Automating insertion of disassembly desynchronization points into an arbitrary program

A research tool for developing obfuscation detection techniques

Evelina Holmgren
Linköpings universitet
Linköping, Sweden
eveho444@student.liu.se

Gustav Eriksson
Linköpings universitet
Linköping, Sweden
guser908@student.liu.se

***Abstract*—Due to digitization, malware is an ever increasing threat. To train antivirus programs to recognize malicious binaries, an automated tool for desynchronizing the disassembly process for arbitrary programs have been produced. It mimics the disassembly desynchronization technique that malware often implements in order to change the control flow and hide its intentions from the AV program. The problem is how this is done, both regarding insertion of desynchronization points and finding free registers for use in so-called opaque predicates. The result of this report is that randomly generated junk bytes, together with a standardized predicate, is able to desynchronize binaries at call instructions. However, the effectiveness and the performance of this technique could be improved.**

I. Introduction

In an increasingly digitalized world, malware is a constantly growing threat. To protect ourselves against this, antivirus software is developed. One of the many important tasks these antivirus programs do is analyzing the malicious binaries in order to find ways to identify them and to help the analysts understand their behaviour. One way they do this is by disassembling the malicious executables back into human-readable assembly code. Malware authors are aware of this strategy though, and can disrupt the process by using an anti-reverse-engineering technique called *disassembler desynchronization*. This works by obfuscating the assembly code by inserting junk bytes before a certain code block that the author wants to hide [5]. By inserting a predicate that makes sure these junk bytes are never actually executed, the behaviour of the program can be kept unchanged. However, a disassembler would attempt to disassemble the junk bytes, which desynchronizes the instruction stream so that the contents of the code block is obscured [5].

To be able to develop better defenses against disassembler desynchronization, a tool will be created that can obfuscate code at given locations and mark these, in order to produce binaries that can be used to study techniques to counter this kind of obfuscation.

This brings a few problems that need solving when creating this disassembler desynchronization tool. Firstly, the disassembler must produce valid assembly code from the altered binary file. If a disassembler can not interpret everything in the binary file as valid instructions, it would not be far-fetched to think that someone would realise that an attempt to obfuscate the code have been made. This in turn makes the process of restoring it to its original state much easier.

Secondly, the predicate responsible for making sure that the junk data is actually never executed, uses registers. Depending on what predicate is used, there must exist one or more registers that is currently not in use by the program, i.e. a register which with certainty is written to before it is read from after the desynchronization point. Note that there is no guarantee that such a register exists at any given time.

These two key problems have been formulated as the main research questions of this project:

1) How to choose the junk bytes that should be inserted at a certain desynchronization point?
2) How can it be determined which registers that can be modified at a certain desynchronization point without compromising the functionality of the program?

The research questions have been handled separately by two groups working concurrently on the project as a whole. Therefore, this report will focus more heavily on research question one, which corresponds to the problem that was handled by the authors.

A. Delimitations

The delimitations on the tool is that it will only target 64-bit x86 Linux systems. Another delimitation is the limited time assigned to this project, which affect the scope and the refinement of the tool.

II. Background

This section provides the necessary background information about the concepts and about the problems that needs to be solved. It focuses on the tools and libraries used and provides an in-depths description of the underlying concepts of disassembly desynchronization.

A. External Tools and Frameworks

Some technical details about the external tools and frameworks are provided below.

1) GCC - GNU Compiler Collection: GCC is the compiler that is used by this tool, and it is important to understand the basics of how it works. The process of creating an executable from source code is approximately the same as most compilers and can be split into four distinct steps, where step two and three are particularly important for this project. [4]

1) Preprocessing - The first step in the GCC-compilation process is to expand macros and included files, remove comments, etc. If it encounters a *#include* in a C program for example, it copies the code from the included file into the current one.
2) Compiling - The compiler produces human readable intermediary code in the form of processor specific assembly code.
3) Assembly - The assembly code produced in the previous step is assembled into object code, which is a sequence of bytes determined specific to the processor the compiler is currently running on. This is structured in a way such that there is no ambiguity, i.e. a complete instruction can not act as a suffix to another one.
4) Linking - In the final step the linker will merge all object code from different modules into one, creating a complete executable file.

An important part of the entire compilation process is the symbol table. The table stores the locations of variables and functions by name, called symbols. The symbol table is mostly used internally by the compiler and operating system, but can be stored in the executable which allows for external access as well. It is possible to declare symbols manually in the assembly file. [3]

2) Pyelftools: Pyelftools is a Python library that can be used to analyze ELF-files and debugging information in DWARF format [1]. ELF stands for *Executable and Linkable Format* and is a standard format for executables [2].

