A Tool for Teaching Reverse Engineering

Clark Taylor[1] and Christian Collberg[1]

[1]Department of Computer Science, University of Arizona

Abstract

Tigress is a freely available source-to-source, C language code obfuscator. The tool allows users to obfuscate existing programs or programs randomly generated by Tigress itself. Tigress is highly flexible, providing a large number of standard obfuscating code transformations, and many variants of each transformation. Tigress may be used in many contexts, but in this paper we describe its use in teaching code reverse engineering techniques. In order to make Tigress easily available and usable to educators and students, we have integrated Tigress into a web application. In addition to directly benefiting education, this new web application offers unique ways to advance research on code obfuscation and reverse engineering.

1 Introduction

In computer science, and computer security in particular, students often learn skills through exercises. Instructors generate the exercises with the goal of mimicking situations found in the real world. However, generating exercises can be both time consuming and difficult; without automation, instructors cannot easily generate individualized challenges for students, but rather must assign and manually administer a single problem to the entire class. Students, on the other hand, often spend significant amounts of time setting up an environment in which they have the tools necessary to solve the problem.

Instruction in code reverse engineering suffers from a lack of easy to use tools to resolve these difficulties. Here, we confront these problems by combining Tigress [1]—an automated C language, source-to-source code obfuscator—with a web application. This application allows the instructor to generate individualized target programs for students to reverse engineer. Each program consists of automatically generated random code which has been obfuscated with a set of transformations. The complexity of the resulting *target program* can be configured by the instructor. The web application then generates virtual machines (VMs) which, in addition to the target program, have been configured with reverse engineering tools selected by the instructor. The students download the VM, deobfuscate the code with the provided tools, and upload the results back to the web application. Grading the results can be both automatic and manual.

During the process described above, the VMs may also collect information about the tools, methods, and processes the students used to solve the exercises. The resulting data sets may reveal the most effective reverse engineering practices, both in actual code deobfuscation as well as in instruction.

Our paper is organized as follows: First, we review previous work. Second, we describe our proposed system. Third, we present our current implementation. Finally, we discuss our experiences in employing this implementation.

2 Related Work

The skills taught by the system we propose include basic reverse engineering methodology and use of standard tools. As an educational tool, our work expands on other developments in teaching computer security skills, particularly drawing from competition-based systems such as picoCTF [2] and iCTF [3].

2.1 Reasons for Reverse Engineering

Cipresso [4] provides an overview of the applications and goals of software reverse engineering. He identifies two legitimate reasons for reverse engineering code: (1) to understand, patch, and maintain legacy code; and (2) to determine the function of an unknown piece of software for security purposes [4]. Other, less legitimate reasons for reverse engineering include gaining access to closed-source code which may be protected by legal or ethical standards such as intellectual property law or national security policies. The legitimate uses inspire the goals of the project here: we wish to educate and train future security professionals in the art of software reverse engineering. In particular, we focus on (2): we want to provide training in how to reverse engineer purposefully obfuscated code. Such training will provide the students with the necessary skills to reverse engineer malware.

2.2 Computer Security Competition

Computer security competitions have become very popular [2, 3]. They take various forms and have different motives: while some seek to train, others emphasize the competitive and entertainment aspects of breaking and entering. Competitions often include challenges which distribute obfuscated code for varying forms of analysis. Typically, these codes have been designed and obfuscated by hand by those managing the competition. Competitors download and deobfuscate the code and extract from it some meaning or token [2]. Some competitions also require peer-to-peer code development and reverse engineering. In such cases, competitors must reverse engineer other competitors' code in order to advance towards a goal such as gaining access to a system [3].

In addition to general computer security competitions, there exist several competitions whose sole purpose is to create [5] or reverse engineer [6] obfuscated code. Often, these competitions function as boundary-pushers, testing the latest tools and methods of code obfuscation and reverse engineering. Some competitions offer polymorphic challenges to add some randomization [7].

