virtualized code on a virtual machine, and protects both control flow and data flow information. It transforms the to-be-protected code from its original Instruction Set Architecture (ISA) into a custom ISA, and embeds an interpreter in the program that emulates the new ISA. Translation from the original high-value logic and data operations into the virtualized instructions effectively invalidates analysis tools unfamiliar with the new ISA. Attackers have to reverse engineer the embedded virtual machine to understand the new instruction set architecture before they analyze the virtualized code.

To meet the requirements in real-world software development, the virtualization obfuscator should achieve the following three goals: i) Strong protection, which means that virtualized code should resist various de-virtualization techniques such as symbolic execution [2], frequency analysis [3], etc. ii) Compatibility, which means the virtualization obfuscator should support different source code types, platforms, instruction sets, and various code obfuscation schemes. iii) Ease of use, i.e., developers only need to make simple marks to complete the obfuscation.

However, state-of-the-art obfuscators cannot meet these three requirements. On the one hand, they cannot resist de-obfuscation techniques, such as frequency analysis. For example, DynOpVm [3] can infer the potential semantics of virtualized instructions by analyzing the frequency of opcode, which could break the protection of current obfuscators. On the other hand, current obfuscators are limited to specific architectures or program languages, which leads to unscalable in real-world software development. VMProtect [4] and Themida [5] only accept x86 and x64 binary executables as input, and are unable to apply to ARM, ARM64, etc. Tigress [6] only supports C language files, and requires developers to perform certain operations for each function to be protected, which is difficult to integrate into the process of development.

In this paper, we present xVMP, an LLVM-based code virtualization obfuscator fulfills these goals. It incorporates the obfuscation process of code virtualization into the compilation to mask the effects of different architectures and program languages. Specifically, xVMP identifies the to-be-protected functions according to developers' annotation and generates virtualized code based on *Low Level Virtual Machine* (LLVM) [7] intermediate representation (IR). After that, it embeds the interpreter of virtualized code into the IR. To evaluate its effectiveness, scalability, and compatibility, we applied xVMP on a microbenchmark and three real-world programs. Experiments show that xVMP's obfuscation is stronger than the state-of-the-art Tigress. xVMP can successfully protect C/C++ code and programs running on X86/64 and ARM32/64 architecture, The overhead of xVMP

* Yikun Hu and Dawu Gu are the corresponding authors.

2640-7574/23/$31.00 ©2023 IEEE
DOI 10.1109/SANER56733.2023.00082

virtualization obfuscation is less than 2% when protecting critical functions in non-hot loops.

The remainder of this paper is structured as follows. In Section § 2, we introduce the code virtualization obfuscation technique. We present the design and implementation of xVMP in Section § 3 and the evaluation results in Section § 4. We discuss related works in Section § 5, future work in Section § 6 and conclude the paper in Section § 7.

The source code of xVMP and instructions are available at: https://github.com/GANGE666/xVMP.

II. Background

In this section, we first introduce the basis of virtualization obfuscation, and then draw out our motivation according to the requirements for an obfuscator.

A. Virtualization Obfuscation

Virtualization-based software protection schemes protect the logic and data in the program by converting the original code into a new bytecode based on the custom instruction set architecture (ISA). The new bytecode is emulated via an embedded virtual machine (i.e. interpreter) at runtime. Since a lot of reverse engineering is required for attackers to learn the custom ISA, virtualization obfuscation could protect valuable assets in software. Specifically, a virtualization-based obfuscator contains the following three core components.

1) Virtual Instruction Set: The custom virtual instruction set is the key to virtualization obfuscation, which determines the complexity of obfuscation. Common ISA such as x86 and ARM already has many mature disassembly and decompilation solutions, and the new virtual instruction set can invalidate existing analysis tools. The new virtualization instruction set means that the obfuscator needs to design the type of operation, and the interaction with the outside of the virtual machine. And the instruction set may be different for each copy of the program, diversifying the code to prevent frequency analysis of opcodes [3].

2) Translator: The translator in the obfuscator is used to convert the original code into virtualized code. The input of the translator is generally expected to be more universal, so as to support more scenarios or more obfuscation schemes. For example, Tigress [6] uses CIL [8] as the input of the translator. At the same time, the translator needs to ensure that the semantics of the input original code and the output virtualized code remain unchanged.

