Comparing the effectiveness of commercial obfuscators against MATE attacks

Conference Paper · December 2016

DOI: 10.1145/3015135.3015143

5 authors, including:

Ramya Manikyam
University of South Alabama
3 PUBLICATIONS **8** CITATIONS
SEE PROFILE

J. Todd Mcdonald
University of South Alabama
101 PUBLICATIONS **550** CITATIONS
SEE PROFILE

Todd R. Andel
University of South Alabama
74 PUBLICATIONS **761** CITATIONS
SEE PROFILE

Samuel H. Russ
University of South Alabama
93 PUBLICATIONS **285** CITATIONS
SEE PROFILE

Comparing the Effectiveness of Commercial Obfuscators against MATE Attacks

Ramya Manikyam
University of South Alabama
rk1421@jagmail.southalabama.edu

J. Todd McDonald
University of South Alabama
jtmcdonald@southalabama.edu

William R. Mahoney
University of Nebraska, Omaha
wmahoney@unomaha.edu

Todd R. Andel
University of South Alabama
tandel@southalabama.edu

Samuel H. Russ
University of South Alabama
sruss@southalabamama.edu

ABSTRACT

The ability to protect software from malicious reverse engineering remains a challenge faced by commercial software companies who invest a large amount of resources in the development of their software product. In order to protect their investment from potential attacks such as illegal copying, tampering, and malicious reverse engineering, most companies utilize some type of protection software, also known as obfuscators, to create variants of their products that are more resilient to adversarial analysis. In this paper, we report on the effectiveness of different commercial obfuscators against traditional man-at-the-end (MATE) attacks where an adversary can utilize tools such as debuggers, disassemblers, and de-compilers as a legitimate end-user of a binary executable. Our case study includes four benchmark programs that have associated adversarial goals categorized as either comprehension or change tasks. We use traditional static and dynamic analysis techniques to identify the adversarial workload and outcomes before and after each program is transformed by a set of three commercial obfuscators. Our results confirm what is typically assumed: an adversary with a reasonable background in the computing disciplines can both comprehend and make changes to any of our completely unprotected programs using standard tools. Additionally, given the same skill set and attack approach, protected programs can still be probed to leak certain information, but none could be successfully altered and saved to create a cracked version. As a contribution, our methodology is unique compared to prior studies on obfuscation effectiveness in that we categorize adversarial skill and delineate program goals into comprehension and change ability, while considering the load time and overhead of obfuscated variants.

Categories and Subject Descriptors

• **Security and privacy~Software reverse engineering** • *Security and privacy~Penetration testing* • **Social and professional topics~Software reverse engineering** • Social and professional topics~Digital rights management

SSPREW '16, December 05-06, 2016, Los Angeles, CA, USA
© 2016 ACM. ISBN 978-1-4503-4841-6/16/09…$15.00
DOI: http://dx.doi.org/10.1145/3015135.3015143

General Terms

SECURITY, EXPERIMENTATION

Keywords

Software protection, obfuscation, commercial obfuscators, man-at-the-end (MATE) attacks, malicious reverse engineering, cracked programs, anti-tamper, static analysis, dynamic analysis

1. INTRODUCTION

Software companies invest a large amount of resources in the development of their products, which includes analysis, design, implementation, testing, and maintenance of the software. Code, data files, algorithms, and other confidential information within software can potentially be partially copied, cloned, reused, or subverted by competitors and malicious adversaries. According to a SafeNet survey in 2012 [1], 33% of global respondents believe that reverse engineering and theft have a major impact on their businesses. An overwhelming 63% of respondents see code protection to prevent reverse engineering as being a challenge. According to the BSA global software survey published in 2014, losses due to unlicensed software installation were approximately $62.7 billion in 2013 [2].

MATE (man-at-the-end) attacks describe the class of malicious actions that an adversary can take against a piece of software [3]. In particular, MATE attacks represent what a legitimate, authorized user of a piece of software might do with that software once it is in their possession. These attacks are prevalent in any setting where the intruder may have complete access to the software application or hardware device. The intruder can modify or tamper with the software applications or hardware devices available to them, which are then used for malicious intentions. This can affect both the intellectual property rights as well as an individual's privacy. For example, MATE attacks might involve tampering with software or hardware to gain unauthorized access to confidential or sensitive data, to gain illegal network bandwidth, or to give a player unfair advantage in a networked game [3].

In software development, reverse engineering is typically considered the inverse of forward engineering. Forward engineering involves taking high level languages (source code) and producing lower level languages (assembly and machine code) through compilation. Source code in forward engineering is derived normally from traditional requirements, analysis, and design activities, which capture the higher levels of abstraction in describing a problem [4, 5]. Reverse engineering begins with lower, very detailed machine code and attempts to reproduce the intermediate assembly and higher level language that produced it. Reverse engineering is used in various legal contexts such as the recovery of lost design information for maintenance purposes, identifying and fixing bugs, and identifying malicious code in a program [6]. Malicious hackers and corporate adversaries can also use reverse engineering to support MATE attacks and potentially violate intellectual property (IP) rights and copyright of the software owner.

In order to discourage or defeat such adversaries, software anti-tamper technologies like obfuscators can be used to change a program in ways that make it more difficult for MATE attacks to succeed [7]. Obfuscators take a target program, in a high or low-level representation, and produce functionally equivalent versions of the program that are potentially more difficult to understand but not necessarily impossible to reverse engineer [8]. Obfuscators are obtainable in the commercial and open source market [9-13] or from academic research communities [14]. The goal of most obfuscators is to attempt to defeat automated analysis tools such as debuggers [15-17] and de-compilers [18-20], while also making the code as meaningless and unintelligible as it can be to a human analyst. Nagra and Collberg [29] delineate four major categories of obfuscating transformations: 1) **abstract**–changes original program organization and design information; 2) **control**–modifies branch and looping constructs to hide runtime pathways; 3) **data**– replaces data structures and text with structures that reveal less information; and 4) **dynamic**–inserts a transformer into the program so that the program continues to change itself at runtime. In addition, anti-tampering techniques can employ code checksums to ensure integrity of execution.

We can measure the effectiveness of an obfuscator partially by whether it is successful in preventing certain MATE attacks. If a MATE attack is successful in an unprotected/un-obfuscated program and the same MATE attack, given the same amount of resources, is not successful in a protected/obfuscated variant, then we can demonstrate a quantitative measure for whether the obfuscator has achieved a particular security goal. In this paper, we measure the effectiveness of different commercial obfuscators on a set of programs that embody common software security challenges. We determine and report on whether such obfuscators can prevent specific MATE attacks and analyze the respective cost/performance tradeoff that incurs from using the obfuscator itself. Our study includes programs in the form of both executable and high-level language code.

The rest of the paper is organized as follows: section 2 provides related work and background which introduce terminology and concepts used; section 3 discusses methodology and steps followed in conducting our case study; section 4 gives the results of the commercial obfuscator case study; finally, we present conclusions and future work in section 5.

2. RELATED WORK AND BACKGROUND

Several areas of computer science and security research inform our case study approach and analysis: we review several of these foundational concepts next.

2.1 Program Analysis

Program analysis can be defined as studying a program, typically with automated tools, and gathering properties about the program such as correctness, liveness, safety, and its ability to recover from errors [21]. Two major types of program analysis include static and dynamic analysis. Static analysis tools collect information based only on the code, in potentially different forms such as at the source or binary level, without executing the code itself. There are two tradeoffs when performing static analysis: 1) the precision of the information that is to be gathered from the analysis and 2) the amount of effort used for runtime performance [22]. Dynamic analysis tools collect information by actually executing a program with specific inputs, tracing internal variable states, and recording overall output. Dynamic analysis allows observation of those parts of a program that executes at any given time, based on different user input. Debugging, profiling and tracing are a few widely used dynamic analysis techniques to extract program properties. For example, the Linux *gprof* program can take advantage of profiled compilation to report call graphs and timing information of user-space code.