2.3 Reverse Engineering Tools

Several code reverse engineering tools have been demonstrated to be effective [8]. Examples include IDA [9], GDB [10], OllyDbg [11], Valgrind [12], and the angr framework [13]. IDA is a debugger with an extensive graphical user interface that visualizes the control flow of binaries. GDB—the familiar command line debugger—has several functions which may be used for reverse engineering, including disassembly, debugging, and direct modification of executable images. OllyDbg is another debugger which contains several features that track the machine state and software interaction. Valgrind is a virtualizing debugger framework which includes prebuilt tools for code execution tracing. The angr tool is a new binary analysis framework with several components that allow users to programmatically disassemble, simulate the execution of, and trace data in binaries. In the system we propose here we make these, and other, tools available to the students.

2.4 Automated Code Obfuscation

Automated code obfuscation comes in several varieties. First, some pieces of software integrate obfuscation into their own code. This is typical of viruses, which self-obfuscate in order to avoid detection [14]. Second, there exist a wide variety of stand-alone tools available to obfuscate code [15, 16]. An obfuscating transformation changes the form of a piece of code, while maintaining its semantics, in order to impede analysis by human reverse engineers or by automatic deobfuscation tools [17].

Of these tools, Tigress [1] is a freely available tool that offers a large collection of transformations. It operates on the C language at the source code level. Tigress has built-in features which allow randomized code generation as well as randomized code transformations. In this project we use Tigress to create new reverse engineering challenges, first by generating random code and then by obfuscating this code.

3 Proposed System

In order to teach code reverse engineering skills, in this paper we propose a system which automatically generates and administers reverse engineering exercises for students to complete. This system contains several features, outlined below, which we implemented in part.

3.1 Administrative Functions

Previously, reverse engineering exercises were generated by hand or from scripts. These challenges had to be individually handled in an ad-hoc fashion over general tools like email. This legacy process creates a large amount of manual overhead in administering exercises, as the instructor must create individual challenges for individual students, distribute those challenges individually, accept and aggregate the answers individually, and grade them—individually. Scripting can help in some of these aspects, but an automated system promises to simplify the process further. Thus, the goal of our system is to require only a small amount of instructor input to create and administer a challenge set.

3.2 Randomization

Reverse engineering competitions do not typically generate individualized targets. In fact, we could not find an example of a system or script that generates randomized reverse engineering problems beyond limited application of simple polymorphic algorithms to otherwise identical code. In a competitive setting such randomization may not be necessary. Pedagogical settings, by contrast, require randomization, as it allows instructors to effectively eliminate problems related to students sharing work or finding previous solutions.

3.3 Tools and Challenge Distribution

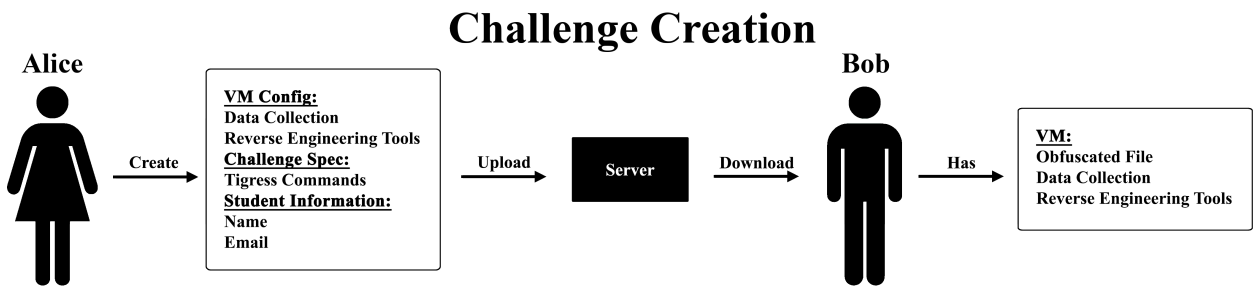
Code reverse engineering employs a variety of methodologies and tools. Teaching effective reverse engineering skills necessarily requires instruction in the use of these tools. It is important to make the tools available to the students in an effective and efficient manner, allowing them to quickly begin to solve problems. Our system includes dynamically configured VMs which provide a pre-built environment to students, with reverse engineering tools already installed. Additionally, these VMs include the actual challenge code, eliminating the step of downloading the obfuscated code manually. Students must only download a single VM file from our system, load it into a VM player, and begin to solve problems.

