**Abstract**

**Virtualization Obfuscation in Conjunction With Alternate Schemes for Binary Obfuscation**

Henry Eshbaugh

henryeshbaugh@gmail.com

Often, an individual may desire to recover the higher-level structure of a computer program from a lower-level representation of the program. Binary representations of computer programs may be converted into assembly-language representations on a 1-to-1 basis (i.e., each assembly-language representation of an instruction corresponds to a given bit pattern) [1]. The assembly-language representation is commonly used to recover higher-level functioning in a process called reverse engineering, using a technique known as disassembly to transform the binary into an assembly-language program [1]. This practice is undesirable for a multitude of reasons [2], as it may lead to copyright infringement or the development of exploits against a target program. The practice of mitigation of reverse engineering is known as binary obfuscation [1]. Essentially, binary obfuscation is the process of making difficult the process of comprehending the obfuscated program. In this paper, I propose a previously-developed technique, virtualization obfuscation, as being more viable than generally thought.

**Keywords**

Obfuscation, Reverse Engineering, Disassembly, Virtualization

**1 Introduction**

There are two broad categories of representations of computer programs - the program’s *source-code representation,* which is human-readable, and the program’s *binary representation*, which is executable by a computer [1]. Source-code representations are converted by a compiler at compile-time or by an interpreter at run-time into a binary representation. In general, programs are only distributed in binary form. However, many tools, such as GNU Objdump [3], GDB [4], Evan Tehran’s EDB [5], and Oleh Yuschuk’s OllyDbg [6] are capable of producing assembly-language representations of a computer program from the binary representation of the program. One manner in which a program may be obfuscated is in producing a binary which, though executable, is impossible to render in assembly - that is, an attempt is made to confuse the assembler by causing it to attempt to disassemble bogus data. [1]

One method of generating a binary which is impossible to disassemble yet executable is by arbitrarily defining a computer architecture and implementing an emulator for that platform. This method, known as *virtualization obfuscation*, renders control-flow and semantic operations in a representation unrecognizable by production disassemblers by sheer virtue of the architecture being arbitrarily defined [7, 8, 9]. Further, virtualization obfuscation is unique in that it is not inherently incompatible with any other obfuscation scheme. For example, it is impossible to reconcile the inherent incompatibility between László’s code-flow-flattening scheme [10] with Popov’s branch-by-trapping scheme [11], as flattening code flow reduces the number of jumps, making Popov’s scheme moot. Virtualization obfuscation is unique in that it poses no innate incompatibilities with any other obfuscation scheme, and, in fact, may even resolve such incompatibilities.

The remainder of the paper is organized as follows - Section 2 lays the groundwork for an understanding of virtualization obfuscation, and section 3 explores the use of virtualization obfuscation in conjunction with alternate strategies for obfuscation.

**2 Virtualization Obfuscation**

**2.1 Overview**

Virtualization obfuscation is a straightforward method of obfuscating computer programs, utilizing the inherent resilience of emulation to complicate the process of reverse-engineering a binary program. This inherent resilience stems from the separation of code and data from the emulator proper and the rendering of the code and data in a format undecipherable by production disassemblers. The code and data are undecipherable by disassemblers, of course, because the instruction set and architecture is arbitrarily defined. Thusly, no production assembler is equipped to render in an assembly language the obfuscated code.

**2.2 Methodology**

Virtualization obfuscation is fairly straightforward. Firstly, an instruction set and architecture (ISA) must be defined. This includes several attributes of the processor, such as whether constant-length or variable-length opcodes are used, addressing modes, and methods by which system calls may be made [12]. Secondly, an emulator must be written. Emulators are, again, fairly straightforward to write. An example emulator is provided in figure 1.

|  |
| --- |
| /\* ... \*/  #define NOP 1  #define ADD 2  #define SYSCALL 3  extern int ip, reg, imm, registers[];  extern char opcodes[];  while (opcodes\_to\_be\_digested()) {  switch(opcodes[ip++]) {  case NOP:  break;  case ADD:  reg = eat\_next\_reg();  imm = eat\_next\_imm();  registers[reg] += imm;  break;  case SYSCALL:  imm = eat\_next\_imm();  switch (imm) {  /\* ... \*/  }  break;  }  anti\_debug();  }  /\* ... \*/ |

***Figure 1: An Emulator, in C.***

Thirdly, a technique for generating binary representations of computer programs for the platform must be defined. Programs may be hand-compiled, an assembler may be written, or a compiler such as GCC or Clang may be ported. Assemblers are trivial to implement using lexer/parser generators such as Bison [14] and Flex [13] or ANTLR [reference], and many compiler architectures, such as LLVM/Clang [16], have been designed for easy retargeting [16].

**2.3 Portability considerations**

Programs written for emulators are as portable as the emulators themselves [17]. As well, an emulator may be written quite easily using only functions from libc, allowing for the emulator to be ported with ease to any platform that conforms to the C standard. This is due to the inherent platform agnosticism of virtualization obfuscation - at most, a small emulator may only require features as advanced as binary file I/O via the libc functions fread() and fwrite() - functions which are supported on all major platforms. Thusly, a virtualization-obfuscated program is easily portable to new platforms.

