Implementing a Disassembly Desynchronization Obfuscator

Ivar Härnqvist Johannes Wilson
Email: {ivaha717, johwi801}@student.liu.se
Supervisor: Ulf Kargén, ulf.kargen@liu.se
Project Report for Information Security Course
Linköping University, Sweden

ABSTRACT

This report presents an implementation of a code obfuscator that works on the assembly stages of code compilation, using so called disassembly desynchronization. This type of obfuscation is aimed at making the disassembly of a program yield code that is different from the code that is executed. This type of obfuscation is of special interest because it allows entire instructions to be potentially hidden from static disassembly.

The presented implementation works automatically on arbitrary codebases that use GCC and GNU Make. It can successfully insert a large number of opaque predicates in a large codebase while preserving the expected program behavior, though it may introduce errors in certain programs.

1. INTRODUCTION

Obfuscation of code is an important tool in discouraging reverse engineering of code [1]. It is used in order to protect a program from malicious reverse engineering [2], but can also be used by malware authors to hide malicious code [6].

Reverse engineering binary code is normally done through the use of a disassembler to extract readable assembly code from the binary [5]. Disassembly of binary code is however not a trivial problem when instructions are of variable length, since the disassembler needs to stay in sync with the instructions [4]. One obfuscation technique that exploits this difficulty is disassembly desynchronization, which tries to disrupt disassembly of the machine code [4]. This form of obfuscation prevents disassemblers from producing correct assembly code, and can even be used to hide function calls entirely from the disassembly. In the case of malware, this can be especially dangerous, as some malware detection techniques rely on searching disassembly for potentially malicious code [3].

In order to study the effects of disassembly desynchronization there is a need for tools that programmatically introduce disassembly desynchronization during compilation of code. This report describes the development of such a tool.

1.1. Disassembly Desynchronization

The purpose of an assembler is to translate human-readable assembly code into binary code that is executable by the CPU. A disassembler attempts to do this process in reverse, producing human-readable assembly code from binary code. If the underlying architecture uses a variable length instruction format, the disassembler needs to correctly identify the beginning of each instruction. Failure to do so will result in the disassembler attempting to translate non-instruction code into instructions, resulting in an incorrect disassembly. Because a program can jump to arbitrary points in the code there is ambiguity in whether to interpret code as instruction or as data. This ambiguity must be resolved by the disassembler.

A recursive disassembler will resolve the ambiguity by interpreting the instructions in order, and if branching instructions are encountered the disassembler will begin disassembling from the new location, interpreting those parts of the memory as instructions [4]. Not all branches have to be taken however, yet such a disassembler would still interpret all code in those locations as instructions.

Disassembly desynchronization works by adding a branch condition that is always evaluated to be either true or false, known as an *opaque predicate*. Assembly code can be inserted into the branch that is not taken without altering the execution of the program. This will cause the disassembler to interpret said code as instructions, causing it to end up out of phase with the actual code. This can be exploited in order to hide certain instructions completely from the disassembled result.

Figure 1 and Figure 2 give an example of hiding the first call to function “foo” using disassembly desynchronization.

In Figure 2, the original instruction still remains (highlighted in orange) but because the disassembler gets out of phase with the code it will interpret different instructions. Also note that the instructions eventually line up again, showing correct instructions from instruction *lea* onward. This is often desirable when obfuscating as it helps hide the presence of obfuscation from the reader.

1.2. Problems/Questions

This report describes the development of a program for automatic obfuscation in the form of disassembly desynchronization. The disassembly desynchronization is performed using opaque predicates resulting in an inserted branch always being taken. The code immediately following the jump is then never executed, allowing insertion of arbitrary ”junk bytes”. These junk bytes are then selected in such a way that the disassembled instructions form different instructions that are still valid, in an attempt to hide the presence of disassembly desynchronization.

```
<+9>:     48 8b 7e 08           mov    0x8(%rsi),%rdi
<+13>:    e8 2e 01 00 00        callq  0x680 <foo>
<+18>:    31 c0                 xor    %eax,%eax
<+20>:    48 83 c4 08           add    $0x8,%rsp
<+24>:    c3                    retq
<+25>:    48 8d 3d d3 01 00 00  lea    0x1d3(%rip),%rdi        # 0x733
<+32>:    e8 1b 01 00 00        callq  0x680 <foo>
<+37>:    eb eb                 jmp    0x552 <main+18>
```