3.4 Data Collection

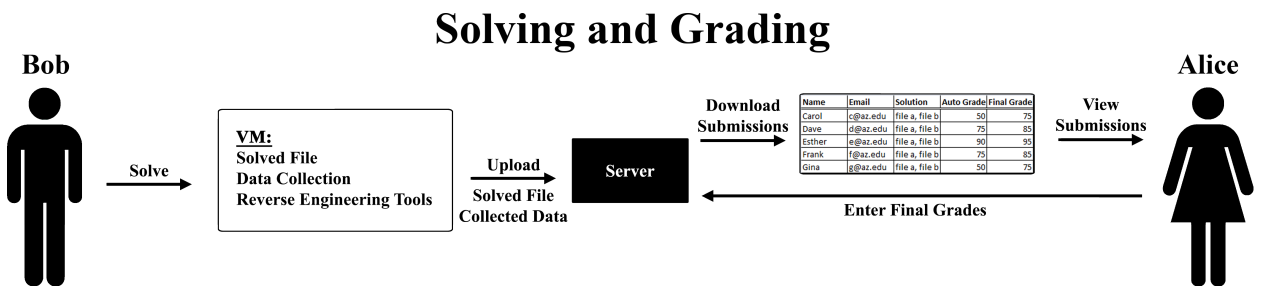
One goal of our system is to create data sets which may be used to evaluate the methods, techniques, and tools used in reverse engineering. Requiring students to use pre-configured VMs allows us to add data collection software. This software may collect various pieces of information from students while they solve challenges: running processes, screenshots, network traffic, system kernel modules, and even high-resolution data from reverse engineering tools. Our goal is to use this data to evaluate pedagogical methodologies and instruction as well as monitor progress. Additionally, these data sets could be used to determine the most effective modes of reverse engineering code, which in turn aids analysis of the effectiveness of code obfuscation itself [18].

3.5 Application Functionality

We will next consider the three main steps in how our proposed system is used: *create* a challenge, *solve* a challenge, and *grade* a challenge. As shown in Figure 1, each of these steps has several parts. Typically, a challenge is created by an instructor by combining a *VM configuration* and a *target program configuration.* The latter is a list of command line arguments for the Tigress obfuscator to create a random program with certain characteristics and then to obfuscate this pro-



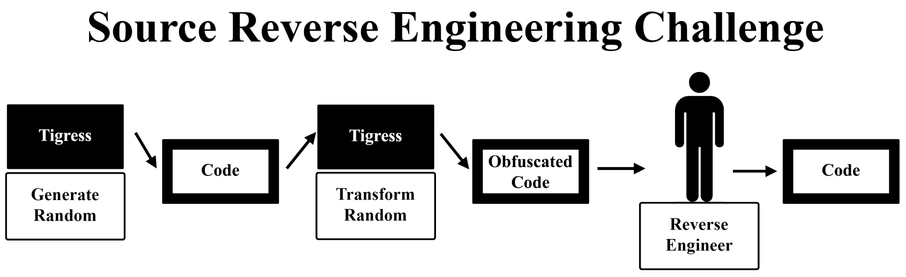
(a) Alice uploads a Challenge package, which Bob uses to generate his Challenge.



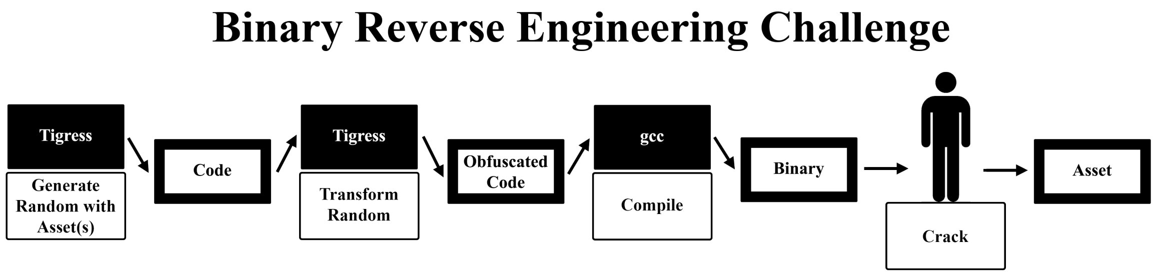
Name	Email	Solution	Auto Grade	Final Grade
Carol	c@az.edu	file a, file b	50	75
Dave	d@az.edu	file a, file b	75	85
Esther	e@az.edu	file a, file b	90	95
Frank	f@az.edu	file a, file b	75	85
Gina	g@az.edu	file a, file b	50	75

(b) Bob solves and uploads his Challenge, which Alice then grades.