2.2 Man-at-the-End (MATE) Attacks

In traditional network security, a man-in-the-middle attack is an attack by malicious intruders who try to gain access to information by impersonating legitimate senders or receivers in a network protocol. In contrast, a man-at-the-end (MATE) attack involves a malicious intruder who has unlimited and authorized physical access to hardware or software involved in the processing of data [3]. MATE attacks therefore represent the potential actions of someone who has legally purchased a piece of software and the potential analysis that they can perform as a legitimate user. A MATE attacker (legal user) who possesses the program code in executable form has several options at their disposal, which include: 1) examining the executable (.exe) file using a disassembler to recreate the assembly language; 2) using the disassembled code to recreate the high level language with a de-compiler, recreating the high level language; 3) running the program in a debugger, examining memory locations, registers, program state, and instructions while it is executing, and/or observing memory locations while the program is running. Man-at-the-end attacks include piracy, tampering, reverse engineering, breaking digital rights management, and protocol discovery.

2.2.1 MATE Attack Model

Man-at-the-end attacks are difficult to defeat because adversaries often use creativity, motivation and ingenuity to tamper with normal program execution. The attacker also has limitless authorized physical access to hardware or software systems to perform primary MATE attacks. Attacks generally follow two stages: an analysis stage and transformation stage [21]. In the analysis stage, information about a program is collected and in the transformation stage, an attacker tries to modify the program based on the information from their analysis.

In the analysis stage, there are several overlapping and repeating phases: 1) a black-box phase where the outputs are recorded based on the inputs given to a program and then the attacker draws the conclusion about the behavior of the program required to perform primary MATE attacks; 2) a dynamic analysis phase where the program is executed repeatedly to record which parts of program gets executed for different inputs; and 3) a static analysis phase where the executable can be directly examined by using disassemblers and de-compilers. In the transformation stage, after drawing the conclusions based on the internal behavior of the programs, the attacker may 1) modify the executable to disable license checks, to alter data in memory locations, to disrupt the integrity of software, to violate vendor's IP rights and/or to violate copyright laws; and 2) the attacker may encapsulate knowledge of the attack in automated script to use in future attacks.

2.2.2 Cracked Programs

MATE attacks may ultimately seek to make unauthorized modifications to code in order to change the behavior of the software; we call this activity code tampering. In order to alter the code to bypass the licensing check, legal user has to successfully analyze the software (through disassembly / de-compilation / debugging), find the location of the licensing check code which checks current date, and successfully changes the code to never abort the license check. Code altering could occur only at runtime, affecting a program process currently in memory, or the altering can be made permanent through changes in the actual executable file. “Cracking” is the term that describes this process of permanent code alteration and modification.

2.3 Measuring Obfuscation Effectiveness

The ability to measure the effectiveness of obfuscation techniques depends on how the obfuscation problem is framed. Measurement can be done through a formal, theoretic definition that typically involves the limits of power on Turing machines or circuits; likewise, heuristic measurements can be defined which reflect the effectiveness of obfuscation and apply those measurements to real-world examples through empirical studies.

2.3.1 Theoretical and Formal Models

All theoretical research on software protection via obfuscation typically points to negative results in terms of the existence of perfect obfuscators. Barak et al. [8] showed that no general obfuscation algorithm exists that can hide all information leaked by a variant program based on the notion of a virtual black box. The basic impossibility result states that it is impossible to achieve perfect semantic security where the variant leaks no more information than the input/output relationships of the original program.

2.3.2 Heuristic and Empirical Approaches

A heuristic approach to measuring obfuscation involves defining characteristics or properties that reflect the strength of a transformation algorithm in achieving goals related to program analysis or reverse engineering. Collberg et al. [23] were the first to propose taxonomy of obfuscating transformations as early as 1997. In their taxonomy, the effectiveness of obfuscation transformations can be measured by three metrics: resilience, cost and potency. Resilience is defined as how good the software is at withstanding an attack from an automatic deobfuscator, whereas potency is defined by the difficulty in understanding the obfuscated program. The cost is defined in terms of the overhead added to the application. A good obfuscation transformation is one which has high resilience and high potency but low costs (in terms of increased execution time and additional memory usage). The disadvantage in measuring the strength of obfuscation technique is that a well-defined security level does not exist. It is impossible to protect software completely from reverse engineering, since ultimately the software still must function according to the requirements. But the techniques used in software obfuscation can make it harder to read, analyze, or understand.

There are several empirical results in the literature that help clarify and frame our research approach. While virtualization of software has become one of the prominent ways to protect intellectual property, Cazalas et al. [24] has shown that both static and dynamic virtualization is still potentially vulnerable to end-user subversion. Static virtualization involves the protection of intellectual property by using process level virtualization. Dynamic process virtualization is similar, but the program can generate instruction set architecture (ISA) at run time. Results show that Code Virtualizer and Strata provided general protection against automated unpacking, automated code injection, and general static analysis. However, Code Virtualizer was still vulnerable to analysis and brute force attack, as well as unpacking.

Collberg et al. [25] provided a framework for heuristics in 2002 and defined four basic properties to measure and compare obfuscation approaches: 1) obscurity, 2) resilience, 3) stealth, and 4) cost. Obfuscation level increases as obscurity, resilience, or stealth of the transformed program, *P’*, are maximized. As the obscurity of *P’* is maximized, comprehension and reverse engineering of *P’* will be more challenging, resulting in an increase in time consumption. Thus, the end goal is to maximize obscurity while minimizing cost to achieve the best form of obfuscation.

Macrbride et al. [26] performed a comparative study of two commercial java obfuscators and studied the performance of the KlassMaster and DashO-Pro obfuscators which use control flow transformations. They measure qualitative performance by using three different sorting algorithms: QuickSort, BubbleSort and RadixSort. Their results show that all obfuscated variants increase computational complexity compared to the running time of the original algorithms. Their study also indicated an interesting relationship between the code size/cyclomatic complexity of the original sorting algorithms and their obfuscated variants. In particular, cyclomatic complexity actually had an inverse relationship to performance loss in the obfuscated variants.

In 2014, Ceccato et al. [27] performed an empirical study in order to quantify and compare the effectiveness of code obfuscation. This study, while having limitations as suggested by the authors, shows that comprehension and source code changing difficulty escalates with the addition of two types of obfuscation techniques: identifier renaming and opaque predicates. Evaluation of effectiveness is calculated based on correctness, time, and efficiency. Correctness is a Boolean value which defines if the task was performed correctly (value 1) or not (value 0). Time accounts for the total time consumed to perform a given task. Efficiency is defined by calculating the number of correctly performed tasks in one minute. Using Fisher’s test, the results showed that no statistical significance can be observed based on the comprehension task, signifying that obfuscation may not impact the possibility of correctly accomplishing program comprehension. However, the study did show that correct comprehension takes approximately 7 and 4 times longer for clear code that is modified by identifier renaming and opaque predicates respectively. Analysis of task completion time suggests that identifier renaming requires more time compared to opaque predicates. Also, both obfuscation techniques caused a significantly longer time to successfully change code than when working with clear code. Overall, the simpler technique of identifier renaming actually provided a better resilience against human-based attacks. As expected, the results show that after enough time, even with obfuscated code, the software can still be understood and tampered with.