Fig. 1. Original program without modification disassembled using GNU Project Debugger (GDB). The first call to function foo is on line two, with its binary code highlighted in orange.

```
<+9>:     48 8b 7e 08           mov    0x8(%rsi),%rdi
<+13>:    31 c0                 xor    %eax,%eax
<+15>:    85 c0                 test   %eax,%eax
<+17>:    74 03                 je     0x556 <main+22>
<+19>:    f7 f3                 div    %ebx
<+21>:    20 e8                 and    %ch,%al
<+23>:    25 01 00 00 31        and    $0x31000001,%eax
<+28>:    c0 48 83 c4           rorb   $0xc4,-0x7d(%rax)
<+32>:    08 c3                 or     %al,%bl
<+34>:    48 8d 3d ca 01 00 00  lea    0x1ca(%rip),%rdi        # 0x733
<+41>:    e8 12 01 00 00        callq  0x680 <foo>
<+46>:    eb eb                 jmp    0x55b <main+27>
```

Fig. 2. Disassembled code after modification. The green bytes add an opaque predicate that will cause the program to always jump to line +22. The bytes in blue are unused by the program but will cause the disassembler to interpret different instructions after the jump.

This project aims to find solutions to the following problems:

1) How to determine which registers can be safely used to construct opaque predicates at a given part of the intermediate assembly for an arbitrary C program.
2) How to determine the optimal number of junk bytes, and their contents, to insert at the desynchronization point to produce valid x86 instructions.

This report focuses on how to insert opaque predicates into assembly code in order to create disassembly desynchronization points. It only briefly discusses how to insert junk bytes to produce valid but incorrect instructions in the fake branch. This is instead covered in a report by a different group working on the same project.

1.3. Scope and Limitations

The project assumes that the code to be obfuscated is for a 64-bit x86 architecture, and can be compiled using GCC and GNU Make.

1.4. Method

To find solutions to the given problems and facilitate automated disassembly-desynchronization of arbitrary C programs, a command-line tool is developed which operates on the intermediate x86 assembly files of the compilation process as well as the final binary program executable.

Different parts of the tool are developed in parallel by two groups of two people each. The tool is tested using a wrapper for the GNU Compiler Collection (GCC) through a test script which assumes the project that is to be tested can be built and have its test suite run using the build automation tool GNU Make.

The general procedure for implementing and testing the tool is to add a new stage or feature and then test it against existing open-source C projects while fixing regressions until all projects produce the same observable behavior as they did before obfuscation and thus make all test suites pass.

The finished tool is benchmarked against performance-intensive tests of different projects to determine what impact the obfuscation has on runtime execution speed. The time to compile and obfuscate the code is also measured to see if the tool adds significant overhead to the compilation process.

2. Background

This section covers background knowledge that may be useful for understanding the rest of the report.

2.1. GNU Compiler Collection

The tool developed for this project assumes that the programs on which it operates are normally compiled using the GNU Compiler Collection (GCC). The main GCC program runs different sub-programs such as the compiler, assembler and linker during compilation. For example, during compilation of C code, GCC may run cc1, as and collect2 for the compilation, assembly and linking steps respectively.

There is built-in support in GCC for directing these subprogram commands to another executable using the -wrapper argument. This way it is possible to intercept the compilation process and perform operations on the intermediary compilation results.

2.2. GNU Assembler

When GCC compiles using cc1 it will output assembly code in the GNU Assembler language. GNU Assembler or GAS uses AT&T assembly syntax by default, meaning the instructions follow the order mnemonic, source and destination last. As well as instructions, the assembly files contain *labels* marking the targets for branch instructions.

2.3. Keystone / Capstone

Keystone and Capstone are two open-source frameworks capable of performing assembly and disassembly respectively. In addition to performing disassembly, Capstone also provide information about the registers and flags used by the disassembled instructions.