Figure 1: This shows the basic use cases. Alice is an instructor and Bob is a student.



(a) This use generates obfuscated code which students may reverse engineer into cleartext (un-obfuscated) code.



(b) This use generates a binary to crack; variations include disabling parts of code or extracting a password.

Figure 2: This displays the two current types of Challenges generated thus far.

gram with a particular set of transformations. Once created, a student downloads the challenge, solves the task, and submits the answer. This creates a *challenge submission*, which contains the solution. The instructor then invokes automatic grading or enters grades manually.

4 Implementation

We implemented the system described above in part; some features are not yet complete.

4.1 Architecture

Our implementation utilizes standard web components: a web server connected to a database. Typical administrative data—including authentication information and data dictating instructor-challenge-student relationships—is stored in the database, as is challenge data such as obfuscation configurations. The web server interacts with the native operating system and file system in order to call Tigress, giving it flags and files to obfuscate. The web application stores the non-obfuscated and final files in a database for download by students and subsequent grading.

4.2 Challenge Creation

In configuring a challenge, instructors may upload a base file with which to start obfuscation. Alternatively, Tigress may also generate a random file upon which to perform obfuscation, ensuring that students receive unique problems; it does this by accepting certain arguments (specified by the instructor) with which it creates random C code with random variables and functions structured in random ways but which always include particular features to reverse engineer [1]. Once the target program is defined, our system uses Tigress to execute selected obfuscating transforms on the target program. These steps introduce further randomness by arbitrarily selecting transform-dependent variables such as function ordering. Figure 2 illustrates how Tigress creates two types of problems: source code reverse engineering and binary cracking.

4.3 Virtual Machines

Currently, our implementation only provides a statically configured VM for students to download. The VM provided is a Kali [19] distribution with the addition of IDA (demo version) and angr. This falls short of the all-in-one solution presented above. However, dynamically generating unique VMs has thus far proven to be too slow and the resulting files too large. To resolve these difficulties in future work, we are considering using dockers and provisioners.

4.4 Grading

The current implementation only allows manual challenge grading. Instructors may review submitted and base files to determine whether the student solved the problem. Grades are then be entered into the system, stored in the database, and then made available for students to review.

5 Use and Results

We used our current system to create and administer two challenges for a computer security course. Despite a few small and typical bugs, students were able to download challenge code, solve those challenges, and upload answers. Two challenges were offered, the second more difficult than the first; students were required to answer one of the two problems.

The easier problem consists of a program that checks the current time before printing a variable. If the time check is not adequately met, then the program produces a segmentation fault. Students were to alter the binary and eliminate the time check and thus unlock an output calculated from a myriad of operations. The time check and variable calculation function is shown in Figure 3. The second problem is similar to the first but adds an additional aspect: in addition to the time check students must also eliminate a password check.

In the submission file, students are required to state the level of difficulty they encountered and the amount of time they spent solving the problem. We only analyzed files submitted for the first, easier problem, as only two students submitted answers for the

```
void SECRET(unsigned long input[1] , unsigned long output[1] ) {
unsigned long state[1] ; //Variable declaration
unsigned long (*output_ref)[1] = output;
unsigned int copy15,copy16,copy12 ;
unsigned short copy17 ; {
 state[0UL] = (input[0UL] << 3UL) | (input[0UL] >> 61UL); //Initial expansion of the input
 copy12 = *((unsigned int *)(& state[0UL]) + 1);
 *((unsigned int *)(& state[0UL]) + 1) = *((unsigned int *)(& state[0UL]) + 0);
 *((unsigned int *)(& state[0UL]) + 0) = copy12;
 struct timeval __cil_tmp13;
 int __cil_tmp14 = gettimeofday(& __cil_tmp13, 0);// Get the time
 long time = __cil_tmp13.tv_sec;
 if ((state[0UL] >> 4UL) & 1UL) { //Second state, control structures to compute the output
   state[0UL] |= (state[0UL] & 63UL) << 4UL;
   copy15 = *((unsigned int *)(& state[0UL]) + 0);
   *((unsigned int *)(& state[0UL]) + 0) = *((unsigned int *)(& state[0UL]) + 1); }
 int failed |= time > 1398629497UL;//This is the time check
 copy17 = *((unsigned short *)(& state[0UL]) + 1); //Expansion phase to compute output
 *((unsigned short *)(& state[0UL]) + 1) = *((unsigned short *)(& state[0UL]) + 2);
 *((unsigned short *)(& state[0UL]) + 2) = copy17
 if (failed) {
   output_ref = 0UL; // Set pointer to NULL to force crash}
 (*output_ref)[0UL] = state[0UL] >> 1UL; }
```