**2.4 Effectiveness**

Virtualization-obfuscated programs are quite difficult to reverse. Coogan et al. observe that virtualization obfuscation is difficult to comprehend statically [7]. This, Coogan furthers, is due to the limited amount of ways to approach disassembling virtualization-obfuscated code. This is due to the fact that code-flow statements are inherently undisassembleable by production disassemblers – for example, when supplied with an arbitrary binary, objdump [3] emits an error – the file format is not recognized. Other disassemblers emit garbage, as the disassemblers are attempting to interpret the bytecode as instructions for a different architecture. Thusly, other methods of disassembling virtualization-obfuscated programs have been developed. Such approaches include static analysis of the emulator in order to decipher the behavior of the emulator in response to opcodes [9], and using tools such as qemu [18] and Ether [19] to generate traces of system calls and arguments made by the target program, a technique developed by Coogan et al. [7]. Coogan’s technique is easily mitigated through the use of anti-debug code, however, which causes the program to behave differently when a foreign binary is tracing the execution of a program [1]. Virtualization obfuscation specifically lends itself to the use of anti-debug code, as loops play an integral part in the execution of virtualization-obfuscated binaries, allowing anti-debug code to be executed periodically. However, great progress has been made in techniques to aid the static analysis of computer binaries. For example, a technique has been developed by Kinder for the automated static analysis of a virtualization-obfuscated binary [9]. To summarize, though virtualization obfuscation may be resilient, progress has been made in neutralizing the technique.

**3 Use of Virtualization Obfuscation in Conjunction With Other Obfuscation Strategies**

**3.1 Overview**

Virtualization obfuscation is unique in that the technique is not mutually exclusive with any other techniques. These techniques may be employed to dramatically complicate the process of reverse-engineering an obfuscated program. Indeed, virtualization obfuscation often resolves incompatibilities through its separation of the program into two elements. As an example, László’s code-flow-flattening scheme [10] may be used to obfuscate the emulator, while Popov’s branch-by-trapping scheme may be adopted for the flow of the obfuscated code [11]. Thusly, virtualization obfuscation resolves inherent incompatibilities between schemes of obfuscation. This increases the difficulty of statically analyzing a binary.

**3.2 Methodology**

The methodology of obfuscating a program in this way is highly dependent on the techniques used to obfuscate a program. Obfuscation techniques should be applied to components of the application as a whole. For example, a packer such as UPX [20] may be applied to the emulator segment of the program, while alternate obfuscation schemes, such as Frank^2’s code-flow obfuscation scheme [21], can be applied to the obfuscated code. In addition, as the code is represented as data, ciphers may be applied to the obfuscated code. This once again expands the power of virtualization obfuscation - for example, each byte of a program may be XOR-ciphered against an arbitrary value (e.g. 0x7B) to produce a further-obfuscated program. In this way, a program may be obfuscated to an arbitrary degree.

**3.3 Portability Considerations**

A binary obfuscated by virtualization and other techniques will be limited in portability, depending on the methods used. For example, if a program is relying upon the use of NtSetInformationThread() to hide threads from a debugger [22], the program will be restricted to running on Windows, as NtSetInformationThread(), obviously, is unique to the Windows API. Furthermore, as whether a thread is hidden may be queried in versions of Windows since Vista, this technique is only applicable on machines running Windows XP and before. Likewise, a program relying upon the sigaction API must, if Windows is a desired target platform, be either linked with some library which implements sigaction(), such as Cygwin [23], or a custom implementation of the sigaction API.

**3.4 Effectiveness of Virtualization Obfuscation In Conjunction With Alternate Obfuscation Strategies**

The reverse-engineering of a computer program which has been obfuscated by a combination of techniques is exceptionally difficult, as the time needed to reverse-engineer a binary obfuscated with multiple techniques grows dramatically with each additional technique used, as the use of particular styles of obfuscation can further complicate the process of defeating any given obfuscation technique (e.g. code-flow flattening may prevent a reverse engineer from recognizing that the binary is unpacking itself). These techniques may also be applied to hide anti-debug code, which handily prevents a reverse engineer from generating instruction traces, and may in turn be concealed by obfuscation (e.g. on Mac OS X, using various mathematical identities to calculate the value of the PT\_DENY\_ATTACH, then preventing foreign binaries from attaching to the process in question using PT\_DENY\_ATTACH [24]).

**4 Conclusions**

Utilizing virtualization obfuscating in conjunction with alternate obfuscation strategies allows one to easily create binary representations of computer programs which are extraordinarily difficult to reverse engineer. This idea capitalizes on the fact that virtualization obfuscation is inherently compatible with any proposed technique for binary obfuscation – its potency is not diminished by its use in conjunction with alternate techniques.

