Program Fragmentation as a Metamorphic Software Protection

Bobby D. Birrer, Richard A. Raines, Rusty O. Baldwin, Barry E. Mullins

Center for Cyberspace Research
Air Force Institute of Technology
{firstname.lastname@afit.edu}

Robert W. Bennington

Anti-Tamper Software Protection Technology Office
Air Force Research Laboratory
{Robert.bennington@wpafb.af.mil}

Abstract

Unauthorized reverse-engineering of programs and algorithms is a major problem for the software industry. Reverse-engineers search for security holes in the program to exploit or try to steal competitors' vital algorithms. To discourage reverse-engineering, developers use a variety of static software protections to obfuscate their programs.

Metamorphic software protections add another layer of protection to traditional static obfuscation techniques, forcing reverse-engineers to adjust their attacks as the protection changes. Program fragmentation combines two obfuscation techniques, outlining and obfuscated jump tables, into a new, metamorphic protection. Sections of code are removed from the main program flow and placed throughout memory, reducing the program's locality. These fragments move and are called using obfuscated jump tables, making program execution difficult to follow.

This research assesses the performance overhead of a program fragmentation engine and provides analysis of its effectiveness against reverse-engineering techniques. Results show that program fragmentation has low overhead and is an effective technique to complicate disassembly of programs using two common disassembler/debugger tools.

1. Introduction

One of the most daunting problems facing software developers is piracy and the theft of intellectual property. Whenever a program is released, it almost immediately comes under attack by those trying to reverse-engineer the program. Whether these reverse-engineers are simply curious about proprietary

algorithms, or want to find vulnerabilities for future exploitation, the general method is the same. The reverse-engineer examines the binary file, converts the machine code to assembly language code, and interprets the assembly language code to determine how the program operates. From there, high-level source code can possibly be generated. With this knowledge, the reverse-engineer can either create competing software at a fraction of the development cost, or develop exploits to attack the software and cause it to fail.

To thwart reverse-engineers, software developers use software protection and obfuscation techniques. These techniques take the executable code and change it in ways that make it harder to understand and reverse-engineer while still maintaining functionality. Given enough time and resources, even a protected program can eventually be reverse-engineered. The goal of software protection and obfuscation is to make the reverse-engineering process more costly than developing the program separately. The current software protection arena constitutes an "arms race," where developers create new protections which reverse-engineers create new tools to break and bypass.

Current protections methods primarily consist of static protections such as encryption, obfuscated control paths, outlining, and opcode shifts [1]. Protections may be designed to target the disassembly process to prevent the reverse-engineer from seeing the correct assembly instructions. Protections can also target program debugging by preventing breakpoints from working. Protections can even target the reverse-engineers themselves by eliminating context clues, removing abstractions, and reducing the locality of the program. These methods each have associated strengths and weaknesses for protecting against different types of reverse-engineering attacks.

However, they all share the disadvantage that they do not change over the course of the program execution. This provides a reverse-engineer with a “stationary target” that is easier to methodically analyze and defeat than a target that moves or changes. Metamorphic protections are a new category of software protections that require reverse-engineers to adapt as the protections change over the course of the program execution.

Our research develops a novel metamorphic technique, program fragmentation, which increases the difficulty of the analysis process and also creates problems for disassemblers and debuggers. We determine the overhead associated with the protection and examine how effective it is at increasing the difficulty of reverse-engineering.

2. Background

Using metamorphism to provide obfuscation for legitimate programs is a new concept, even though virus and other malware writers have been using it for some time to avoid detection programs. Metamorphic engines in malware, however, change the program just enough between propagations to avoid detection by signature-based tools [2]. Commonly suggested metamorphic techniques include instruction substitution/reordering, changing conditionals, “garbage” instruction insertion, dynamic opcode shifts, and subroutine reordering [2, 9].