Figure 3: Example generated code.

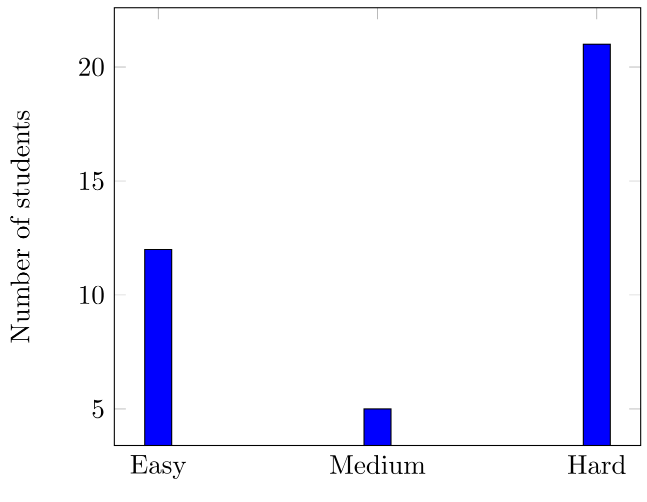


Figure 4: This graph displays students' self reported difficulty in solving Challenge 1.

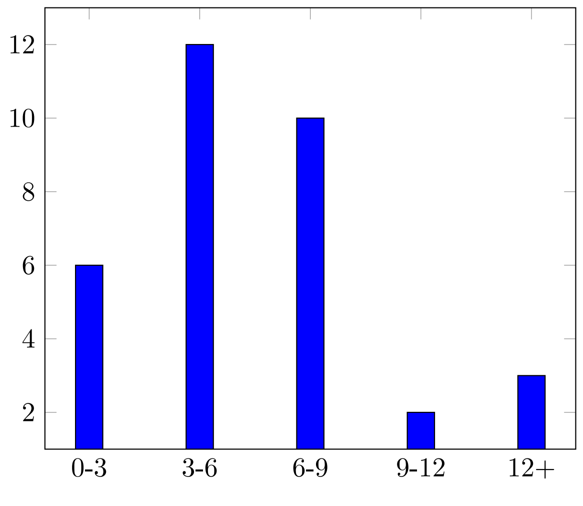


Figure 5: This graph displays students' self reported time spent solving Challenge 1, in hours.

second, harder problem. Figure 4 displays a summary of students' reported level of difficulty; most found the problem either easy or hard. This likely corresponds to students' prior experience. Some of the difficulty students encountered derived from minor issues with the new system implementation and process; examples of such issues include difficulty of downloading the VM as well as general problems with VM players. We see that students spent an average of about 5.5 hours solving the problem; the distribution of student time spent solving problems is shown in Figure 5[1]. Most students were able to complete the assignment, which indicates that our system provided an effective means of generating and administering reverse engineering challenges. Additionally, students' general ability to complete the assigned challenge in a reasonable amount of time indicates that the assignment was likely successful in teaching reverse engineering skills to the students here.

6 Future Work

Our current focus is to improve the current system implementation to bring it closer to the proposed system. The current system lacks dynamic VM creation; the problems we encountered when implementing this must be resolved in the future. We will furthermore incorporate data collection facilities in order to generate usage data for analysis. Finally, we will add facilities to support semi-automatic grading. The latter poses significant problems. Some generated challenges require finding some type of hidden token and may be easily graded. Determining whether a submission has successfully reverse engineered a more general obfuscated target program, however, is less straightforward. In such cases, there exist two criteria a grader must consider. First, the grader must determine identical functionality between the target program and the supposedly deobfuscated submission. This may be accomplished by comparing input and output of the target and submitted programs. Second, the grader must be able to determine whether the submitted program has successfully deobfuscated the target program—that is, whether the submitted program is the equivalent of the non-obfuscated version of the target program. Control flow graphs comparisons may aid in determining that equivalence [20].

