Software Obfuscation with Layered Security

XU, Hui

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy
in
Computer Science and Engineering

The Chinese University of Hong Kong
September 2018

Thesis Assessment Committee

Abstract of thesis entitled:
Software Obfuscation with Layered Security
Submitted by XU, Hui
for the degree of Doctor of Philosophy
at The Chinese University of Hong Kong in September 2018

Software obfuscation is an essential technique that can protect software intellectual properties from man-at-the-end attacks. It transforms computer programs to new versions which are semantically equivalent to the original ones but much harder to understand. Nevertheless, practical obfuscation techniques are not as secure as other security primitives, such as cryptography. A notable challenge is that software is much complicated with composite components. It is therefore hard to develop a single obfuscation approach which is secure-against-all. To tackle the problem, we investigate the idea of *layered obfuscation* in this thesis, whereas an obfuscation solution should integrate several different obfuscation techniques to provide layered security, and each obfuscation technique only corresponds to particular threats.

This thesis contains two parts. In the first part, we introduce the concept of layered obfuscation and systematize the knowledge of present obfuscation techniques. We develop a novel taxonomy of existing obfuscation techniques, aiming to facilitate the design of layered obfuscation solutions and promote the adoption of the idea in practice. With our taxonomy hierarchy, the obfuscation techniques under different branches are orthogonal to each other, and thus developers can easily choose and combine them based on their specific requirements.

In the second part, we enrich the obfuscation taxonomy with several new obfuscation techniques and provide developers with more options in designing competent obfuscation solutions. Our first technique relates to the deobfuscation attacks based on symbolic execution. It improves the resilience of present control-flow obfuscation techniques to such adversaries by deliberately introducing challenging program analysis problems during obfuscation. Our second obfuscation technique focuses on app tampering attacks. We address such issues by introducing diversities among the obfuscated apps of different users and therefore enable the apps with resilience to large-scale attacks. Our third technique is related to deep learning software. To protect well-trained machine learning models from piracy, we propose to obfuscate the structure of private neural networks via a simulation-based method.

In summary, this thesis proposes an overall theme that a critical path towards achieving reliable obfuscation is layered security, and we make several contributions on promoting the idea. The thesis is a rethinking of the dilemma faced by software obfuscation, and we hope it can inspire more discussions and facilitate the development of the area.

論文題目：軟件混淆的分層安全方法

作者　　：徐輝

學校　　：香港中文大學

學系　　：計算機科學與工程學系

修讀學位：哲學博士

摘要　　：

軟件混淆技術是保護軟件知識產權的重要方法，可對攻擊者在用戶端的發起的逆向攻擊起到壹定的防禦作用。它的工作原理是將目標代碼變換為另外壹種難以理解的版本，並且保證語意層面等價。但是，實際可用的軟件混淆技術並不能像其它安全技術（如密碼）壹樣有嚴格的安全保證。本論文研究分層混淆的方法。我們的方法認為每種混淆技術應該面向特定的安全風險，而不是解決所有問題。壹個軟件混淆方案應當集成不同層面的代碼混淆技術來增強混淆軟件的安全性。

本論文包括兩個主要部分。第壹部分介紹分層混淆的理念並且將現有的混淆技術和知識進行歸類。為了輔助開發者設計分層混淆方案和選取合適的混淆技術，我們建立了壹套完整的代碼混淆技術生態系統。按照我們的生態系統架構，不同分支下的混淆技術均為正交的。開發者可以根據需求選取相應分支下的混淆技術並進行有效組合以達到好的分層混

淆效果。

論文第二部分介紹了幾種新的混淆方法，從而豐富了現有的混淆技術生態系統，為開發者設計分層混淆方案提供了更多的技術選擇。第壹個攻擊模型與基於符號執行的攻擊者有關。我們通過在混淆過程中引入程序分析難題來增強現有混淆方法的安全性。第二個攻擊模型針對應用程序篡改。我們的方法是在混淆過程中引入代碼多樣性來防止大規模攻擊的發生。第三個模型面向深度學習軟件的剽竊。我們提出了壹種通過模型仿真來混淆私有神經網絡的結構的技術。

綜上所述，本論文提出實現可靠軟件混淆的壹個重要途徑是分層安全，並且對促進該理念的發展做出了壹些貢獻。本論文重新思考了軟件混淆技術發展所面臨困境。我們希望它可以啟發更多討論並最終促進該領域的發展。

Acknowledgement

I feel highly privileged to express my gratitude to a number of people who helped me in finishing this thesis.

First and foremost, I would like to thank my supervisor, Prof. Michael R. Lyu, who made it possible for me to pursue my Ph.D. study at CUHK and led me into the world of scientific research. Prof. Lyu gave me the most freedom in choosing research topics and exploring research ideas. He provided me with adequate resources for research and guided me with his experiences and broad horizons. He has been very kind and very generous in encouraging me when I achieved small progress. I feel very fortunate to join his research lab, and I think I have learned a lot on this wonderful journey. I am sure that these lessons will be beneficial for my whole life.