Metamorphism has several advantages over static protection techniques. Protections that are different and constantly changing make it difficult for an automated tool to simply search the program for certain protection methods and remove them. Dynamic protections such as encryption also make it harder for automated tools to analyze the program without actually running it, slowing down the reverse-engineering process.

Reverse-engineers assume certain paradigms are followed when programs are written, one of which is that programs do not normally modify themselves. Therefore, most tools are not designed to successfully handle metamorphic programs. Metamorphic programs are especially useful in confusing debuggers, as changing the program often corrupts software breakpoints. The program can write over the 0xCC byte inserted by the debugger for a software breakpoint as the protection modifies the program, keeping the debugger from regaining control at the expected location. Breaking the expected paradigm also has a significant effect on a human reverse-engineer, simply because they now have to reverse a constantly changing program. This greatly increases not only the

complexity of the program, but also the annoyance factor of the reverse-engineer [2, 9].

Metamorphic programs also receive an additional benefit, as they are potentially less vulnerable to buffer-overflows. Previous research [4] shows that buffer-overflow attacks can be thwarted by slightly different versions of the program. Buffer-overflows rely on injecting a very precise series of bytes to overflow the buffer and jump to the desired location in the target program. Any changes to the target that modify the location of functions and subroutines can prevent the buffer-overflow from working correctly. While [4] focused on generating a large number of differing programs by inserting static instructions, the same effect could be obtained through metamorphic transforms. As the program constantly changes during execution in a possibly non-deterministic manner, it is almost impossible for a buffer-overflow writer to generate a correct string of data to overflow the buffer.

While metamorphism offers strong obfuscation, so far little research has been done on the topic. At the time of this writing, there are only a handful of papers on metamorphic transforms for code obfuscation. The first, a paper by Yuichiro Kanzaki et al., describes a technique in which legitimate instructions are replaced with junk instructions and restored immediately before execution. Once the instructions are executed, they are once again replaced with junk instructions. While reporting that execution time of a program with the protection increased by a factor of sixteen, it did not specify how long an individual switch takes to execute. Because the instruction switches occur randomly throughout the program, it is impossible to determine how much of the overhead is due to switches in loops being executed multiple times and how much is due to the inherent cost of the switch [5].

Madou describes a technique where basic blocks are paired together based on how similar they are. These paired subroutines are called clusters. Each cluster is assigned an area of memory which is used to create the subroutines for that cluster. Whenever the program calls a subroutine within a cluster, it takes the current content of the cluster (a subroutine) and rewrites it as the desired subroutine. This technique causes significant problems for the reverse-engineer as subroutines that share similar structures but are otherwise unrelated occupy the same memory space during the program's execution. As only portions of the subroutine change, it is difficult for the reverse-engineer to determine what the cluster is doing for a particular execution [7].

Dube analyzes the performance of both stack smashing and subroutine reordering by examining the performance costs associated with each individual transform. His research shows that individual opcode

shifts/stack-smashes can be executed on the order of ten nanoseconds and subroutine reordering can be accomplished in less than fifteen microseconds. This granularity is useful, since a developer can properly determine the effect of placing the metamorphic transforms in a program. The research describes the effects of the transforms on two of the most common debuggers, OllyDbg and IDA Pro, and quantifies the effectiveness of opcode shifts by counting the number of incorrectly disassembled instructions [2].

3. Program Fragmentation

This research determines the potential usefulness of program fragmentation as a metamorphic obfuscation transform. Program fragmentation takes multiple segments of code from various portions of the program and constantly scatters them throughout the program. By removing sections of code from a subroutine or basic block, the spatial locality is disturbed, making it harder for the reverse-engineer to understand the section of code, as shown in Figure 1. These memory sections are copied from the normal, inline program flow to random locations throughout the text segment. Part of the memory at the original location is replaced by a procedure call or jump to the new location of the fragment while the remainder of the memory is written over with random instructions.