[1]The data presented here has been ruled IRB exempt by the University of Arizona.

In addition to these concrete improvements on the current implementation, future work encompasses work on creating novel challenge generation scripts as well as additional work on Tigress. As more challenges are developed by instructors, they may be easily shared with instructors everywhere. Due to randomization, challenge reuse does not pose a problem; students will not be able find or share answers to randomized exercises.

7 Conclusion

Reverse engineering code is a vital skill in several fields within Computer Science. Teaching reverse engineering and, in particular, contemporary methods and tools used in reverse engineering is not an easy task. Without automation, instructors have to manually obfuscate uniform code they themselves develop. This paper proposes an application which automates the process and describes our initial implementation of that system. Using the Tigress C source code obfuscator, our application allows instructors to automatically create randomized obfuscated code for individual students; instances of challenges that students download share only general objectives but not common code. Providing a virtual environment preconfigured with common reverse engineering tools further simplifies the learning process.

Initial results demonstrate the efficacy of the current implementation of the system. Further development of this system holds additional promise by enabling the generation of data sets useful for research in reverse engineering.

Acknowledgments

We thank David Christy for creating, administering, and grading the challenges. This project was funded in part by NSF grant CNS-1145913.

References

[1] C. Collberg. The Tigress C Diversifier/Obfuscator. [Online]. Available: http://tigress.cs.arizona.edu/index.html

[2] P. Chapman, J. Burket, and D. Brumley, "PicoCTF: A Game-Based Computer Security Competition for High School Students," in *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*. USENIX Association, 2014.

[3] G. Vigna, K. Borgolte, J. Corbetta, A. Doupé, Y. Fratantonio, L. Invernizzi, D. Kirat, and Y. Shoshitaishvili, "Ten Years of iCTF: The Good, The Bad, and The Ugly," in *2014 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 14)*. USENIX Association, Aug. 2014.

[4] T. Cipresso, "Software Reverse Engineering Education," 2009.

[5] L. Broukhis, S. Cooper, and L. C. Noll. The International Obfuscated C Code Contest. [Online]. Available: http://www.ioccc.org/

[6] LayerOne 2016 - (De)Obfuscation Contest. [Online]. Available: https://obf.afm.la/

[7] W. chang Feng, "A Scaffolded, Metamorphic CTF for Reverse Engineering," in *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*. Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: http://blogs.usenix.org/conference/3gse15/summit-program/presentation/feng

[8] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse Engineering Obfuscated Code," in *12th Working Conference on Reverse Engineering (WCRE'05)*, 11 2005, pp. 10 pp.–.

[9] IDA: About. [Online]. Available: https://www.hex-rays.com/products/ida/

[10] GDB: The GNU Project Debugger. [Online]. Available: https://www.gnu.org/software/gdb/

[11] OllyDbg 2.01. [Online]. Available: http://www.ollydbg.de/version2.html

[12] Valgrind. [Online]. Available: http://valgrind.org/

[13] angr. [Online]. Available: http://angr.io/

[14] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Springer Journal in Computer Virology*, vol. 3, no. 3, pp. 211–220, 8 2008.

[15] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – Software Protection for the Masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed., 2015, pp. 3–9.

[16] B. Bertholon, S. Varrette, and P. Bouvry, *JShadObf: A JavaScript Obfuscator Based on Multi-Objective Optimization Algorithms*. Springer Berlin Heidelberg, 2013, pp. 336–349.

[17] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Addison-Wesley Professional, 2009.

[18] S. Banescu, M. Ochoa, and A. Pretschner, "A Framework for Measuring Software Obfuscation Resilience against Automated Attacks," in *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, 2015, pp. 45–51.

[19] T. Heriyanto, L. Allen, and S. Ali, *Kali Linux: Assuring Security By Penetration Testing*. Packt Publishing, 2014.

[20] P. P. Chan and C. Collberg, "A Method to Evaluate CFG Comparison Algorithms." [Online]. Available: http://cfgsim.cs.arizona.edu/qsic14-slides.pdf