2.4. Liveness analysis

To determine which registers are available for use at a certain point in the assembly without direct communication from the compiler, the tool needs to parse the generated assembly code and generate a *control flow graph* (CFG) in order to perform *liveness analysis*. The control flow graph is a directed graph of the code's possible execution paths where the nodes are snippets of code called *basic blocks* which do not contain any branches. Liveness analysis traverses the control flow graph to find registers which are not needed by future instructions and which can therefore be safely used by our tool when inserting opaque predicates.

2.5. State of the art

The paper by Cullen Linn and Saumya Debray [4] describes a successful implementation of a program performing disassembly desynchronization with junk byte insertion using two methods. One method inserts bytes before suitable basic blocks, while the other modifies unconditional branches to jump to a different target, allowing junk byte insertion. Our project instead attempts to produce disassembly desynchronization at arbitrary points in the code.

3. Design and implementation

The desynchronization tool consists of two stages - predicate insertion and junk byte selection. The first stage performs liveness analysis on the given assembly code and uses that information to determine which registers it can use to insert opaque predicates at random locations in the code. After each inserted predicate, it also appends some placeholder junk bytes which will be skipped during execution. The second stage is run after the assembly code has been assembled and linked to binary machine code, and its job is to modify the placeholder bytes in the binary such that they form valid instructions that merge with the subsequent code and thus cause disassemblers to misinterpret them. This report only covers the first stage of the tool, since the second stage was implemented by a different group.

3.1. Implementation

The program for inserting predicates is implemented using C++20. It takes a number of assembly file names as command-line arguments and works as specified by a configuration file containing templates for opaque predicates. An example of such a predicate template is given in Figure 3. It then reads each assembly file and constructs CFGs from the instructions. The CFGs are used to perform the liveness analysis, which provides possible locations for insertion of predicates. The program then randomly selects predicates based on the given configuration and inserts them at random intervals in each file, also specified by the configuration.

```
1   predicate xorl_jz(reg_a: r32) {
2       xorl %reg_a, %reg_a
3       jz DESYNC
4   }
```

Fig. 3. Example of a predicate template. The code within the brackets will be what is inserted, with the register names replaced by registers of appropriate size, and with DESYNC replaced by a target label after the junk bytes.

3.2. Liveness Analysis

Liveness analysis is performed based on information given by the disassembly library Capstone. To obtain this information, each instruction is first assembled using the assembly library Keystone, so that they can then be disassembled using Capstone. The information provided by Capstone is used to identify branching instructions, which separate basic blocks in the CFG. The branch instructions are then matched with their respective destination labels to generate the list of potential successors of the basic block. Labels are always treated as possible branch targets, regardless of whether there is a corresponding branch or not. This is because the label could for example mark a function that is called from a different file. Similarly, branch instructions for which the target label cannot be found are treated as if every register and flag is live, since the branch could be referencing a function in a different file that could read any register or flag.

Capstone is also used to provide information regarding register liveness and flag usage. For each instruction, the liveness of each register and each flag is tracked using two bit vectors, where a bit is set if the corresponding register or flag is live. Liveness is calculated iteratively starting from the bottom of each basic block going to the top, with the bottom instruction being calculated from the union of live registers from the successors of the basic block. If the basic block has no successors, or if one of the successors are an unknown location it is assumed that any register or flag can be live at the end of the basic block.

Instructions that write to a register provide an available register for instructions before it, while registers that are read

need to maintain their value from their previous write, and are thus not available for instructions before it. Using this rule, each basic block is updated bottom to top, and if the first instruction had its liveness updated, the predecessors of the basic block must also be updated. This continues until the liveness for each basic block has converged.

3.3. Predicate Insertion

The user specifies the opaque predicates to be used as templates. A template contains all instructions needed to guarantee that a branching instruction is performed. The required bit lengths of the registers are specified in the template, but in order to know which flags a given predicate will modify the program needs to process them using Keystone and Capstone.

Knowing which registers a predicate needs, which flags it will modify, along with the liveness analysis for each instruction, it is possible to tell whether a given predicate can be inserted into the assembly code without altering the execution of the program. The predicates are inserted by traversing the assembly code in the file from top to bottom, picking a predicate at random, selecting candidate locations based on the interval specified in the configuration and, if possible, adding the modified template using any free registers.