Figure 1: Conceptual illustration of a fragmented section of code

Whenever the program executes and reaches the original location of the fragment, the program flow will automatically transfer to the new location of the fragment and continue execution. Once the desired instructions have been executed, the program flow will return to the instruction immediately following the section of memory that originally contained the fragment, skipping over the inserted random instructions. Figure 2 illustrates this concept by

showing a normal program on the left and fragmented program on the right.

Fragmentation is very similar to outlining as it abstracts away portions of linear code into individual fragments which are called like subroutines. However, program fragmentation differs from pure outlining because the fragments change locations during the execution of the program. A construct similar to a function-manager or lookup table is needed to keep track of the fragment locations. Rather than directly calling or jumping to the fragment, the program calls the function-manager which then transfers the program execution to the proper fragment location. This technique is very similar to subroutine reordering, but has a much smaller granularity as smaller segments of code can be placed into fragments, rather than entire subroutines.

Figure 2: Conceptual illustration of execution flow of a fragmented program

The test program fragmentation module is programmed in C++ and uses a source-level package to implement the metamorphic engine. The metamorphic engine consists of a fragment storage area, an obfuscated fragment lookup table, and the procedures to perform the fragmentation and associated fragment lookups.

Fragment locations are chosen at the source-level, which simplifies the bookkeeping for the metamorphic engine. The x86 architecture uses variable-length instructions, so it is difficult and costly to fragment at the assembly level. The metamorphic engine would have to disassemble the instruction sequence to properly fragment the instructions. Otherwise, the metamorphic engine could potentially move only portions of an instruction, not moving all of the necessary opcodes. Fragmenting at the source-level ensures that entire assembly instructions are moved without having to perform disassembly.

4. Experiment Methodology

This research determines whether program fragmentation can be added to a variety of programs without greatly increasing the original programs' overhead. The size of the newly transformed program and the additional execution time provides a quantitative measurement of how much the transforms slow down the program. Furthermore, individual execution performance cost for each transformation is also important. Transforms that occur inside loops might execute numerous times, greatly increasing the overhead. It is important to discover what portions of the transform are most costly and how they might be mitigated or the incurred cost reduced.

A qualitative analysis of the effectiveness of the transform enables software developers to fully weigh the cost of the transform with an expected benefit. The program can be disassembled and examined to determine how much protection the obfuscation adds. Examining the resulting code provides a method of estimating how much additional time would be needed to reverse the obfuscated program.

To test the performance of the program fragmentation engine, the obfuscation technique is applied to the five benchmarks of the SciMark 2.0 benchmark suite: fast Fourier transform (FFT), sparse-matrix multiplication (SMM), dense LU matrix factorization (LU), Jacobi successive over-relaxation (SOR), and Monte Carlo integration (MC). These benchmarks are chosen to represent code containing proprietary algorithms, which is a main target for obfuscation. The SciMark 2.0 kernel and benchmark programs are modified slightly to make the runs deterministic. Overhead is measured as the execution time difference between the original, unobfuscated benchmark program and the benchmark program with program fragmentation applied.

The main factors for the experiment are the benchmark program being protected and the size of the fragment storage blocks. Benchmark is chosen as a factor to determine whether instruction mix affects the overhead of the program fragmentation.

Size is chosen because the implementation uses homogenous storage blocks to hold the fragments, regardless of the individual fragment sizes. Therefore, moving fragments is accomplished by moving the entire contents of the storage blocks, and larger blocks take longer to move. Block size levels are chosen based on feasible sizes for fragments.

Sensitivity analysis showed that a reasonable lower bound for fragment size is on the order of 25 assembly instructions. Twenty-five assembly level instructions implement most basic source-level instruction

sequences. Reducing the size beyond this point, however, limits the instruction that can be placed into fragments. As the benchmarks have an average instruction size of approximately three bytes, this requires 75-byte storage blocks. The upper bound is chosen as 600 bytes, as any larger and the fragments approach the size of entire subroutines and would not provide a sufficient level of fragmentation.

