Determining the Quality of An Obfuscation

Corbin Souffrant and Rashid Tahir

May 12, 2013

1 Introduction

When releasing software, one common problem that companies consider is ways to protect their intellectual property. Whether this is a proprietary algorithm, a secret code, or for some other reason, it poses a problem that needs to be solved. One solution to this issue is obfuscation, which tries to solve this problem by transforming software into a form that is left unintelligible to both human and automated adversaries. Obfuscation is also used for malicious reasons, such as malware hiding its actions from anti-virus detection software.

The problem with obfuscation arises in the fact that it is not a perfect form of security. With the rise of obfuscation, came the rise in the abilities of people to be able to reverse engineer, or deobfuscate the code back into it's original form. Eventually an attacker will be able to successfully find out the function of the software.[2] Attackers have a variety of tools available to them that allow them to analyze many different areas of the software, but you can really narrow it down to two main camps: static analysis, which attempts to analyze the code itself, and dynamic analysis, which attempts to analyze the execution of the software.

There are many reasons as to why a framework to measure the strength of an obfuscation is useful. The reason that you are applying obfuscation to a program in the first place, is an additional security measure to be able to thwart attackers. When you put in time and resources to developing such a routine, you want to be sure that it will be effective. Being able to show that the obfuscation is somewhat strong, would help prove that the work you had been doing was more meaningful. Perhaps a correlation can be shown between the strength of an obfuscation routine, and the amount of time that it takes to crack it. That is, could one estimate the length of time that it takes for an attacker to successfully reverse engineer their code given a score? If you are able to hold it within some confidence interval you would have a powerful tool to aid in you software's security. Companies might even be able to feel at ease about their obfuscation routine staying strong until the next version is released, or until their estimated profit margins go below a threshold.

A good definition of security in obfuscation is a problem that needs further study. What has been found is that the cryptographic standards do not necessarily apply to the real world strength. In this paper we will discuss the factors for defining the relative strength of an obfuscation routine. We will begin in

section 2 discussing previous work and the issues that have arose, in section 3 we will begin with a background of terminology for our arguments, followed by the abilities of static and dynamic analysis analysis in sections 4 and 5 respectively. We then end with a discussion in section 6 on how the 4 factors are determined by these techniques

2 Related Work

One of the first works in defining security measures for obfuscation was published by Collberg, in his paper "A Taxonomy of Obfuscating Transformations"[2]. In this paper he gave a set of properties for obfuscated code, and its strength. These were defined as: Potency, the difficulty for a human adversary to read the code; Resiliance, the difficulty for an automated tool to deobfuscate the software; stealth, how hard it is to detect the fact that your software is obfuscated; and cost, how much performance loss there is from your obfuscated code. These terms, while providing a good starting ground for obfuscation evaluation, do not provide a way to make a comparison of the relative strength of your algorithms.

Barak[1] attempted to continue this research in which he created the notion of a virtual black box to define obfuscation. The results of this research showed that obfuscation was impossible in a general case because there are a set of properties that can not be obfuscated, and any program that contains these properties is thus impossible. Research stemmed off of this paper in order to provide positive and negative results of obfuscation, one of the more relevant ones being the idea that in best possible obfuscation[5], the obfuscated function will provide no more information than the original program itself did.

While there has been some previous study in evaluating the quality of an obfuscation algorithm, they have either been completely theoretical, or completely analytical. Studies like this do not allow for a practical framework that can be applied in software. The issue is that with cases like these, there is no easy way to generate cases that can allow for evaluation such that one could give meaningful results. What we add to these studies is a novel way of representing obfuscation strength in such a way that they can be applied to a set of analytical tools to provide a quantifier of the algorithms strength.

A similar technique has been applied in a paper previously[6], in which they looked at evaluating the quality of obfuscation by analyzing 4 different java obfuscators and used the transformations given by these tools in order to develop a mathematical model in which to rank the obfuscations. This technique is limited in the fact that it does not account for new techniques that are produced in obfuscations, as well as the fact that they are limited to a java set of tools.