3) Virtual Machine (Interpreter): The interpreter is embedded in the obfuscated program to emulate virtualized code and it has virtual registers and virtual memory. The design of the interpreter is close to a CPU, and it is divided into five steps: fetching instructions, decoding instructions, dispatching according to the opcode, processing in the corresponding handler, and determining whether the emulation is completed or not. The interpreter is the entry point for an attacker to perform reverse engineering. The attacker needs to analyze the logic of the interpreter to understand the custom instruction set. So the interpreter needs to be protected by traditional obfuscation schemes to protect the mapping of virtual instruction sets to native instructions. However, virtualization obfuscation will cause high overhead, so usuall, only a few important logic functions are selected for virtualization obfuscation.

B. Motivation

To well-protect valuable assets in software, a virtualization obfuscator should be resistant to various de-obfuscation techniques, and compatible with different source code types as well as architectures. In addition, to improve the strength, an obfuscator should support multiple obfuscation schemes.

Unfortunately, the state-of-the-art virtualization obfuscators cannot fulfill the above requirements. On the one hand, they lack effective protection for virtualization instructions (Tigress only xors a constant for all instructions) and many works in recent years have broken through them by using frequency analysis [3] and symbolic execution [2]. On the other hand, VMProtect [4] and Themida [5] only support virtualize executables on X86 and X64 architectures, and they are not open source. The most advanced academic virtualization scheme Tigress [6] can only support C language source code, and since its input must be a single file containing the full program source code, it is difficult to blend in the software development cycle. This motivates our work.

III. Design and Implementation

In this section, we first explain the overview of xVMP, then describe the details of its design. After that, we explain the implementation and usage of xVMP.

A. Overview

In this paper, we present xVMP, an LLVM-based virtualization obfuscator to fulfill the three goals, i.e., ease of use, scalability, and compatibility. Figure 1 depicts the workflow of xVMP. It accepts the IR of source code as input, which contains simple function annotations added by developers to indicate the functions to be protected. It then performs the following steps to generate virtualized IR:

1) **Target Functions Locating.** xVMP first compiles the input source code into the LLVM IR, and identifies to-be-protected functions according to the function annotation.
2) **IR Code Virtualizing.** xVMP then translates the IR of target functions into a semantically equivalent virtualized code, and encrypts the virtualized code.
3) **Interpreter Embedding.** xVMP embeds the interpreter into the IR module and modifies the calling convention, including incoming parameters and the return value.
4) **Virtualized Code Encryption.** xVMP generates an independent key for each basic block and encrypts the instructions in the basic block.

Finally, the compiler performs traditional obfuscation or subsequent optimization on the output IR module, and generates code to obtain the executable file.

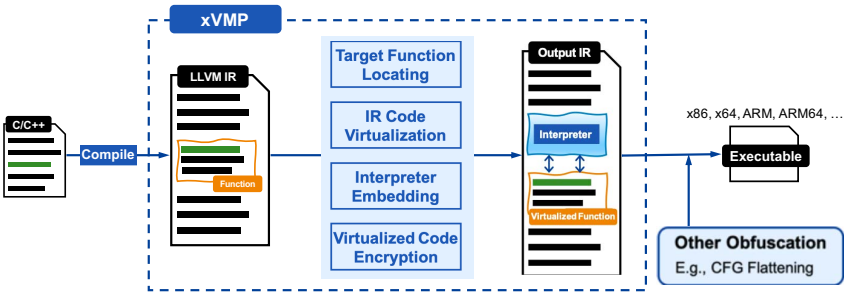

Fig. 1: The workflow of xVMP, which incorporates the obfuscation process into the compilation. Multiple obfuscation schemes are supported to add to the output IR for obfuscation strengthening.

B. Virtual instruction set

We design a Turing-complete virtual instruction set, which contains ten types of basic operations, and its instruction length is variable. Instruction types include *variable allocation*, *load*, *store*, *binary operation*, *pointer operation*, *compare*, *type conversion*, *branch*, *call* and *return*. In the demo, we only designed basic operations, but these basic operations can also be easily split and expanded. The instruction set is based on dummy variables, and the location, bit width and type of the dummy variables are packed in the value field of the instruction for the interpreter to analyze at runtime. To facilitate the operation of native variables, we set three types of values: virtual machine variables, native variables, and constants.