3. EXPERIMENTAL CASE STUDY

The goal of our research is to measure the effectiveness of different commercial obfuscators on a set of programs that embody common software security challenges. In order to accomplish this goal, we conducted a series of experiments to illuminate the difficulty of reverse engineering on software programs that have been transformed by a commercial obfuscation program. As a first step, we chose a set of 4 benchmark programs on which we perform adversarial analysis and reverse engineering (section 3.1). Next, for each program *P* in the benchmark set, we identify a specific set of adversarial goals that are unique to each program (section 3.2). Our specific goals include both reconnaissance activities (*comprehension tasks*) and implementation of realized exploits (*change tasks*). The output of an adversarial goal is either yes or no depending upon the methodologies and techniques used in the form of MATE attacks. We then use different static and dynamic analysis tools on each benchmark program (that is unprotected) in order to achieve the adversarial goals that are outlined (section 3.3). Next, we use a set of commercial obfuscators to protect each benchmark program (section 3.4). We then perform the identical static and dynamic analysis activities on each obfuscated variant to see if the adversarial goals can be achieved using the same techniques that we used on the unprotected version. Figure 1 illustrates our case study approach.

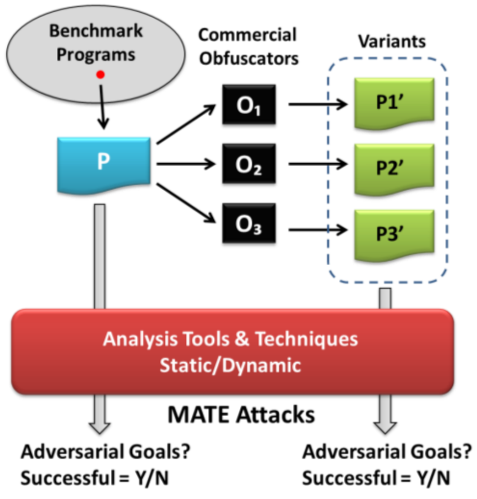

Benchmark Programs
Commercial Obfuscators
Variants
P
O1
O2
O3
P1'
P2'
P3'
Analysis Tools & Techniques Static/Dynamic
MATE Attacks
Adversarial Goals? Successful = Y/N
Adversarial Goals? Successful = Y/N

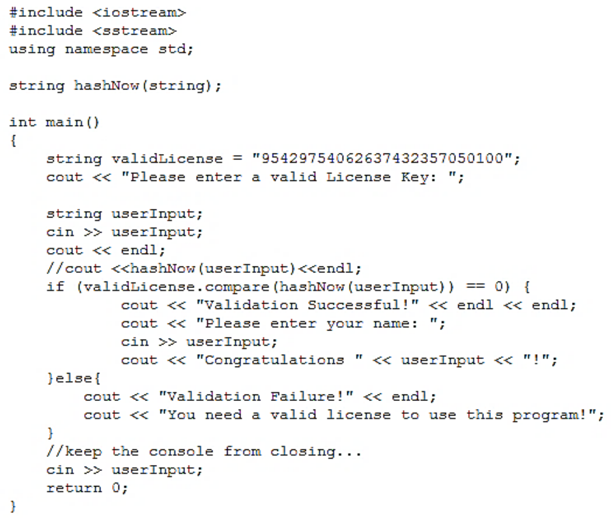
Figure 1. Case Study Methodology

3.1 Benchmark Programs

To perform our experiment, we first select programs that will be the target of MATE attacks and that will also be protected using commercial obfuscation tools. We narrowed the study to programs that run on Windows and have a reasonably small size in terms of source code or assembly language instructions. We made this decision partially because of familiarity: in prior work we used two of the programs (Microsoft 3-D Pinball and Tetris) to teach cyber security and ethical hacking principles in information assurance curriculum and as part of high school field trip activities. Next, we focus on programs that have unique functions, but embody key operations that are found in commercial software with significant monetary investment. We also chose programs where an adversarial goal can be easily identified and related to either a *comprehension* or *change* task that results in either failure or success. We postulated that commercial obfuscators, which require payment for their use, should provide a return on investment when they are used, even to protect small or simple programs. For functionality, we focused on four program features: 1) licensed usage, 2) license code verification, 3) random number usage, and 4) identification of an algorithm for determining program outcome. In order to take advantage of certain obfuscator features, we also needed at least one program written at the source code level. To fulfill these requirements we chose Microsoft 3-D Pinball, Tetris, WinRAR, and a custom C++ license check program (the source code is provided in Figure 2).

3.2 Adversarial Goals

The adversarial goals are unique to each program and depending on the goal, we perform MATE attack using static and dynamic analysis to either *comprehend* program information or actually *change* the program executable either at runtime or permanently by creating a "cracked" version. We define goals that link an observable program outcome or piece of program information to some underlying code that produces that output.



```
#include <iostream>
#include <sstream>
using namespace std;

string hashNow(string);

int main()
{
    string validLicense = "95429754062637432357050100";
    cout << "Please enter a valid License Key: ";

    string userInput;
    cin >> userInput;
    cout << endl;
    //cout <<hashNow(userInput)<<endl;
    if (validLicense.compare(hashNow(userInput)) == 0) {
            cout << "Validation Successful!" << endl << endl;
            cout << "Please enter your name: ";
            cin >> userInput;
            cout << "Congratulations " << userInput << "!";
    }else{
        cout << "Validation Failure!" << endl;
        cout << "You need a valid license to use this program!";
    }
    //keep the console from closing...
    cin >> userInput;
    return 0;
}
```

Figure 2. Custom C++ License Check Source Code

For each program, we delineate a set of tasks that are accomplished through program analysis. Each task is either a *comprehension* task or a *change* task depending on our adversarial goal. Successful completion of the task indicates that an adversarial goal has been accomplished. Comprehension tasks represent the ability of the attacker to understand the logic of the program whereas a change task represents the ability for the attacker to perform a permanent modification of the executable program that essentially creates a cracked version of the program. Table 1 summarizes all of the tasks and goals that we use in our experimental methodology where CMP refers to a *comprehension* task and CHG refers to a *change* task.

Table 1. Adversarial Goals

P	Task	Type	Goals
3-D Pinball	T1	CMP	Each game uses 3 pinballs. Identify memory location where the number of pinballs is stored.
	T2	CMP	Game ends when pinballs left reach 0: identify code section that checks for game over condition.
	T3	CHG	Invalidate code section that checks for game over condition by freezing
	T4	CHG	Create cracked version with infinite number of pinballs.
Tetris	T1	CMP	Given 7 types of Tetris block chosen randomly, identify memory location of random function used in choice.
	T2	CMP	Identify section of code where Tetris block is chosen
	T3	CHG	Invalidate code section that makes random selection with NOPs.
	T4	CHG	Create cracked version with one Tetris block.
WinRAR	T1	CMP	Identify code section that checks expiration of trial period, which expires 40 days after download.
	T2	CHG	Invalidate code section that checks trial expiration by changing jump from conditional to unconditional.
	T3	CHG	Create cracked version with no purchase requirement.
Custom	T1	CMP	Identify code section that compares hash value of stored license check with license entered by user.
	T2	CHG	Invalidate code that checks license key for validation by changing from conditional to unconditional.
	T3	CHG	Create cracked version with a bypassed license check.