3) Capstone: Capstone is a lightweight disassembly framework that can be used to analyze binaries. It is compatible many hardware architectures and has support for multiple programming languages. [6]

B. Disassembly Desynchronization

As briefly explained in the introduction, disassembly desynchronization is a technique used to obstruct the process of reverse-engineering source code from an executable file. There are a few different ways to do it, but the one technique this tool will implement is the use of junk bytes and predicates. The core strategy is to plant junk data into the compiled assembly file, and then ensure that segment is never executed. Since the disassembler cannot deduce that these junk bytes are unreachable by the program, it will try to interpret them as instructions. Depending on the data, the disassembler may use the subsequent real instructions as arguments or op-code suffixes and thus fail to recreate the original assembly code. It is in theory possible for the disassembler to continue to interpret the bytes incorrectly and remain desynchronized for the rest of the program, but in practice they have been shown to resynchronize quite quickly. This usually happens 2-3 instructions after where the desynchronization initially happened, hereby referred to as the *desynchronization point*. This entails that if the user wants to ensure that a specific function call or another instruction will be hidden, the desynchronization point will have to be placed right before this instruction in the assembly file. [5]

In order to locate the desynchronization points in the binary, a uniquely named new symbol is placed right after the predicate and before the junk bytes in the assembly file. When the assembler assembles the file it stores the location of that symbol in the symbol table, making it relatively easy to access it with pyelftools. This allows for easy modification of the junk bytes directly in the binary.

As stated in the introduction, in order to reliably insert effective desynchronization points into an existing program without altering the behaviour of that program, the two problems formulated as research questions will have to be solved.

1) Research question 1: The main goal of the junk data is to disrupt the disassembler and tricking it into incorrectly interpret the program. However, it must also produce code that could be believed to be an actual program, and make the desynchronization points blend in. This raises some extra criteria for the junk bytes to be accepted. First and foremost, they must result in valid assembly code. Otherwise it would be easy for someone to realize that the code have been modified. Secondly, there should be some level of randomness present in order to hide the deynchronization points, and to make it harder to predict their locations.
2) Research question 2: The predicate is the part of this solution that is responsible for making sure the junk bytes are never actually executed when the program is running. Since many of the more effective predicates needs to modify a register in order to work, it is critical to make sure that this register is not currently in use by the program. Otherwise the program may crash or stop functioning properly. This problem is handled by the other group. For more information, see their report.

III. Method

The method used during the project is partly a literature study where information were gathered and a development phase where the software were produced. The development tasks were divided between the groups in order to speed up the process.

A. Literature Study

To gather the necessary information needed for the implementations and for the development of this report, a literature study has been conducted. The majority of the articles that have been used was recommended as a reading starting point in the beginning of the project. Aside from these, the majority of the material used is the official documentation for relevant libraries and software.

B. Development

The development strategy that has been conducted during the project consist of weekly meetings with the assigned project supervisor. In these meetings the current implementation have been discussed and studied in order to produce new tasks for the upcoming week in an iterative way. During this time the team members have collected the necessary information and worked on implementing the tasks. When problems have been encountered, discussions with the supervisor have taken place in order to clarify and explain the problem and find suitable solutions.

All programming has been conducted in pairs in order to enable discussion. During the sessions one in the pair writes the code and the other dictates. This allows for cooperation and opens up for discussion where both in the pair are involved in the design decisions and design compromises.

C. Cooperation

The project have been divided into two sub projects, where two groups have taken one research question each. This report focuses on the first research question. The cooperation between the groups have consisted of joint meetings where the progression on both fronts have been discussed as well as the design of the interface between the tools.

IV. Solution and Analysis

This section describes implementation of the tools and the design decisions taken. It also contains the results from some benchmark tests. Lastly it discusses the results and gives suggestions for future improvements.

A. Design and Implementation

The part of the project that revolves around finding suitable junk bytes was implemented in Python. It is divided into two parts: a prototype for inserting desynchronization points, and a junk byte generator which contains the important functionality for the desynchronization of the binary.

1) Prototype: The first step in the development phase was to create a simple prototype for inserting assembly code into an assembly file. This was done in a very simple way; the same predicate and junk bytes were inserted at every function call. At the start of each desynchronization point, symbols named incrementally was inserted to point out their position in the file. This is done by inserting a simple line of code at the given position of the format ”desynchpoint[index]_[size]:”, where index is the number of the desynchronization point and the size is the number of junk bytes that follows. The symbols are initialized as global in the beginning of binary, before the start of the main function. This initialization is done by inserting the string ”.globl desynchpoint[index_[size]”. This enables us to find the symbol in the symbol table at a later stage.