C. Translator

The translator of xVMP, translates the IR instructions of the to-be-protected function into instructions in the virtual instruction set. However, since the length of instruction is variable, the translator needs to leave a hole when it encounters a branch instruction, and fill the target address into the hole after all instructions are translated.

D. Interpreter

The interpreter used to execute the virtualization code is written in C language. After compiling it into IR, it will be embedded into the IR of the program together with the virtualized code. Subsequently, xVMP will modify the calling convention of the to-be-protected function, write the incoming parameters into the corresponding dummy variables, take the return value out of the corresponding dummy variables when the virtualized code returns, and finally delete the original IR of the function. To support calling native functions from virtualized codes, the interpreter adopts a dynamic distribution method, extracts the required parameters from virtual variables, and constructs a native calling convention to call native functions.

E. Security Enhancements

To prevent de-virtualization techniques such as symbolic execution, frequency analysis, etc., we protect virtualization instructions in two ways. i) Firstly, we use n-to-1 opcode mapping to increase the diversity of virtual instructions, that is, each type of operation can correspond to multiple opcodes.

```
1  __attribute((__annotate__("vmp")))
2  void gen_deckey (unsigned char *decrypt_key, char *passwd) {
3    // generate decrypt key from the password
4    // ...
5  }
```

Fig. 2: An annotation example in the source code.

ii) Secondly, we encrypt virtualized instructions in each basic block, and dynamically decrypt the instructions at runtime. The translator sets an independent key for each basic block, uses this key to encrypt all instructions in the basic block, and embeds the key in the header of the basic block. When the interpreter executes to a basic block, it reads the key and decrypts the instruction being executed.

F. Implementation and Usage

xVMP is implemented upon LLVM 8.0.0. We implement the main part of xVMP in a pass of LLVM, including the translator and IR modification. The LLVM pass is implemented in about 1800 lines of C++, and the interpreter part is implemented by 700 lines of C. Besides that, we have about 1000 lines of code for test cases.

To apply virtualization obfuscation using xVMP, you only need to add an annotation at the declaration or definition of the to-be-protected function (as shown in Figure 2). However, because virtualization protection will introduce a large overhead, for performance considerations, key data processing functions before the hotspot loop, such as key generation functions, should be obfuscated.

IV. Evaluation

In this section, we report the results of the following three experiments. First, we show how xVMP performs against de-virtualization. Then we present the compatibility for different architectures, source codes and other obfuscation schemes. Finally, we evaluate the overhead.

A. Experiment Setup

We ran experiments on a server running Ubuntu 20.04 with two AMD Ryzen Threadripper 3970X and 252GB RAM.

We use two sets of programs to evaluate xVMP. The first set is a microbenchmark consisting of five C and three C++ sample programs. Because the state-of-the-art open source

TABLE I: De-obfuscation for the microbenchmark protected by Tigress and XVMP

Case	Original Ins Count	Tigress Virtualized		XVMP Virtualized		Deobf Ins Count
		Obf Ins count	Deobf time(s)	Obf Ins count	Deobf time(s)	
Sample 1	53	1,956	0.13	9,859	361.35	45
Sample 2	57	1,916	0.12	15,872	Out of time[1]	46
Sample 3	99	22,636	1.45	133,632	Out of time[1]	57
Sample 4	330	31,511	2.09	186,655	Out of time[1]	126
Sample 5	231	60,425	3,729	384,529	Out of time[1]	122

[1] The de-virtualization tool cannot simplify the obfuscated program within 10 hours.
[2] The instructions are counted via the dynamic taint analysis engine Triton.

virtualization obfuscation scheme Tigress [6] can only accept a single C file as input, and the automatic deobfuscation scheme [2] is difficult to apply to more complex programs, we use this set for obfuscation strength and compatibility evaluation. The second set contains 3 real-world software, including the command-line encryption program ccrypt-1.11, the widely used cryptography library OpenSSL-1.0.2u, and the database leveldb-1.23 written in C++, and we use them to evaluate the overhead of XVMP.

In the experiments, XVMP is mainly compared with the state-of-the-art open-source virtualization obfuscator Tigress-3.1.

B. Strength of XVMP Obfuscation

In this experiment, we use an automated de-virtualization tool to analyze the obfuscated programs of XVMP and Tigress to evaluate the strength and effectiveness of XVMP. Since most of the automatic de-virtualization work is outdated and unable to work [9], we chose Salwan's work [2] and patched the sample to make it work normally. It is based on combining taint tracking, symbolic execution and code simplification.