3 Background

In this section, I will define the background necessary for the understanding of an obfuscation evaluation framework. As previously defined, there are 4 qualitative factors to consider when looking at the effectiveness of an obfuscation algorithm. Potency, is the difficulty for a human adversary to be able to decipher the obfuscated code. This is not a very good measure of effectiveness, an increase of the length of code or creating additional loops may make it harder for a human

to read, but not necessarily harder for an automated analysis to decipher.

In order to compensate for this, another variable, resilience, is introduced. This is a measure of how hard it is for an automated analysis to deobfuscate the software. The other 2 parameters that define the strength of the obfuscation are the cost, which measures the performance loss when applying the transformations, and the stealth which measures how hard it is to detect that there is an obfuscation.

One of the main issues in developing a metric to analyze obfuscation strength, is that one needs to define an adversary that is not too weak nor too strong. The method we chose to use for a framework was to consider an adversary who has access to a set of automated tools to aid in deciphering the code. It is known that the fact that your obfuscation will be broken if someone is given enough time.[2] This leads us to the conclusion that the potency metric will have less of an overall effect than the resilience metric. (Although it still has some effect to the degree that a person shouldn't be able to easily read through your disassembled binary).

In analyzing a metric to provide an evaluation of the quality of an obfuscation, we chose to break up the analysis into 2 sections. The first section is static analysis which will cover the methods that are used to deobfuscate text statically, the next section will cover dynamic analysis which will cover the analysis from the point of view of deobfuscation from features that you can measure while the binary is running. After we discuss the techniques in both of those events, it will allow us to discuss a feasible metric that takes into account the techniques that an adversary would use and draw a conclusion.

4 Static Analysis

Most obfuscation research has revolved around the scenario of static analysis. One paper in particular[9] attempted to prove the hardness of an automated static analysis (Although in the scenario of static analysis for malware detection). What they showed in this paper, was that static analysis can be reduced to the 3SAT problem, which makes automation of static deobfuscation NP-hard. This means static analysis is something hard to automate, since factors of the obfuscation routine can be slightly changed on each build, causing you to redo parts of your tool.

Regardless of that fact, static analysis techniques are still used, since they can prove to be extremely useful in weak obfuscations and there are a lot of powerful tools that have been developed both commercially[3] and in the research community[7]. These tools provide a plethora of useful features that help in deciphering the functionality of source code. They provide the ability to look at the control flow in detail, which can help you deduce obfuscation transformations by visualizing the call graphs.

A thesis helped to define the concepts in a static analysis[8], through their research in identifying strength in an obfuscation. The claimed that there were 3 different forms of analysis that were done with the source code: cloning, which attempts to clone out portions of execution paths in order to separate them from their original path; static path feasibility analysis, which is a constraint based

analysis that tries to decide whether or not a path is feasible; There is also a hybrid dynamic/static analysis which attempts to show a reduction in the possible paths the program actually uses by what the program is actually executing.

5 Dynamic Analysis

Unlike in the case of static analysis, there has been little research in the way of dynamic analysis in order to deobfuscate programs. No work has been done in the proof of hardness for dynamic analysis that we know of as well. This leads us to make some assumptions in some of our metrics for these scenarios, but we do not believe any of them are too restricting. Although as seen in the previous section, there has been some research that has attempted to add in additional strengths to the static analysis by combining it with a hybrid dynamic analysis to reduce the number of paths needed to study, since some might never be taken during the execution of the program.[10]

Another paper from the malware detection world[4], gave an overview of the tools that are currently used. Some of the features that dynamic analysis tools can do are: function call monitoring, this allows you to look at system calls, function parameters, and API calls; Information-flow tracking, which lets you look at how data moves through the program; Multi-Path Execution, allows an automated approach to look at every path of the control flow and determine what is happening at each branch point; and finally clustering, which allows you to statistically determine how similar functionalities are, and group them into clusters.