The original file was read line by line, and concatenated with the predicates and symbols and stored in a string. At the end, when the whole file had been processed, the original file was overwritten with the new string. The purpose of the prototype was to find a way to easily insert desynchronization points into a already existing file.

```
# Pseudo code for prototype.py
for file in files
    line = file.readline
    while line
        if "call" in line
            generate junk bytes of given
             ↪  length
            create predicate
            add predicate to file string
            add symbol to string with
             ↪  symbol declarations
        else if "main:" in line
            save index of "main:"
        add line to file string
        line = file.readline
    add symbol string at index of "main:"
    overwrite file with file string
```

Later on new features to the prototype was implemented. Instead on inserting the same junk bytes at each desynchronization point, single byte-instructions were chosen randomly from a list and inserted. The prototype now supports both fixed number of junk bytes and random number of junk bytes. This is controlled via the terminal when the prototype is run. The user can choose a fixed number of junk bytes between one and three, or letting the prototype randomly choose a number between one and three at each desynchronization point.

However, all desynchronization point will have the same form and will be located at each *call* instruction.

2) Desynchronization tool: When the prototype was developed, the junk byte generator was created. This tool finds and generates new junk bytes to the binary and inserts them into the file at the same location as the placeholder junk bytes that the prototype inserted.

The desynchronization tool works by reading the binary as an ELF file and accessing the symbol table of the binary using the *pyelftools* library. From this table the symbols of the desynchronization points that were inserted by the prototype can be accessed. The list with the desynchronization symbols can then be used to get the byte offset to where the symbols are located in the binary. This list is sorted in reverse order, so we start in the end of the file and go backwards. This is done in order to ensure that no desynchronization alters the behaviour of the next desynchronization point, which could be the case if we iterate over it from the beginning to the end.

Then we iterate over the symbols in the list and use the offsets to extract a code snippet from the binary starting from that point. The length of this code snippet can be changed to any number of bytes, but is now set to 50 bytes.

The first step is to disassemble this original code snippet that have the inserted single-byte instructions in the beginning.

The reason for this is to see how far the disassembler can disassemble and use this length to compare with the length it can disassemble after junk bytes used for desynchronization have been inserted. The tool that is used to do this disassembling is *Capstone*. A list containing the original assembly instructions of the code snippet is also saved to be used after the desynchronization have been done.

Then we enter a while-loop that runs until the code is desynchronized. Inside the loop we try to find junk bytes that creates a disassembly output that have the same number of bytes as the original code snippet, which means that the disassembler should be able to disassemble to the same address. Additionally, we check that the call instruction that we want to hide is not present at the same address as before, which is how we confirms that the code has been successfully desynchronized.

When the junk bytes are generated, the tool always starts by trying to insert the maximum number of junk bytes. For example, if there are three junk bytes inserted into the code, it starts by trying to find working combinations of three junk byte. However, when it has tried a given amount of times, set to 1000 in the code, it goes on by trying combinations of two bytes and after another 1000 tries it test one byte instead.

The first junk byte in the combination of two or three cannot be a working single byte op-code because that will not desynchronize the code. The byte cannot be a x86-prefix either since those only alters the functionality of the following instruction, thereby failing to hide it. For this reason, all single byte op-codes and prefixes are read in from two text files and are used to create a list of single bytes that are not present in one of these, and therefore might work to insert. So when the first junk bytes, no matter if the number of junk bytes that are generated are one, two or three, it must be chosen from this list of bytes. The other one or two bytes that are generated are randomly chosen from the interval 0-255, which is the size of a single byte.

When a combination of junk bytes that desynchronizes the code have been found, it is inserted into the file at the corresponding symbol's offset. When doing this we must take into account the number of placeholder bytes that are present and the number of junk bytes that have been generated. In the case where no combination of the same number of bytes could be found, we must insert the desynchronizing junk bytes just before the call instruction and leave the other placeholder byte or bytes that the prototype inserted.

Then it continues with the next desynchronization point's symbol in the symbol list until the whole code have been obfuscated.

```
# Pseudo code for desync_jb_generator.py
open binary
fetch desynchronization symbols from
 ↪  symbol table
get symbol offesets
while symbol in symbol list
    extract code snippet starting from
     ↪  offset of symbol
    disassemble code snippet and get
     ↪  length and instruction list
    while(not desynchronized)
        find junk bytes
        if junk bytes
            insert junk bytes to code
             ↪  snippet
            disassemble code snippet
            get disassemble length
            if desync length == original
             ↪  length AND different
             ↪  instructions at "call"
                desynchronized = TRUE
        else
            break
    if junk bytes
        insert junk bytes in original file
```

B. *Performance*

In order to ensure that the program is efficient enough, some benchmark tests were performed. This includes measurements for the amount of time, in milliseconds, consumed by each desynchronization loop for each desynchronization point, the amount of loops that were required to find suitable junk bytes, and the amount of desynchronization points the were succesfully or unsuccessfully desynchronized. The tests were run five times for each junk byte configuration available in the prototype, on a C-program called *bzip2*. The configurations include a fixed number of junk bytes between one and three, or random amount of junk bytes in the same interval. The results are presented in Table I. All desynchronization points were always successfully desynchronized.