We used Tigress and XVMP to virtualize five C samples in the microbenchmark, and tried to de-virtualize them. We counted the number of instructions executed by the original program, the obfuscated program, and the deobfuscated program to evaluate the complexity. The results are shown in Table I. It can be seen that all the obfuscated programs of Tigress have been cracked, while only one sample of XVMP has been cracked, and the rest of the samples failed within a limited time. Because XVMP encrypts the virtualization code at the granularity of basic blocks, it greatly increases the complexity of the obfuscator for symbolic execution, making it difficult to simplify the virtualization code.

This experiment shows that the obfuscation by XVMP is difficult to be simplified by the automatic deobfuscator, and its obfuscation is stronger than Tigress.

C. Compatibility

In this experiment, we evaluate XVMP's compatibility with different types of source code, different architectures, and traditional obfuscation.

We set up three C++ programs in the microbenchmark, which were implemented using classes, inheritance and polymorphism, and function overloading in object-oriented programming. XVMP can effectively protect the C++ functions which annotated in the source code, and the virtualized functions have the same semantics.

To test the compatibility of multiple architectures, we obfuscated all C and C++ files in microbenchmark with XVMP, compile them into executable files of X86, X64, ARM32, and ARM64 architectures, and use Qemu [10] to check the program behavior. The results show that the obfuscated executable files generated by all the test files in the microbenchmark can run normally under these four architectures.

To prove that XVMP is compatible with the traditional obfuscation scheme based on LLVM IR, we set the effective order in the pass manager, so that the obfuscation option in Obfuscator-LLVM can be used to protect the XVMP interpreter. We obfuscated the program with options of control flow flattening and bogus control flow. The results show that the program protected by XVMP can still run normally after being protected by Obfuscator-LLVM.

D. Overhead

We test the overhead of XVMP using three real-world pieces of software. For OpenSSL, we tested the average overhead of one million 1K-bytes block encryption by AES-CBC-128 using OpenSSL speed. We use ccrypt to encrypt 1MB of random data and use c_test in the leveldb test suite.

The results are shown in Table II, We select one function at a time for virtualization obfuscation, including two hotspot functions and three non-hotspot functions. We can see that when protecting the hotspot loop functions `xKeyAddition` and `CRYPTO_cbc128_encrypt` in ccrypt and OpenSSL, the overhead will be tens to hundreds of times. However, in real-world applications, virtualization obfuscation is more used to protect key generation and key logic, such as the function `hashstring` in ccrypt that calculates symmetric keys through passwords. Therefore, we protect three non-hot but critical functions with less than 2% overhead. Because Tigress does not support C++, it cannot protect leveldb.

We tested the code bloating effect brought by XVMP. Because the interpreter is embedded into the program, the program size will expand, but the actual code size increased by XVMP is less than 40K.

V. Related Works

As one of the strongest obfuscation schemes, virtualization obfuscation includes VMProtect [4] and Themida [5], which have been successfully commercialized, as well as the state-of-the-art Tigress [6] in academia. VMProtect and THEdima

TABLE II: Application Overhead under protection of Tigress and XVMP

Case	Obfuscated Function	Original Program		Tigress Virtualized		XVMP Virtualized	
		time(s)	size(KB)	time(s)	size(KB)	time(s)	size(KB)
ccrypt	`xKeyAddition`[1]	0.10	77	0.41(4x)	150	4.23(43x)	102
openssl	`CRYPTO_cbc128_encrypt`[1]	3.08	2,735	72.63(24x)	2,741	190.96(616x)	2,769
ccrypt	`hashstring`	0.096	77	0.097(1%)	150	0.098(2%)	102
openssl	`idea_set_encrypt_key`	8.27	2,735	8.29(1%)	2,741	8.30(1%)	2,769
leveldb[2]	`leveldb_get`	2.59	668	Not support[2]	Not support[2]	2.63(1%)	692

[1] `xKeyAddition` and `CRYPTO_cbc128_encrypt` are hotspot functions, so the overhead of virtualization is much higher than non-hotspot functions.
[2] leveldb is a C++ project and Tigress is not support C++.

can only protect X86 and X64 binary programs, and because they are not open source, they are a black box for developers and difficult to control. Tigress can only accept a whole program single C file as input, which means that the original compilation process needs to be broken and is very unfriendly to developers. Moreover, these obfuscators lack effective protection for virtualization instructions.