As well, the obfuscation of virtualization-obfuscated software via alternate means while utilizing anti-debug code offers a platform extraordinarily resilient to the conventional approach to reverse-engineering virtualization-obfuscated binaries, i.e., tracing execution. This forces the reverse engineer to confront the binary through static analysis, a technique made exceptionally difficult through the obfuscation of the emulator and virtualized operations.

Overall, virtualization obfuscation is an incredibly powerful technique which may easily be augmented by the use of alternate obfuscation techniques.

**5 References**

[1] Eliam, Eldad. *Reversing: Secrets of Reverse Engineering.* Eliam offers an introductory text on the practice of reverse engineering, discussing legal and ethical ramifications as well as technical information and terminology.

[2] Gomulkiewicz, Robert W. *The Problem of Reverse Engineering.* Gomulkiewicz discusses the ethical implications of the practice of reverse engineering.

[3] *GNU Binutils.* <https://www.gnu.org/software/binutils/>. A description of GNU binary utilities.

[4] *GDB: The GNU Project Debugger.* <https://www.gnu.org/software/gdb/>. A description of GDB.

[5] Evan Tehran. *EDB (Evan’s Debugger)*. <http://freecode.com/projects/edebugger>. A Qt4-based plugin-architecture debugger.

[6] Oleh Yuschuk. *OllyDbg*. <http://www.ollydbg.de/>. Often considered the seminal freeware debugger for reverse engineering.

[7] Coogan, Kevin, et al. *Deobfuscation of Virtualization-Obfuscated Software.* A description of techniques to deobfuscate virtualization-obfuscated programs.

[8] Rolles, Rolf. *Unpacking Virtualization Obfuscators.* Rolles offers an overview of techniques used to reverse engineer virtualization-obfuscated software.

[9] Kinder, Johannes. *Towards Static Analysis of Virtualization-Obfuscated Binaries.* 19th Working Conference, Reverse Engineering, Kingston, Ontario. A description of techniques used to statically analyze virtualization-obfuscated code.

[10] László, Tímea, and Kiss, Ákos. *Obfuscating C++ Programs Via Control Flow Flattening.* University of Szeged, Department of Software Engineering. László and Ákos describe a technique for obfuscating programs by flattening code flow.

[11] Popov, Igor V, et al. *Binary Obfuscation Using Signals.* 16th USENIX Security Symposium. Popov outlines techniques for binary obfuscation using unconditional branching caused by trapping.

[12] Bowan, Johnathan. *Standard Microprocessor Programming Cards.* https://www.sciencedirect.com/science/article/pii/0141933185901164. Bowan details the process of microprocessor design.

[13] GNU Software Foundation. *Flex – A GNU Lexer Generator.* <https://www.gnu.org/software/flex/>*.* Overview of the Flex lexer generator.

[14] GNU Software Foundation. *Bison – A GNU Parser Generator.* <https://www.gnu.org/software/bison/>. Overview of the Bison parser generator.

[15] Parr, Terrence. *ANTLR – ANother Tool for Language Recognition.* <http://www.antlr.org/>. Overview of the ANTLR lexer/parser generator.

[16] Apple, Inc. *Clang: A C Language Family Frontend for LLVM.* <http://clang.llvm.org/>. Overview of the clang compiler.

[17] Roulo, Mark. *Java’s Three Types of Portability.* <http://www.javaworld.com/javaworld/jw-05-1997/jw-05-portability.html>*.* A discussion of the portability of virtualized programs.

[18] Bellard, Fabrice. *QEMU: A Generic and Open-Source Machine Emulator.* <http://wiki.qemu.org/Main_Page>. An overview of the qemu emulator.

[19] Dinaburg, Artem, et al. "Ether: malware analysis via hardware virtualization extensions." *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008. Dinaburg details the process of malware analysis using tracing tools.

[20] Oberhumer, Markus. *UPX: The Ultimate Packer for eXecutables.* <http://upx.sourceforge.net/>. Overview of the UPX packer.

[21] Frank^2. *Trolling Reverse Engineers with Math.* DEF CON 18. <https://www.defcon.org/images/defcon-18/dc-18-presentations/frank%5E2/DEFCON-18-frank%5E2-Trolling-With-Math.pdf>. Overview of the process by which a program’s control flow may be obfuscated.

[22] Shields, Tyler. *Anti-Debugging Series – Part III.* Veracode. An overview of obfuscation techniques utilizing the Windows API functions NtSetInformationProcess() and NtSetInformationThread().

[23] Cygwin Project. *Cygwin: A Linux API Layer for Windows.* <http://www.cygwin.com/>*.* An overview of the Cygwin project.

[24] Renard, Mathieu. *Hacking iOS Applications.* <http://reverse.put.as/wp-content/uploads/2011/06/Mathieu-RENARD-Hack.lu-Hacking-iOS-Applications.pdf>. An overview of multiple Darwin binary obfuscation techniques and how to defeat them, including the usage of PT\_DENY\_ATTACH.