These techniques will give you a different approach to analyzing software, but are extremely powerful in deducing the functionality of software. For instance, a clustering algorithm could keep track of System Calls and Information Flow in order to determine if a program is doing similar things to another. This would aid in the fact that you would then be able to determine which execution paths are ones that are worth looking at.

6 Discussion

Now that we understand the tools an attacker can use to analyze an obfuscated program, we can make a claim about what makes one obfuscated program stronger than another one. We are defining our metrics off of the model where an adversary is using a combination of static and analysis tools in order to analyze the obfuscated software.

Potency: Potency is identifiable by the legibility of the code. Factors to include an analysis are control flow transformations such as loop nesting or control flow flattening, and obscuring information such as variable hiding or function parameters. These are hard to identify, since it's attempting to quantify a human factor, but they were previously defined in another metric study.[6]

Resiliance: Resilience is identifiable by things that make an analysis harder to perform. This can include tricks such as anti-debugging and anti-reversing techniques, path-execution obscuring such as putting in dummy paths or in-

creasing the number of paths that the program executes altogether. Hiding information flow can also cause an adversary to have a harder time deciphering your program's structure.

Stealth and Cost: While stealth and cost are 2 factors for an obfuscation routine, under our attack model they aren't really as important of factors to consider. Since we are under the assumption that there is an attacker actively analyzing our obfuscation routine, the stealth factor would add to if they can decide what is and isn't obfuscated, since not necessarily the entire program will be. Cost will certainly be running in polynomial time. These 2 factors will be included into the overall strength only as more of a slight curve in similar routines i.e. something more efficient that has the same potency and resilience ratings should be a better obfuscation function.

7 Conclusion

Determining the strength of your obfuscation involves an analysis of the factors listed above. Further work towards implementing a software framework for analyzing an obfuscation strength would be possible, but the point to keep in mind is that an obfuscation that runs well on one program might not necessarily work well for another. Barak proved that there was no perfect general case obfuscation[1], but the strength of an obfuscation for different programs can have different effects. What the framework needs to keep in mind is the 4 qualitative factors provided for the efficiency, and determine what is affecting each of them.

In order to obtain a more quantitative result, a framework could be developed that would be able to measure the strength. What it would involve would be implementing an automated analysis tool that looks at the different factors of static and dynamic analysis. Given this, it would judge the difficulty of the obfuscation. It's not possible to do so without an experimental experiment, but given a set of binaries, the framework would help in the aid of assigning regression weights in the quality of an obfuscation. The results given from the framework wouldn't necessarily be conclusive on the true obfuscation quality, but would provide a first step towards the identification on the strength of your routine. Because human analysis is conducted as well as an automated analysis, a framework to quantify the difficulty of how hard something is for a human would be subjective at best, but that still does not make an automated answer a useful factor, since it will be able to show the strength against an automated adversary. That is, how hard is it for a general deobfuscation program to exist.

References

[1] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Advances in CryptologyCRYPTO 2001*, pages 1–18. Springer, 2001.

[2] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[3] Chris Eagle. Attacking obfuscated code with ida pro. *Black Hat*, 2004.

[4] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[5] Shafi Goldwasser and Guy N Rothblum. On best-possible obfuscation. In *Theory of Cryptography*, pages 194–213. Springer, 2007.

[6] Matthew Karnick, Jeffrey MacBride, Sean McGinnis, Ying Tang, and Ravi Ramachandran. A qualitative analysis of java obfuscation. In *Proceedings of 10th IASTED International Conference on Software Engineering and Applications, Dallas TX, USA*, 2006.

[7] Matias Madou, Ludo Van Put, and Koen De Bosschere. Loco: An interactive code (de) obfuscation tool. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 140–144. ACM, 2006.

[8] Anirban Majumdar. *Design and evaluation of software obfuscations*. PhD thesis, The University of Auckland New Zealand, 2008.

[9] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430. IEEE, 2007.

[10] Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding*, pages 270–284. Springer, 2011.