Deobfuscation
Reverse Engineering Obfuscated Code*

Sharath K. Udupa Saumya K. Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA.
{sku, debray}@cs.arizona.edu

Matias Madou
Ghent University
St.-Pietersnieuwstraat 41
B-9000 Ghent, Belgium.
mmadou@elis.ugent.be

Abstract

In recent years, code obfuscation has attracted attention as a low cost approach to improving software security by making it difficult for attackers to understand the inner workings of proprietary software systems. This paper examines techniques for automatic deobfuscation of obfuscated programs, as a step towards reverse engineering such programs. Our results indicate that much of the effects of code obfuscation, designed to increase the difficulty of static analyses, can be defeated using simple combinations of straightforward static and dynamic analyses. Our results have applications to both software engineering and software security. In the context of software engineering, we show how dynamic analyses can be used to enhance reverse engineering, even for code that has been designed to be difficult to reverse engineer. For software security, our results serve as an attack model for code obfuscators, and can help with the development of obfuscation techniques that are more resilient to straightforward reverse engineering.

1 Introduction

In recent years, code obfuscation has attracted some attention as a low cost approach to improving software security [4, 7, 8, 21, 22, 29]. The goal of code obfuscation is to make it difficult for an attacker to reverse engineer programs. The idea is to prevent an attacker from understanding the inner workings of a program by making the obfuscated program "too difficult" to understand—that is, by making the task of reverse engineering the program "too expensive" in terms of the resources or time required to do so. Obfuscation has also been used to protect "software watermarks" and fingerprints, which are designed to thwart software piracy [1, 7, 8]. The presumption is that making it difficult for attackers to understand the internal workings of a program prevents them from discovering vulnerabilities in the code, and serves to protect the program owner's intellectual property.

It is important to note, however, that code obfuscation is merely a technique. Just as it can be used to protect software against attackers, so too it can be used to hide malicious content. For example, certain kinds of sophisticated computer viruses, e.g., polymorphic viruses, have resorted to using obfuscation techniques to prevent detection by virus scanners [27].

This raises two closely related questions. The first question, from a software engineering pespective, is: *What sorts of techniques are useful for understanding obfuscated code?* For example, suppose we have downloaded, from a web site, a file purporting to be a security patch for some application. Before applying the patch, we may want to verify that the file does not contain any malicious payload. How can we verify this if the contents of the file have been obfuscated? The second question, from a security perspective, is: *what are the weaknesses of current code obfuscation techniques, and how can we address them?* If our obfuscation schemes are ineffective in thwarting attackers from reverse engineering the code, then they are not only useless, but are in fact worse than useless: they increase the time and space requirements of the program, and can contribute to a false sense of security that keeps other security measures from being deployed. Thus, identifying any weaknesses in current obfuscation schemes by developing and testing attack models can lead to better obfuscation schemes and concomitant improvements in software security.

This paper aims to address the questions raised above, regarding techniques for understanding obfuscated code and the strengths and weaknesses of sophisticated obfuscation algorithms. We describe a suite of code transformations and program analyses that can be used to identify and remove obfuscation code and thereby help reverse engineer obfuscated programs. We use these techniques to examine the resilience of the *control flow*

*The work of S. Udupa and S. Debray was supported in part by the National Science Foundation under grants CNS-0410918 and CCR-0113633. The work of M. Madou was supported in part by Ghent University and the Fund for Scientific Research-Flanders (FWO-Flanders).