3.3 Attack Methodologies

For a MATE attack to be considered successful, it must satisfy the adversary goals for the targeted benchmark program as displayed in Table 1. We performed MATE attacks on an unmodified executable first in order to form a basis for comparison. We then transformed each program using different forms of obfuscation techniques offered by commercial tools in order to compare their capabilities. Each experiment included one or more comprehension and change tasks. The comprehension tasks required the user to perform reconnaissance techniques using static/dynamic tools including Cheat Engine [17], OllyDbg [15], and IDA Pro [16]. Given successful reconnaissance through comprehension, we conducted change tasks to determine if a permanently modified cracked version could be produced. We evaluated whether the fourteen tasks in Table 1 were successful.

3.4 Commercial Obfuscation Tools

There are many commercial obfuscators available on the market, which use various techniques to protect software from malicious reverse engineering or tampering. We chose the following obfuscation tools primarily based on availability from our own prior grant funding to support program protection research: 1) VMProtect [9]; 2) Themida [10]; and 3) Code Virtualizer [11]. Table 2 provides a summary of the standard protection techniques offered by these commercial products where VMP refers to VMProtect, TH refers to Themida, and CV refers to Code Virtualizer. In table 2, the category of the transformation technique, from [29], is delineated at abstract (AB), data (DT), control (CF), dynamic (DY), and tamper-proofing (TP).

Table 2. Standard Features for each Commercial Obfuscator

	Product			Category				
	VMP	TH	CV	AB	DT	CF	DY	TP
Full Mutation	X	X	X	X	X	X		
Entry Point Obfuscation	X	X	X			X		
Anti-Debugger Protection	X	X						X
Anti-Virtualization	X							X
Memory/Import Protection	X							X
Advanced API-Wrapping		X		X				X
Tamper-Detection		X						X
Fake Stack Emulation			X	X				
String Virtualization			X		X			

We delineate two categories of techniques used by the commercial obfuscators we used in our case study: *standard features* are options provided by most of the obfuscator whereas *all features* are combinations of techniques unique to each obfuscator including their standard features. The set of standard features that are available in most of the obfuscators are: 1) **Anti-debugger protection** which provides sophisticated protection by implementing an algorithm to detect if a static or dynamic debugger is trying to attach onto the executable; 2) **Entry point obfuscation** which is an anti-heuristic method that encrypts the entry point of the application, executing it from a secure location in memory; 3) **Full mutation** in which the mutation method applied is based on obfuscation which adds excessive garbage commands, dead parts, and random conditional jumps; 4) **Anti-Virtualization** which disallows the application being run in a virtualized environment; 5) **Memory and Import protection** which provides protection against memory patching and importing libraries by the adversary; 6) **Advanced API-Wrapping** which is an anti-API scanners technique that avoids reconstruction of original application; 7) **Anti-Patching** which checks for the modification in the protected application by a cracker or a virus; 8) **Fake stack emulation** which will assume the stack is initialized to zero for all non-allocated entries; and 9) **String Virtualizer** which provides protection to strings inside macro markers.

Each obfuscator also had a large number of additional features and custom techniques. We organized our case study to also consider whether using all available protection options in each commercial obfuscator would produce any appreciable difference in the outcome for adversary analysis. We specified tests using unique additional techniques as *all features*.

3.5 Measuring Protection Effectiveness

In other case studies on obfuscation effectiveness [27] or assessment of hacker skills [28], experiments relied on a large pool of potential programmers and test subjects to assess cognitively or subjectively the skill of those involved. These studies allowed a broad assessment of skill or time required to accomplish specific tasks related to reverse engineering and program deobfuscation, mainly from the perspective of the human involved. As an advantage, these studies did not require a long period of time to conduct. In our approach, we used only one reverse engineer to assess effectiveness of obfuscation techniques and answer questions related to effectiveness. In particular, we focused more on the typical knowledge gained and acquired by someone with prior education in computing disciplines that is learning how to use particular static and dynamic program analysis tools. As a result, our study took a longer period of time for knowledge gain and execution of the case study, with more specialized tools and skill sets required. It would not be feasible to accomplish this with a larger number of students or participants in the context of a college academic environment.

3.5.1 Profiling a Knowledgeable Engineer

In their study on classification of hackers by knowledge exchange behavior, Zhang et al. [28] classify hackers into four categories: 1) **Guru hackers** are knowledgeable and share ideas with others; 2) **Casual hackers** are observers who derive useful information from forums; 3) **Learning hackers** are experts who learn from the forums; and 4) **Novice hackers** are new learners that use forums for shorter time periods of learning. We acknowledge that there is a vast difference among potential human reverse engineers in terms of skill. For this case study, we propose different skill sets in decreasing order of expertise as *Ninja Engineer*, *Skilled Engineer*, *Knowledgeable Engineer*, and *Novice* similar to the categories of hackers proposed by [28], shown in Table 3.

Table 3. Proposed Categories of Malicious Reverse Engineers

Category	Tool Use	Skilled Use	Education Training	Experience Applied Skill
Ninja	5	5	2-5	4-5
Skilled	3-5	3-5	2-5	2-4
Knowledgeable	2	1-2	2-3	0-1
Novice	0-1	0	0-1	0

Obviously, our categories are subjective assessments of skill that are determined largely by education, hands-on experience, and background of the reverse engineer. Table 3 summarizes our subjective categories of reverse engineer skill sets with the ratings assigned in ranges of 0 to 5. *Tool Use* describes the ability to potentially use MATE-supporting analysis tools without in-depth knowledge of how they work. *Skilled Use* describes the ability to expertly use static or dynamic analysis tools or automated processes. *Education Training* represents academic background in the computing disciplines: 0 and 1 indicate potentially High School level of education or some form of training (online) with tutorials or hacker videos/material, 2 indicates Bachelor's level, 3 indicates Master's level, 4 indicates doctoral level or specialized Master's research in security, 5 indicates graduate level education and specialized training in software security and program analysis. *Experience Applied Skill* represents specific skill in applied reverse engineering, with 5 being on par with a malware analyst in a top malware analysis research laboratory /company or elite programmers in hacker groups like Anonymous or Legion of Doom.

We define a *Ninja Engineer,* for example, that is able to perform a wide variety of analysis that goes beyond what a tool or automated process may produce in terms of comprehension or code change ability. At the other extreme, a *Novice* represents someone with no in-depth knowledge of computer science or information systems, but can potentially mechanically use a tool such as a debugger, fuzzer, or script-kiddy routine. For the purposes of our experiment, we have one reverse engineer who can be represented as a *Knowledgeable Engineer*: someone with an academic background in one of the computing disciplines (Information Technology, Information Systems, Computer Science, etc.), but who has never spent any considerable time previously attempting to reverse engineer or subvert an application. This category represents an end-user who can read technical material on malicious reverse engineering and understand the basic concepts described. In addition, a *Knowledgeable Engineer* can use various adversarial tools (such as IdaPro, OllyDbg, and Cheat Engine) with some level of skill because they have an academic background. A *Skilled Engineer* shares the same traits as a *Knowledgeable Engineer*, but they have potentially specialized in educational background involving security, cryptography, or program analysis. Likewise, a *Skilled Engineer* has spent some considerable amount of time understanding and applying theory related to program analysis and reverse engineering.

Our experiments are naturally affected by internal validity because our single *Knowledgeable Engineer* gained skill over the course of a typical Master's degree program in computer science. As a result, the pace at which analysis tasks were fulfilled improved over time compared to previous attempts. This induced learning effect influenced results, but we note that our reverse engineer did not progress to a higher skill tier (i.e., Skilled Engineer) as a result of performing the case study.