Fragment locations are set to obtain a minimum number of fragment executions. The location affects the number of executed fragments, as placing fragments within loops causes the fragment to be executed multiple times. The test benchmarks run at least one million fragment executions to provide overhead differences on the magnitude of seconds. However, because the benchmark programs use series of nested loops to perform the required calculations it is impossible to make the number of executions the same for each benchmark.

The factors and their levels are displayed in Table 1. Other factors such as varying the implementation are considered beyond the scope of the research and left for future work.

Table 1. Factor levels

Factor	Levels
Workload (Benchmarks)	FFT
	SOR
	MC
	SMM
	LU
Fragment Size	75
	150
	300
	600

5. Results and Analysis

The experiments provide 5,000 distinct measurements of execution time. One thousand baseline executions are measured to characterize the performance of the five unmodified benchmarks and 4,000 additional executions are used to measure the overhead associated with each of the obfuscation configurations. For analysis, the 200 samples of each obfuscation configuration are considered to directly correspond to the 200 trials of the associated benchmark baseline. Each baseline time is subtracted from the corresponding measured time for the obfuscated program, giving the execution time added by the program fragmentation. This execution time is then divided by the observed number of executed morph points to give the primary metric of estimated overhead per morph point. The resulting data is 4,000

measurements of overhead per morph which is analyzed using Minitab.

The main effects for block size, benchmark and their interaction are determined using Two-Way ANOVA. Figure 3 is a graphical representation of these effects. The horizontal lines show the 95% confidence interval boundaries, with the values displayed to the right of each chart. Any values falling outside of the boundaries are statistically significant at 95% confidence.

Figure 3. Effects of factors on overhead

Overhead per morph clearly shows a positive linear relationship with block size. This is expected as larger blocks require the metamorphic engine to move more bytes between fragment storage blocks, increasing the execution time. The overhead by block size is shown in Table 2. For block sizes of 75 bytes, fragments can be executed in less than one microsecond which is unlikely to be noticed in an interactive application.

Table 2. Overhead per fragment based on size

Size	Mean (us)	Std Dev (us)	95% CI (us)
75	0.936	0.0292265	[0.932, 0.940]
150	1.175	0.0142192	[1.173, 1.177]
300	1.78	0.0149885	[1.778, 1.782]
600	2.991	0.0221858	[2.988, 2.994]

The effect of the benchmark on overhead per morph was expected to be very limited, as the instruction mix should not affect the overhead per morph. However, SMM has significantly higher overhead compared to the other benchmarks as shown in Table 3. Further analysis partially attributes the increased overhead to a larger number of source-level fragments placed in the SMM code.

Table 3. Overhead per fragment by benchmark

Benchmark	Mean (us)	Std Dev (us)	95% CI (us)
FFT	1.69905	0.797025	[1.588, 1.810]
LU	1.73779	0.79842	[1.627, 1.849]
MC	1.71658	0.789687	[1.607, 1.826]
SMM	1.8423	0.814799	[1.729, 1.955]
SOR	1.72936	0.799198	[1.618, 1.840]

While the structure of the non-SMM benchmarks uses a single source-code level morph point to provide the desired number of executed transforms, SMM uses four source-code level morph points to provide the minimum of one million morphs. Reducing the number of source-code morph fragments in SMM decreases the overhead per morph, but significantly increases variance because of the smaller number of morphs. The increased variance prevents the difference from being statistically significant at a 95% confidence level.

The interactions plot shows that several of the interactions appear to have small, but statistically significant deviations from the overall mean. Further analysis shows the effects are not large enough to be relevant when compared to the impact of block size on the overhead.

6. Effectiveness Observations

To determine how effective program fragmentation is at hampering reverse-engineers, it is applied to a program that performs a series of basic math operations. The protected program is analyzed using OllyDbg, a commonly-used linear sweep disassembler/debugger, and IDA Pro, a popular recursive-transversal disassembler/debugger. How well or poorly the tools perform while examining the code provides insight into how effective the technique could be in protecting software.