TABLE I
BENCHMARK TESTS FOR DIFFERENT BINARIES AND CONFIGURATIONS.

Config	Run	Max ms	Min ms	Avg ms	Max (#)	Avg (#)
Fixed 1	Run 1	42.28	1.80	7.53	106	35.17
	Run 2	48.44	1.79	7.31	106	35.17
	Run 3	43.58	1.80	7.65	106	35.17
	Run 4	36.79	1.96	7.42	106	35.17
	Run 5	217.33	1.82	8.61	106	35.17
	Avg	77.69	1.83	7.70	106	35.17
Fixed 2	Run 1	27.12	1.69	4.15	96	9.85
	Run 2	32.59	1.78	3.73	94	9.91
	Run 3	25.31	1.66	3.51	77	9.46
	Run 4	21.64	1.66	3.48	98	9.38
	Run 5	22.15	1.63	3.69	104	9.52
	Avg	25.76	1.68	3.71	93.8	9.63
Fixed 3	Run 1	15.86	1.65	3.37	74	8.20
	Run 2	19.68	1.84	3.73	99	8.44
	Run 3	29.38	1.80	3.72	71	8.64
	Run 4	17.01	1.80	3.48	98	9.08
	Run 5	19.20	1.61	3.63	103	8.50
	Avg	20.19	1.74	3.59	89	8.57
Rand	Run 1	43.65	1.61	4.68	106	16.1
	Run 2	28.61	1.53	5.30	106	17.67
	Run 3	25.37	1.59	4.15	152	16.17
	Run 4	22.40	1.64	4.66	112	17.64
	Run 5	38.83	1.66	4.62	106	17.30
	Avg	31.77	1.61	4.68	116.4	16.97

C. Evaluation

Judging by the benchmark tests, the junk byte generator seems very reliable. Out of twenty runs on different configurations, each involving 474 desynchronization points, it never failed to find suitable junk bytes. When the amount of junk bytes were fixed to two and three it only took approximately 9 and 10 loops respectively. There is little difference in the minimum amount of time taken since all of them needed just one loop at best. In summary, the tool seems to perform at best when the number of junk bytes are fixed to three, as it has the best results in every category. When the configuration is set to random, it predictably performs at a level comparable to an average of all the fixed configurations combined.

Interestingly, it seemed to perform at worst when there was only one junk byte to change, as seen from the comparatively high execution times and large amount of loops needed when the number of junk bytes were fixed to a single one. A possible explanation for this would be that the op-code for the call instruction is less frequent as a suffix than a random byte. If this is the case, the strikening difference in performance by the different configurations would not be present if the desynchronization points were inserted at random or at a specific interval, rather than before every call-instruction.

A potential problem that currently exists in the junk byte generation tool is that it uses two precomputed lists containing single byte op-codes and prefixes. The problem with these lists is that there is no guarantee that they are complete or do not contain any miscategorized bytes. This could potentially give junk byte sequences that do not result in effectively desynchronized code. As the implementation is done today, the desynchronization check mechanism could potentially miss if a prefix is inserted before the call instruction and still believe this is desynchronized when in fact it is not.

D. Possible improvements

Possible improvements that could be implemented in future projects is to check how far the code have been desynchronized and choose junk bytes that gives the longest desynchronization. This would give better results because it would be harder to find the desynchronization points and understand the original control flow.

Another improvement regarding the junk byte generation, is to use some sort of machine learning to find which junk bytes gives the best results in which case and learn a model to faster find these and insert them to lower the execution time and increase the performance.

One last improvement is to change the programming language to for example C++ in order to speed up the computations and thereby increase the performance.

V. Conclusion

Choosing the junk bytes randomly in Python reliably produces desynchronized code in a fairly efficient manner. By fixating the amount of junk bytes inserted at each desynchronization point to three it will perform at best, assuming that every desynchronization point is inserted before a call instruction. However there are room for optimization, possibly by using a faster language such as C++ or by using machine learning techniques.

References

[1] Eli Bendersky. pyelftools. https://github.com/eliben/pyelftools, 2011.

[2] Wikipedia contributors. Executable and linkable format. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format, 2021. [Online; accessed 8-May-2021].

[3] Brian Gough. *An introduction to GCC*. Network Theory Limited, 2004.

[4] Vikash Kumar. Compiling a c program:- behind the scenes. https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/, 2021.

[5] 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, page 290–299, New York, NY, USA, 2003. Association for Computing Machinery.

[6] Nguyen Anh Quynh. Capstone. https://github.com/aquynh/capstone, 2013.