3.5.2 Case Study Hypotheses

Based on our definition of adversary goals in Table 1, we formulated a set of null hypotheses that guided the interpretation of our experimental results. Table 4 summarizes this set of hypotheses. We define three hypotheses (H01-H03) that are used to evaluate the strength of standard obfuscation techniques shared by all three obfuscators (*standard features* seen in Table 2). We define three other hypotheses (H04-H06) to evaluate the strength of each obfuscator's best set of all techniques (*all features*). These one-tailed (one-direction) investigations characterize the quality of the resulting obfuscated variant to decrease an attacker's capability for three things: 1) comprehend/understand a program as part of reconnaissance activities (H01/H04); 2) change a program at runtime (H02/H05); and 3) change a program permanently by creating a cracked executable (H03/H06). Hypothesis H07, H08 and H09 are two-tailed (bi-directional) experiments that describe the quality difference between common and all techniques in commercial obfuscators.

Table 4. Formulated Null Hypotheses

On Quality of Standard Obfuscator Techniques	
H_{01}	Standard techniques do not significantly decrease capability of an attacker to perform a comprehension task.
H_{02}	Standard techniques do not significantly decrease capability of an attacker to perform a change task at runtime.
H_{03}	Standard techniques do not significantly decrease capability of an attacker to perform a permanent change task.
On Quality of All Obfuscator Techniques	
H_{04}	All techniques do not significantly decrease capability of an attacker to perform a comprehension task.
H_{05}	All techniques do not significantly decrease capability of an attacker to perform a change task at runtime.
H_{06}	All techniques do not significantly decrease capability of an attacker to perform a permanent change task.
On Comparison of Standard vs All Obfuscator Techniques	
H_{07}	There is no difference between *standard* techniques and *all* techniques used by an obfuscator in significantly decreasing capability of an attacker to perform a comprehension task.
H_{08}	There is no difference between *standard* techniques and *all* techniques used by an obfuscator in significantly decreasing capability of an attacker to perform a change task at runtime.
H_{09}	There is no difference between *standard* techniques and *all* techniques used by an obfuscator in significantly decreasing capability of an attacker to perform permanent change task.

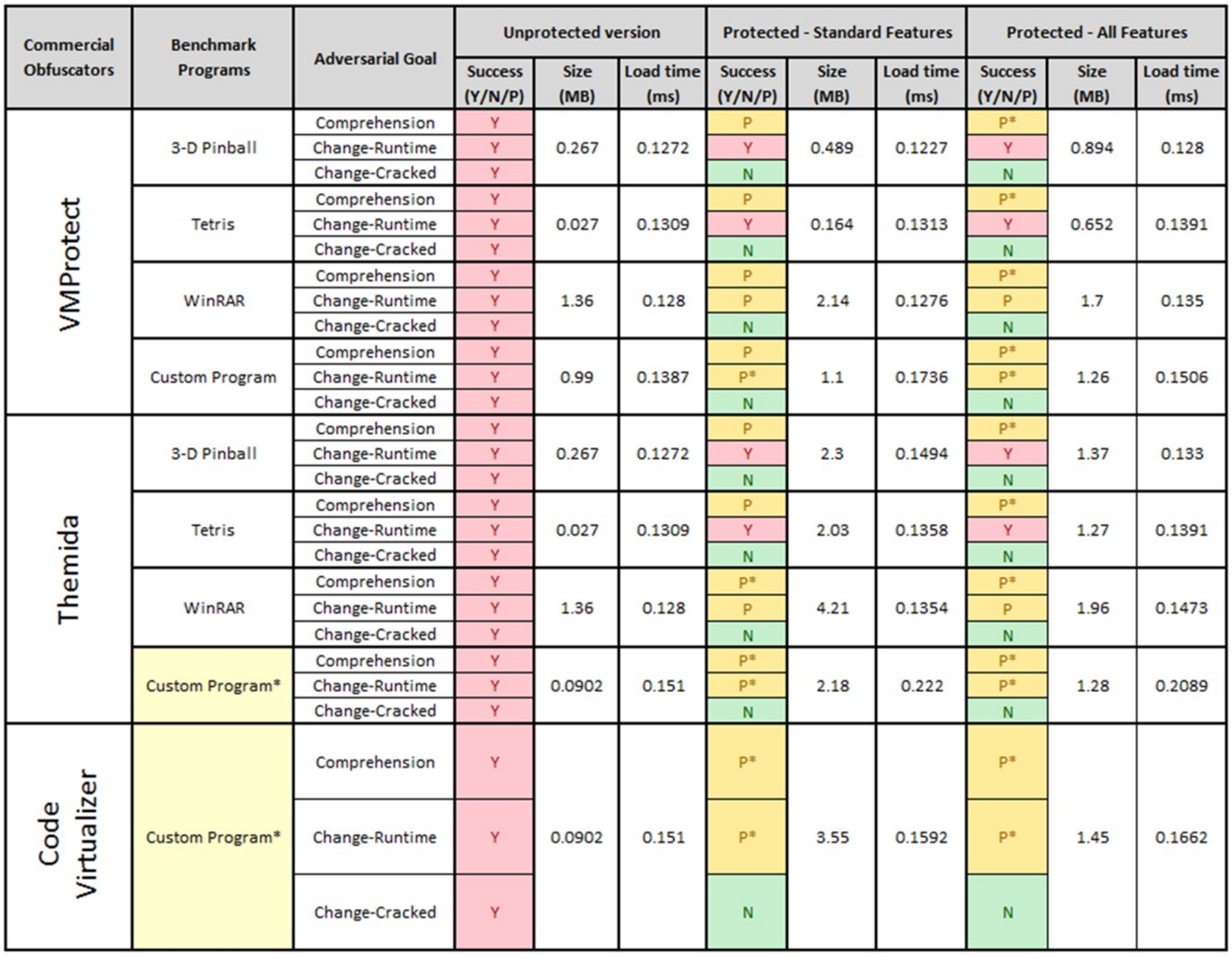
These null hypotheses suggest two dependent variables, i.e., the capability of performing *comprehension* tasks, and the capability of performing *change* tasks (at runtime or with a permanently cracked version of the program). The tasks in Table 1 are designed in such a way that there is either success or failure. To measure successful performance of a comprehension task, the adversary must be able to locate the desired sensitive data in memory or code. To measure the success of the change tasks, the adversary must be able to modify the binary at runtime and also produce a permanently modified executable (cracked) version with the desired results from Table 1. Generally, a null hypothesis is assumed to be true until proven wrong. In our experimental methodology, the adversary wins if the null hypothesis is assumed to be true and the obfuscator wins if the null hypothesis is proven false. Based on the outcome of our experiments, we interpret the results of our hypothesis to be either true (the adversary wins) or false (the obfuscator wins).

This set of null hypotheses produces an abstract analysis in terms of the ability (effectiveness) of an attacker to succeed in achieving concrete goals. Effectiveness is the ability of the attacker to perform the attack, resulting in either a success or fail result. However, we also measure load-time and size overhead of obfuscated variants. Load-time refers to the amount of time taken by the application to be loaded to execution state by the operating system. Overhead refers to the size of the process space in memory for a program.