3.4. GCC wrapper

To automate the process of desynchronizing large codebases, the tool provides a Bash script that compiles a given project using GCC and intercepts the different compilation steps in order to interleave the two stages of the desynchronization tool appropriately. It also provides a test script which uses GNU Make to compile a given project with the GCC wrapper and run its test suite to verify that no errors were introduced by the predicates.

4. Results

The desynchronization tool was tested by using it to compile two C projects and then running the tests for each project. The projects selected for testing were coreutils and libpng. Two different configurations were used for each project, and for each test the following information was collected:

1) Time to compile
2) Number of instructions processed
3) Number of predicates inserted
4) Number of tests passing or failing
5) Execution time of tests

The configurations used to desynchronize the projects both used the Mersenne Twister 19937 pseudo-random number generator with a seed of 493119347, a uniform junk byte count distribution between 1 and 3 bytes and a normal predicate interval distribution with a standard deviation of 10, clamped using the formula ”round(max(x, 1))”, where x is the generated value from the distribution. The mean value of the predicate interval normal distribution was set to 20 and 100 instructions for the respective configurations. The type of predicate to be inserted was always an ”xorl” instruction of a single 32-bit register followed by a ”jz” branch instruction that skips past the junk bytes.

The number of predicates successfully inserted and number of tests passing after insertion is shown in Table 1 and Table 2.

4.1. Benchmarks

To test what effect the predicate insertion has on compilation and execution time, the two test projects were compiled with the configuration specified above using a mean predicate interval of 20 instructions. The projects were recompiled with the command ”make -j 12 clean && make -j 12” and had their test suites run with ”make -j 12 check”. Note that for coreutils, the tests for gnulib were not included. This is because, by default, the tests for gnulib will not run automatically unless all the tests for coreutils pass. The gnulib tests were excluded by passing the option ”SUBDIRS=.” to the ”check” target. The compilation and test execution times were measured using the GNU Time program. Each benchmark was repeated at least 5 times. The average relative results are shown in Figure 4.

The following system was used to run the benchmarks:

- Kernel: Linux 5.12.1
- GCC version: 10.2.0
- GCC optimization level: -O3
- CPU: Intel Core i7-8700K
- RAM: 16 GB, 3200 MHz

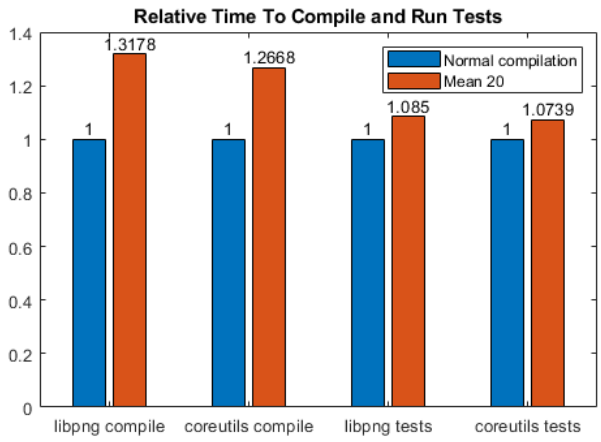

Fig. 4. Bar graph showing relative time to compile and test libpng and coreutils using the program. Bars in orange show the factor by which it is slower to compile and test using our program compared to normal compilation.

5. Discussion

Figure 4 shows that the desynchronization tool increases compilation time by approximately 30 % and execution time by approximately 8 %. Table 2 shows that using the tool may cause some tests to fail.

5.1. Evaluation of Results

The insertion of opaque predicates occasionally causes the tests to fail. The tool would be more useful if it could guarantee that inserted predicates will not introduce errors.

TABLE 1
LIBPNG, TOTAL INSTRUCTIONS: 108302, TOTAL TESTS: 32

Mean Interval Distribution	Predicates Inserted	Average Interval	Tests Passed	Tests Failed	Tests Skipped
20	3310	33.02	32	0	0
100	1001	109.19	32	0	0
None	-	-	32	0	0

TABLE 2
COREUTILS, TOTAL INSTRUCTIONS: 215505, TOTAL TESTS: 626

Mean Interval Distribution	Predicates Inserted	Average Interval	Tests Passed	Tests Failed	Tests Skipped
20	1753	47.14	495	13	116
100	1001	215.29	508	2	116
None	-	-	509	0	117

The increase in execution time for the obfuscated program does not seem significant as indicated by the small difference in testing time.