XVMP is implemented based on LLVM IR, which is not only effectively compatible with more source code types and architectures, but also implements stronger protection for virtualized instructions to resist automatic de-obfuscation. Moreover, it's developer friendly.

In addition to virtualization obfuscation, static obfuscation schemes also include control flow flattening, bogus functions, mixed boolean-arithmetic (MBA), etc. Obfuscator-LLVM [11] is one of the most popular open-source obfuscators, providing obfuscation options including opaque predicates, bogus control flow, and control-flow flattening. It also works on top of LLVM IR, and provides function annotations for developers to mark. However, there are already a lot of automated de-obfuscation works for Obfuscator-LLVM [12], [13], which threatens the security of Obfuscator-LLVM protection.

XVMP also works on LLVM IR, which is well compatible with the obfuscation scheme on Obfuscator-LLVM and provides stronger virtualization protection.

VI. Future Work

There exist many features for future additions to XVMP. For example, the types of virtualization instructions can be expanded to further enhance the diversity of instructions and make the interpreter more complex. Due to the high overhead of the current solution, we will optimize the performance without compromising security. Because virtualization obfuscation cannot be used to harden the entire program, due to the overhead, we also plan to introduce anti-debug detection [14], [15] to prevent attackers from spying on the program at runtime. We also plan to add support for more LLVM IR instructions, such as LLVM intrinsics.

VII. Conclusion

In this paper, we describe the design and implementation of XVMP, an LLVM IR-based virtualization obfuscator that enhanced the security of virtualized instructions. The implementation based on LLVM IR makes our obfuscator compatible and scalable. The design based on instructions dynamic decryption makes our obfuscator stronger. Our evaluation also shows that developers can easily use XVMP to harden real-world software. We will make the tool available as open source for future research.

Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable feedback to improve our manuscript. This work is partially supported by the National Key Research and Development Program of China (No. 2021YFB3101402).

References

[1] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.
[2] J. Salwan, S. Bardin, and M.-L. Potet, "Symbolic deobfuscation: From virtualized code back to the original," in Detection of Intrusions and Malware, and Vulnerability Assessment, Cham, 2018, pp. 372–392.
[3] X. Cheng, Y. Lin, D. Gao, and C. Jia, "Dynopvm: Vm-based software obfuscation with dynamic opcode mapping," in Applied Cryptography and Network Security: 17th International Conference, ACNS 2019.
[4] VMProtect Software, "Vmprotect," https://vmpsoft.com/, 2022, [Online; accessed 8-Nov-2022].
[5] Oreans Technologies, "Advanced windows software protection system," https://www.oreans.com/Themida.php, 2022, [Online; accessed 8-Nov-2022].
[6] Christian Collberg, "The tigress c diversifier/obfuscator," http://tigress.cs.arizona.edu/, 2022, [Online; accessed 8-Nov-2022].
[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," San Jose, CA, USA, Mar 2004, pp. 75–88.
[8] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in Compiler Construction, R. N. Horspool, Ed., Berlin, Heidelberg, 2002.
[9] P. Kochberger, S. Schrittwieser, S. Schweighofer, P. Kieseberg, and E. Weippl, "Sok: Automatic deobfuscation of virtualization-protected applications," in Proceedings of the 16th International Conference on Availability, Reliability and Security, ser. ARES 21, New York, NY, USA, 2021.
[10] Qemu org, "Qemu," https://www.qemu.org/, 2022, [Online; accessed 8-Nov-2022].
[11] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, 2015.
[12] R. Tofighi-Shirazi, I.-M. Asavoae, P. Elbaz-Vincent, and T.-H. Le, "Defeating opaque predicates statically through machine learning and binary analysis," ser. SPRO'19, 2019.
[13] Kareem El-Faramawi and Toshi Piazza, "llvm-deobfuscator," https://github.com/RPISEC/llvm-deobfuscator, 2017, [Online; accessed 8-Nov-2022].
[14] H. Cho, J. Lim, H. Kim, and J. H. Yi, "Anti-debugging scheme for protecting mobile apps on android platform," J. Supercomput., vol. 72, no. 1, p. 232–246, jan 2016.
[15] M. Schallner, "Beginners guide to basic linux anti anti debugging techniques," Code Breakers Magazine, vol. 1, 2006.