4. RESULTS

In executing the case study, we measured the performance of each commercial obfuscator in terms of 1) *effectiveness*: ability of the attacker to perform the attack in terms of yes, no and partial; 2) *Load-time*: Loading time in milliseconds gathered through Windows PowerShell measure command; and 3) *overhead*: size or memory increase. As a key result, we observe that in unprotected

Table 5. Case Study Results



Commercial Obfuscators	Benchmark Programs	Adversarial Goal	Unprotected version			Protected - Standard Features			Protected - All Features		
			Success (Y/N/P)	Size (MB)	Load time (ms)	Success (Y/N/P)	Size (MB)	Load time (ms)	Success (Y/N/P)	Size (MB)	Load time (ms)
VMProtect	3-D Pinball	Comprehension	Y	0.267	0.1272	P	0.489	0.1227	P*	0.894	0.128
		Change-Runtime	Y			Y			Y		
		Change-Cracked	Y			N			N		
	Tetris	Comprehension	Y	0.027	0.1309	P	0.164	0.1313	P*	0.652	0.1391
		Change-Runtime	Y			Y			Y		
		Change-Cracked	Y			N			N		
	WinRAR	Comprehension	Y	1.36	0.128	P	2.14	0.1276	P*	1.7	0.135
		Change-Runtime	Y			P			P		
		Change-Cracked	Y			N			N		
	Custom Program	Comprehension	Y	0.99	0.1387	P	1.1	0.1736	P*	1.26	0.1506
		Change-Runtime	Y			P*			P*		
		Change-Cracked	Y			N			N		
Themida	3-D Pinball	Comprehension	Y	0.267	0.1272	P	2.3	0.1494	P*	1.37	0.133
		Change-Runtime	Y			Y			Y		
		Change-Cracked	Y			N			N		
	Tetris	Comprehension	Y	0.027	0.1309	P	2.03	0.1358	P*	1.27	0.1391
		Change-Runtime	Y			Y			Y		
		Change-Cracked	Y			N			N		
	WinRAR	Comprehension	Y	1.36	0.128	P*	4.21	0.1354	P*	1.96	0.1473
		Change-Runtime	Y			P			P		
		Change-Cracked	Y			N			N		
	Custom Program*	Comprehension	Y	0.0902	0.151	P*	2.18	0.222	P*	1.28	0.2089
		Change-Runtime	Y			P*			P*		
		Change-Cracked	Y			N			N		
Code Virtualizer	Custom Program*	Comprehension	Y	0.0902	0.151	P*	3.55	0.1592	P*	1.45	0.1662
		Change-Runtime	Y			P*			P*		
		Change-Cracked	Y			N			N		

versions of all four of our benchmark programs, an adversary with a reasonable computing background (our sample *Knowledgeable Engineer*) could achieve all major goals including comprehension, making runtime changes, and making permanent changes. Table 5 provides a summary of baseline values for size and load-time for 3-D Pinball, Tetris, WinRAR, and our custom program. We explain table notations and values next.

4.1 Comparative Analysis by Obfuscator

VMProtect: When using VMProtect to create obfuscated variants, we could not create cracked versions of any of the benchmark programs using either *standard* features or *all* features of the product. However, the knowledgeable engineer was able to make changes during runtime to protected versions of 3-D pinball and Tetris when both standard features and all features of VMProtect were used. In case of WinRAR, the opcode and hex view were visible using Cheat Engine (denoted as *P*) and in the custom program, opcodes were visible but not the hex view (denoted as *P**).

Our knowledgeable engineer was also successful in the *comprehension* task that consists of finding required data in a specific memory location in all the protected (standard feature) versions with the help of both static and dynamic analyzers, but required more time when compared with unprotected versions (denoted by *P*). The same task in the protected (all features) versions was possible only with dynamic analyzers but not static analyzers (denoted by *P**).

In terms of performance/cost of the obfuscator, load time was not significantly different, but the percentage increase in size varied from 11% to 2314%. In three out of the four program cases, additional features incurred greater size overhead.

Themida: When using Themida to create obfuscated variants, our knowledgeable engineer could not create cracked versions of any of the benchmark programs that were protected with either standard features or all features of the product. Themida was not able to protect the benchmark programs 3–D Pinball and Tetris against run-time modifications, but it did provide some protection in WinRAR and in our custom program. In WinRAR, the op codes and hex values are visible with the help of dynamic analysis (denoted by *P*) and in our custom program, the opcode was visible, but not hex values (denoted by *P**).

Our knowledgeable engineer accomplished the comprehension task required for finding data in memory locations for 3-D Pinball and Tetris when these programs were protected with standard features of the obfuscators, with the help of static and dynamic
)

analysis (denoted by *P*). When protected with all features of the obfuscators, comprehension tasks with 3-D Pinball and Tetris were successful only with the help of dynamic analysis (denoted by *P**). The knowledgeable engineer was only successful in the comprehension task for protected (standard and all) versions of WinRAR and the custom program only with the help of dynamic analysis (denoted by *P**).

We observed little difference in load time between unprotected programs and protected versions created with Themida. Themida's compression feature is used when all features are chosen to reduce the size of the obfuscated variants. The difference can be seen in protected (standard features) which doesn't use compression feature and protected (all features) which uses compression feature.

Code Virtualizer: Code Virtualizer requires source code in order to produce virtualizing obfuscation as a protection technique, and we only had 1 benchmark program in source code form. Therefore, we only tested our custom program using Code Virtualizer with standard and all features. When using Code Virtualizer, we were not able to create cracked versions of our protected program using either feature set (denoted by *N*). Making changes at runtime using dynamic analysis showed the opcode but not hex values (denoted by *P**), and performing the comprehension task was possible only with the help of dynamic analysis but not static analysis (denoted by *P**).

There was little difference in recorded load time between the unprotected program and protected versions using Code Virtualizer. However, from a user experience perspective, we noticed approximately 2-3 minutes delay while validating the license key. The size of the program increased in protected versions using standard techniques to 3835% (when compression was not used), but when compression was used as part of all features, we observed size increase of only 1507%.

4.2 Comparative Analysis by Program Goal

In terms of effectiveness, load time, and overhead, the three commercial obfuscators that we evaluated shared similar outcomes.

3-D Pinball: The adversarial goal associated with 3-D Pinball was geared toward changing program execution integrity, where an adversary can change the outcome of a program by making observations of runtime behavior and then changing memory locations to affect a certain outcome. In this regard, both commercial obfuscators we tested (VMProtect and Themida) were ineffective at protecting from adversarial comprehension (reconnaissance) in locating memory addresses of key values or from changing those values at runtime through disassembly and memory changes. However, both obfuscators were able to prevent cracked versions that permanently implemented adversarial changes. In terms of load time, both obfuscators produced variants with negligible increase in load time performance. In terms of overhead, Themida produced considerably larger variants than VMProtect. Figure 3 provides a summary analysis of overhead for standard features used by each obfuscator and Figure 4 provides a summary analysis of overhead for all features used by each obfuscator.

Tetris: The adversarial goal associated with Tetris was geared toward programs that use random numbers for key operations, and whether an adversary could successfully replace random numbers with known values. In this regard, both commercial obfuscators we tested (VMProtect and Themida) were ineffective at protection from adversarial comprehension (reconnaissance) in locating memory addresses associated with the random number generator or overriding the randomness at runtime (through disassembly and memory changes). However, both obfuscators were able to prevent cracked versions that permanently implemented adversarial changes. In terms of load time, both obfuscators produced variants with negligible increase in load time performance. In terms of overhead, Themida produced considerably larger variants than VMProtect.