There is a significant difference in the desired mean predicate interval requested by the user and the resulting average interval. This is due to the insertion being delayed until a suitable location is encountered, even after exceeding the target interval. The program will not compensate for these missing predicates, resulting in a lower than expected average number of predicates.

5.2. Sources of Error

The errors introduced by the predicates indicate an incorrect liveness analysis, since errors only appear when using a predicate that modifies a register. These errors could be caused by errors during the construction of the control flow graph since it forms the basis of the liveness analysis. There could also be bugs in the implementation of the liveness analysis algorithm itself. A few implementation bugs were discovered during the development process, so it is possible that some still remain.

Because the program relies on the register and flag information provided by Capstone through disassembly of instructions, Keystone and Capstone could be possible error sources as any errors in either framework will propagate to the program, resulting in incorrect liveness analysis.

We only analyze each predicate once on program startup, using a placeholder register. It is possible that some instructions behave differently depending on the register provided in the operand, in which case our initial analysis of a predicate may not be correct for every choice of register. A possible remedy could be to always analyze the predicate again when registers have been chosen, in order to ensure that the instructions won't use additional flags.

During liveness analysis, the program ignores any non-instruction assembly directives in the file. It is possible that some of these directives play a role in the liveness of some flags or register, in which case this will be missed by the program. They could also play a role in the control flow of the program.

5.3. Future Work

Once the liveness analysis has been adjusted to perform the insertion of predicates with full accuracy, a few additional features may be desired. One such feature may be to perform the selection of registers from the available ones at random. The current implementation is to select the first register available, but some other selection method may be desirable.

Another improvement is to adjust the interval selection so that the resulting average interval more closely reflects the one specified by the configuration. This could be accomplished by selecting the predicate locations while allowing backtracking and then sorting the locations by their position in the assembly file before writing the predicates to the output in order. The current implementation relies on the predicates being written from top to bottom.

Because this type of obfuscation can be used in malware, a possible future use for this program could be to generate test examples of disassembly desynchronization. Those examples can then be used to train machine learning algorithms to automatically detect the presence of disassembly desynchronization.

6. CONCLUSIONS

Our program has shown that this method of automatically inserting disassembly desynchronization is feasible. Though before the tool is to be used in any sensitive application, there are some errors that need to be corrected.

REFERENCES

[1] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. San Diego, California, USA: Association for Computing Machinery, 1998, pp. 184–196. ISBN: 0897919793. DOI: 10.1145/268946.268962. URL: https://doi-org.e.bibl.liu.se/10.1145/268946.268962.

[2] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. “Control Flow Based Obfuscation”. In: DRM ’05. Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 83–92. ISBN: 1595932305. DOI: 10.1145/1102546.1102561. URL: https://doi-org.e.bibl.liu.se/10.1145/1102546.1102561.

[3] R. Isawa et al. “Evaluating Disassembly-Code Based Similarity between IoT Malware Samples”. In: *2018 13th Asia Joint Conference on Information Security (AsiaJCIS)*. 2018, pp. 89–94. DOI: 10.1109/AsiaJCIS.2018.00023.

[4] Cullen Linn and Saumya Debray. “Obfuscation of Executable Code to Improve Resistance to Static Disassembly”. In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. CCS ’03. Washington D.C., USA: Association for Computing Machinery, 2003, pp. 290–299. ISBN: 1581137389. DOI: 10.1145/948109.948149. URL: https://doi-org.e.bibl.liu.se/10.1145/948109.948149.

[5] Chengbin Pang et al. *SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask*. 2020. arXiv: 2007.14266 [cs.CR].

[6] B. Wanswett and H. K. Kalita. “The Threat of Obfuscated Zero Day Polymorphic Malwares: An Analysis”. In: *2015 International Conference on Computational Intelligence and Communication Networks (CICN)*. 2015, pp. 1188–1193. DOI: 10.1109/CICN.2015.230.