Both tools have difficulty in correctly handling the fragmented program. Because program fragmentation replaces the fragmented instructions with random data bytes, the fragmented section of code is disassembled incorrectly. The random bytes perform an opcode shift which causes both disassemblers to become misaligned to instruction boundaries, and display incorrect assembly instructions to the user. Figure 5 shows the OllyDbg disassembly of a fragmented section of code. In this case, the OllyDbg incorrectly disassembles a RETN instruction and marks the rest of the section as data bytes even though it actually contains instructions.

Figure 5. OllyDbg disassembly of fragmented section of code.

Program fragmentation also prevents breakpoints from working as expected in both the fragmented area of the program and the fragments themselves. In the fragmented area, if the user places a breakpoint on one of the random data bytes, thinking it is a valid instruction, the breakpoint is never encountered and does not stop execution. In addition, placing breakpoints in a fragment causes the program to crash. When a breakpoint is placed on an instruction, both tools replace the first byte of the instruction with a 0xCC byte. When the fragment is moved, the 0xCC byte is moved rather than the original byte of the instruction. This corrupts the instruction sequence, causing invalid execution of the program.

Likewise attempting to step over a call to a fragment causes the program to continue execution until it either encounters a breakpoint or reaches the end of the program, rather than stopping after completing the call. To step over an instruction, both tools appear to place a breakpoint on the instruction immediately following the stepped-over instruction. However, when a fragment completes, program execution does not return to the instruction immediately following the call, which in a fragmented program is random bytes. Instead, the execution returns to the first valid instruction following the fragment. Therefore, the inserted breakpoint is never encountered, and does not return control of the program to the reverse-engineer.

7. Conclusion

This research shows program fragmentation is viable as an effective, low-cost metamorphic software protection. It shows that the protection creates problems for two of the most common reverse-engineering tools and makes the task of a human reverse-engineer more difficult. It also determines that the performance cost for executing the transforms is

low enough to allow it to be used throughout a program.

The results of this research show that program fragmentation can be useful in protecting software programs from reverse-engineering. By making reverse-engineering more difficult, this technique can help keep proprietary algorithms safe from competitor reverse-engineers, prevent malicious users from finding holes to exploit in software, and even help keep government and military software from being reverse-engineered by other nations.

Further research into program fragmentation and other metamorphic techniques can provide a layered defense that improves the security of all software, benefiting both developers and users.

10. References:

- [1] Collberg, Christian, Clark Thomborson, and Douglas Low. "A Taxonomy of Obfuscating Transformations". Technical report, Department of Computer Science, University of Auckland, 1997.
- [2] Dube, Thomas. "Metamorphism As A Software Protection For Non-Malicious Code". Master's thesis, Air Force Institute of Technology, 2006.
- [3] Eilam, Eldad. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, 2005.
- [4] Gao, Xufen. "Metamorphic Software for Buffer Overflow Mitigation". Technical report, Department of Computer Science, San Jose State University, 2005.
- [5] Kanzaki, Yuichiro, Akito Monden, Masahide Nakamura, and Ken ichi Matsumoto. "Exploiting Self-Modification Mechanism for Program Protection". In *Proc. of the 27th Annual International Computer Software and Applications Conference*. 2003.
- [6] Linn, Cullen and Saumya Debray. "Obfuscation of Executable Code to Improve Resistance to Static Disassembly". Technical report, Department of Computer Science, University of Arizona, 2003.
- [7] Madou, Matias, Bertrand Anckaert, PatrickMoseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. "Software Protection Through Dynamic Code Mutation". Technical report, Department of Electronics and Information Systems, Ghent University, 2005.
- [8] Szor, Peter. *Art of Computer Virus Research and Defense*. Addison-Wesley, February 2005.