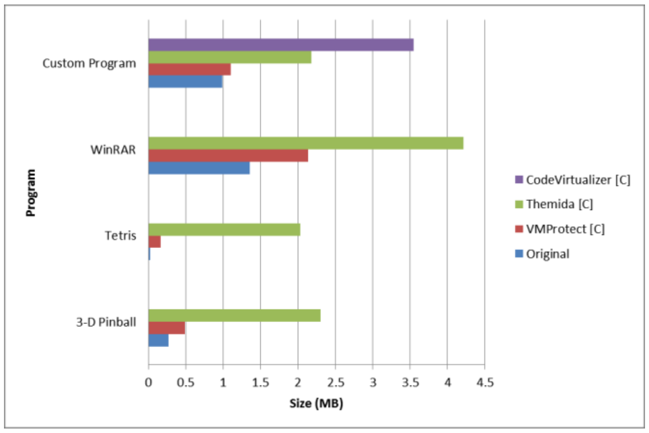


Figure 3. Overhead Analysis (Standard Features)

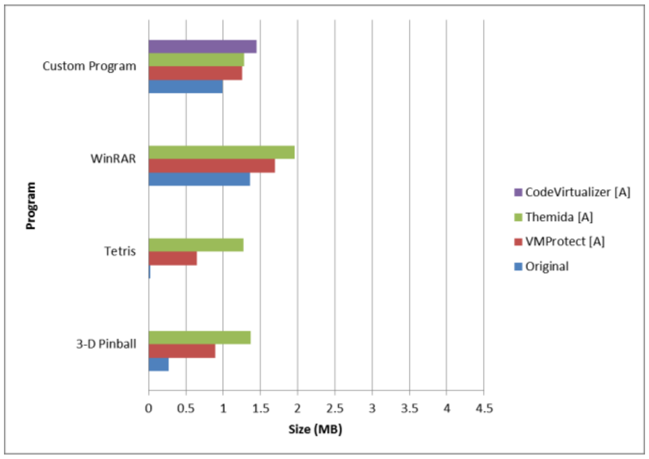


Figure 4. Overhead Analysis (All Features)

WinRar: The adversarial goal associated with WinRAR was geared toward programs that contain some type of license verification check based on date, which is by far one of the most popular targets of hackers who want to create cracked versions of software. In this regard, both commercial obfuscators we tested (VMProtect and Themida) were ineffective at protection from adversarial comprehension (reconnaissance) in locating the section of code that checks for expiration of trial period at runtime (through disassembly and memory changes). However, both obfuscators were able to prevent cracked versions of WinRAR that permanently implemented adversarial changes. In terms of load time, both obfuscators produced variants with negligible increase in performance. In terms of overhead, Themida produced considerably larger variants than VMProtect.

Custom License Check: The program property associated with our custom program was geared toward programs that contain a license key of some kind, similar to WinRAR, but with the adversarial goal being the ability to change control flow or negate the license check in some way. In this regard, all three commercial obfuscators we tested (VMProtect, Themida, and Code Virtualizer) were ineffective at protection from adversarial comprehension (reconnaissance) in locating the section of code that uses license key for validation at runtime (through disassembly and memory changes). However, all three obfuscators were able to prevent cracked versions that permanently implemented adversarial changes. In terms of load time, only Code Virtualizer had a noticeable difference in performance degradation. In terms of overhead, Code Virtualizer produced the largest variants in terms of size.

4.3 Summary of Results

Runtime modifications to protected versions of 3-D Pinball and Tetris were successful when using both commercial obfuscators that we evaluated. In this case, the obfuscators would not prevent a MATE attack when dynamic analysis such as debuggers and runtime disassemblers are used. Modifications to the protected versions of WinRAR and custom program at runtime required more time than the unprotected versions using the same set of skills. Of particular interest, the case study results demonstrate that all three commercial obfuscators were effective in stopping would-be hackers from creating permanently modified, cracked versions of the original program. Our results also demonstrate that no commercial obfuscator we studied could fully protect a program from ad-hoc adversarial comprehension or changes while the program is running.

We can rate the performance of VMProtect in our case study as follows: 1) *Effectiveness*: protection against run time modification was ineffective; 2) *Load time*: the difference in the loading time of protected and unprotected versions is not significant; 3) *Overhead*: the size of the protected version is high.

We can rate the effectiveness of Themida in our study as follows: 1) *Effectiveness*: protection against run time modification was ineffective; 2) *Load time*: there was minimal difference in the loading time; 3) *Overhead*: the size of the programs was higher than the protected programs from VMProtect even though Themida provided a compression feature.

We can rate the effectiveness of Code Virtualizer in our study as follows: 1) *Effectiveness*: protection against run time modification was partially effective as it required more time and a larger skill set; 2) *Load time*: there was not much difference in loading time, but the program takes approximately 2-3 minutes to validate the license key in the protected version, whereas it only requires 3-5 seconds in the unprotected version; 3) *Overhead*: the size of the protected version is much higher than the protected versions of VMProtect and Themida even though Code Virtualizer provided compression.

In terms of our original premises regarding adversarial tasks, we formulated six null hypotheses in Table 3 that framed how we would assess the performance of the commercial obfuscators. We formulated one-tailed null hypotheses H01-H03 and H04-H06 to assess common techniques which all of the obfuscators share versus the best possible obfuscation that each can produce. We formulated two-tailed null hypotheses H07-H09 to infer the quality difference between common and all techniques in the obfuscator.

Table 6 summarizes the results of our analysis from the case study and the appropriate outcome, whether true or false, of each null hypothesis. In Table 6, VMP refers to VMProtect, TH refers to Themida, and CV refers to Code Virtualizer. The adversary wins if the null hypothesis is proven true and the obfuscator wins if the null hypothesis is proven wrong. In this case, we indicated that the null hypothesis was true (adversary wins) because standard/all techniques of commercial obfuscators did not decrease the capability of the attacker (our single *Knowledgeable Engineer*) to perform comprehension or change tasks at runtime. The null hypothesis is false (obfuscator wins) for standard/all techniques of commercial obfuscators because they decreased the capability of the attacker (our single *Knowledgeable Engineer*) to perform comprehension and change tasks for permanent program modifications (cracking).

Table 6. Evaluation of Null Hypotheses

On Quality of Standard Obfuscator Techniques				
	Obfuscator:	VMP	TH	CV
H_{01}	Comprehension task	True	True	False
H_{02}	Runtime change	True	True	False
H_{03}	Permanent change	False	False	False
On Quality of All Obfuscator Techniques				
	Obfuscator:	VMP	TH	CV
H_{04}	Comprehension task	False	False	False
H_{05}	Runtime change	True	True	False
H_{06}	Permanent change	False	False	False
On Comparison of Standard vs All Obfuscator Techniques				
	Obfuscator:	VMP	TH	CV
H_{07}	No difference between standard vs all for comprehension	False	False	False
H_{08}	No difference between standard vs all for runtime change	True	True	False
H_{09}	No difference between standard vs all for permanent change	False	False	False

5. CONCLUSION AND FUTURE WORK

As contributions, this paper provides one of the few published attempts to evaluate commercial obfuscators. Our approach is unique in that we propose a rating for the level of the reverse engineer in light of null hypotheses that are defined along either comprehension or change tasks. We fix experimental variables by performing only the same tasks that are allowed by the powers of MATE-based tools. However, our approach is not the only potential way to evaluate the effectiveness or potency of obfuscation, future work will be geared to adjusting the approach to more formally accepted methods of evaluation that may be adopted by the research community.

Though it is hard to gauge the entire space of end-users that might potentially perform MATE attacks against protected software programs, we would estimate that protecting programs from being "cracked" by a *Knowledgeable Engineer* would cover 80-85% of potential customers. In future work we would seek to assess how reverse engineer categories might better be used as a risk assessment tool for software companies that are considering whether or not to use commercial or open-source obfuscators to protect their code. Based on our study, we would assess that a software company would be at high risk of intellectual property theft on a mass scale (mass redistribution of a cracked executable) if they choose not to use any protection. Likewise, the use of protection software might eliminate the risk of 80-85% of customers who might be considered a *Knowledgeable Engineer* or *Novice*. Our results are in many ways equivalent to practical perspectives on virus detection software: virus protection is not perfect, but it is high risk to not use virus protection software of some kind.

We can identify several potential areas of follow-on work related to this study. First, our study assumes a certain level of skill for the adversary that is consistent with an educated end-user with above-average knowledge of computing principles (a *Knowledgeable Engineer*). The static and dynamic analysis tasks used to frame the MATE attacks were fixed in our experimental context. In the real world, attackers with greater skill levels in reverse engineering and more in-depth knowledge of program analysis techniques would try other forms of analyses and potentially advance beyond the level of analysis provided by tools such as Cheat Engine [17] or IDA Pro [16]. Future research would provide a way to incorporate such context into the results of the case study and provide a point of comparison. A broader study might include a larger body of students and/or computing professionals as part of the analysis. In general, the theory assumes that given enough time and resources, an attacker can successfully de-obfuscate any obfuscated program, and future work could explore this boundary of time, resources, and skill level.

The major risk for software programs with embedded secrets is that a small number of skilled attackers will create permanently modified versions of a program and make them available to others. In this scenario, obfuscators that protect against even 99% of potential end-users would help little. Our case study does not consider advanced attackers (*Ninjas* or *Skilled Engineers*), but this group ultimately represents the greatest threat to high-value software targets. Accordingly, in future work we will focus on how to characterize the skills or knowledge that such advanced malicious reverse engineers use and how such skills translate to successful exploitation beyond what automated tools or methods provide. Adversary skill characterization and measurement of obfuscation potency represent two hard problem areas for practical obfuscation research.

The number and type of obfuscators we used for our research was fairly small. Future work could explore a wider variety of non-commercial and research obfuscators to provide a broader picture of protection possibilities. Due to time constraints, we were also not able to take advantage of all commercial obfuscators that we had access to such as RedGate's Smart Assembly [18].

One limiting factor in our case study analysis centered on the types of programs we used (binary vs. source) and the languages involved (C++ in our case). In most cases, obfuscators prefer or require source level access to the original program to provide the highest or the best levels of protection. Several obfuscators that we had access to, for example, were specific to Java or Microsoft .NET source frameworks [13, 19]. To keep our comparisons as even as possible, we focused on common ground with x86 program executables (3-D Pinball, Tetris, and WinRAR) or source code that was compiled to x86 code (as in the case of our custom program). Future research could evaluate the effectiveness of other obfuscators using different source level languages as a result.

6. ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under grants DGE-1303384 and CNS-1305369.

REFERENCES

[1] Anon, 2013. *Defending Against the Triple Threat to Intellectual Property*. Retrieved June 15, 2015, from http://www.safenet-inc.com/resource/Resources7Detail.aspx?cat=srm&sc=341

[2] Anon, 2014. *The Compliance Gap*. Retrieved June 28, 2015, from http://globalstudy.bsa.org/2013/

[3] Akhunzada, A., Sookah, M., Anuar, NB., Gani, A., Ahmed, E., Shiraz, M., Furnell, S., Hayat, A. and Khan, M.K., 2015. Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48:44-57.

[4] Pressman, R., 2005. *Software engineering,* Boston, Mass.: McGraw-Hill.

[5] Kotonya, G. and Sommerville, I., 1998. *Requirements engineering*, Chichester: J. Wiley.

[6] Garg, M. Mamta Garg. 2009. Reverse Engineering – Roadmap to Effective Software Design. *International Journal of Recent Trends in Engineering*, 1(2), 186–188, 2009.

[7] Falcarin, P., Collberg, C., Atallah, M. and Jakubowski, M., 2011. Guest Editors' Introduction: Software Protection. *IEEE Softw.*, 28(2), pp.24-27.

[8] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K., 2012. On the (im)possibility of obfuscating programs. *Journal of the ACM,* 59(2), 1-48.

[9] Vmpsoft.com, 2003. VMProtect Software Protection: VMProtect. Retrieved August 3, 2015, from http://vmpsoft.com/products/vmprotect/

[10] Oreans.com, 2004. *Oreans Technology* : Themida. Retrieved August 12, 2015, from http://www.oreans.com/themida.php

[11] Oreans.com, 2006. *Oreans Technology* : Code Virtualizer. Retrieved August 12, 2015, from http://www.oreans.com/codevirtualizer.php

[12] Red-gate.com, Redgate's SmartAssembly. Retrieved August 13, 2015, from https://www.red-gate.com/products/dotnet-development/smartassembly/

[13] Ssware.com, 2010. Crypto Obfuscator For .Net - Obfuscator With Code Protection, Exception Reporting, Optimization. Retrieved August 14, 2015, from http://www.ssware.com/cryptoobfuscator/obfuscator-net.htm

[14] Collberg, C., 2015. *The Tigress C Diversifier/Obfuscator*. Retrieved August 14, 2015, from Arizona University: Tigress.cs.arizona.edu.

[15] Ollydbg.de, 2013. OllyDbg. Retrieved September 8, 2015, from http://www.ollydbg.de/

[16] hex-rays.com, Hex-Rays Home: IDA. Retrieved September 20, 2015, from https://www.hex-rays.com/products/ida/

[17] Cheatengine.org, Cheat Engine. Retrieved September 25, 2015, from http://www.cheatengine.org/index.php

[18] Red-gate.com, .NET Decompiler: Decompile Any .NET Code | .NET Reflector. Retrieved August 23, 2015, from https://www.red-gate.com/products/dotnet-development/reflector/

[19] Jetbrains.com, Free .NET decompiler : JetBrains dotPeek. Retrieved October 10, 2015, from https://www.jetbrains.com/decompiler/

[20] Ilspy.net, ILSpy. Retrieved October 17, 2015, from http://ilspy.net/

[21] Collberg, C. and Nagra, J., 2009. *Surreptitious software*, Boston, Mass.: Addison-Wesley.

[22] Emanuelsson, P. and Nilsson, U., 2008. A Comparative Study of Industrial Static Analysis Tools. *Electronic Notes in Theoretical Computer Science*, 217, pp.5-21.

[23] Collberg, C., Thomborson, C., and Low, D. 1997. A taxonomy of obfuscating transformations. *Department of Computer Science, The University of Auckland, New Zealand*.

[24] Cazalas J., McDonald, J.T., Andel, T. R., and Stakhanova, N. Probing the Limits of Virtualized Software Protection. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop (PPREW-4).* ACM, New York, NY, USA, Article 5, 11 pages, 2014,

[25] Collberg, C. and Thomborson, C., 2002. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8), pp.735-746.

[26] Macbride, J., Mascioli, C., Marks, S., Tang, Y., Head, L.M. and Ramach, P., 2005. A comparative study of java obfuscators. In *Proceedings of the IASTED International Conference on Software Engineering and Applications (SEA 2005)*. 14–16.

[27] Ceccato, M., Di Penta, M., Falcarin, P., Ricca, F., Torchiano, M. and Tonella, P., 2013. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques*. Empirical Software Engineering.*

[28] Zhang, X., Tsang, A., Yue, W.T. and Chau, M., 2015. The classification of hackers by knowledge exchange behaviors. *Information Systems Frontiers*, 17(6), pp.1239-1251.

[29] Nagra, Jasvir and Collberg, C. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection* (Addison-Wesley Software Security Series). Pearson Education, 2010.