I am grateful to my thesis assessment committee members, Prof. Qiang Xu and Prof. Pak Ching Lee, for their constructive comments and valuable suggestions to this thesis and all my term reports. Great thanks to Prof. Jiannong Cao from Hong Kong Polytechnic University who kindly served as the external examiner for this thesis.

I thank my coauthors who contribute a lot to my research projects, including Dr. Yangfan Zhou, Dr. Yu Kang, Yuxin Su, Cuiyun Gao, Zirui Zhao, and Fengzhi Tu. I enjoy the collaboration with them. When I first joined this lab, Dr. Yangfan Zhou and Dr. Yu Kang often worked closely with me until midnight. They helped me a lot in finding valuable research topics, developing systematic research ideas, and writing logical papers. I thank their valuable guidance and contribution to the research work in this thesis.

I would also like to thank my friends in Room 101, Jieming Zhu, Pinjia He, Junjie Hu, Jichuan Zeng, Jian Li, Shilin He, Pengpeng Liu, Yue Wang, Weibin Wu, and Zhuangbin Chen, whose spirits of pursuing top research encouraged me.

Last but not least, I would like to thank my parents and my wife. Without their deep love and constant support, this thesis would never have been completed.

To my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents our work towards obfuscating real-world software with layered security. In this chapter, we discuss the motivations of our research and summarize the contributions we make.

1.1 Rationale

Sofware obfuscation transforms computer programs to new versions which are semantically equivalent with the original ones but much harder to understand [40]. It is a technique which protects software intellectual properties against MATE (Man-At-The-End) attacks [39]. The concept was originally introduced at the International Obfuscated C Code Contest in 1984, which awarded creative C source codes with "smelly styles". Later in 1997, Collberg *et al.* [40] published a milestone paper discussing the taxonomy of obfuscation transformations for Java programs. Since then, the technique has become indispensable for software protection. There are many practical obfuscation approaches developed, such as lexical obfuscation with ProGuard[1] and control-flow obfuscation with Obfuscator-LLVM [93].

[1]https://www.guardsquare.com/en/products/proguard

1.1.1 Critical Challenge of Obfuscation

Although obfuscation has been developed for over 30 years, the questions yet unsolved are how much developers can trust the technique and how to design reliable obfuscation solutions. Such issues are very critical because obfuscation is a security primitive. To tackle these questions, we have surveyed the literature of both theoretical and practical obfuscation research.

From the theoretical perspective, many discussions (*e.g.,* [12, 71, 109, 201] on this problem have arisen in recent years. The representative ones include the negative result showed by Barak *et al.* [12] that we cannot obfuscate all program with black-box security, and the positive result presented by Garg *et al.* [71] that graded encoding is a promising obfuscation algorithm for achieving a weaker security notion: indistinguishability. However, we cannot apply these results to practical software obfuscation directly because there are obvious gaps in between. Note that such theoretical research focuses on obfuscating computation models (*e.g.,* circuits or Turing Machines) instead of real codes. While computation models are mathematical and their properties are usually provable, real codes are more complicated and their properties are hard to prove. In practice, we generally program software with high-level programming languages which cannot be reduced to pure mathematical representations easily.

From the practical area, we attempt to find some clues for designing reliable obfuscation solutions. We find that present obfuscation research generally assumes a specific code format (*e.g.,* Java bytecodes or assembly codes) for obfuscation. However, real-world software can be more complicated than that. For instance, an Android app (Figure 1.1) contains several different components, such as Java codes, native codes, third-party libraries, and other resources. Securely obfuscating the whole app with only one approach is nearly impossible. Moreover, merely applying some obfuscation

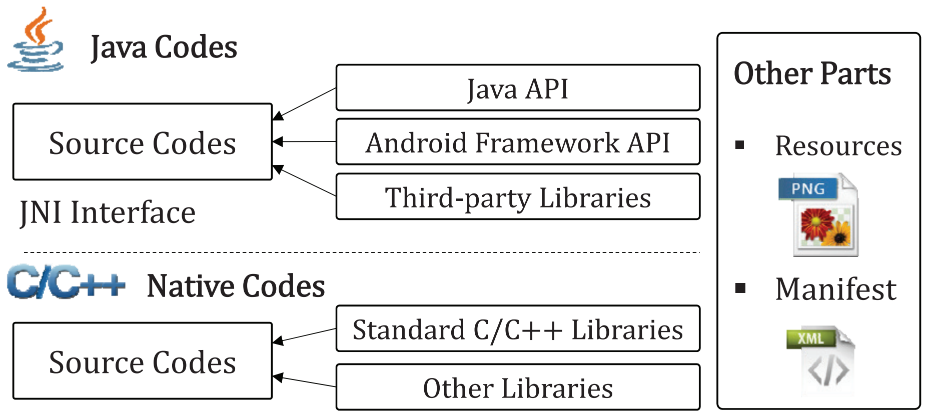


Figure 1.1: The components of Android apps.

techniques in an ad-hoc way can achieve very limited obscurity because it lacks a holistic design. In particular, the remaining unobfuscated information could jeopardize the obfuscated software. For instance, the lexical obfuscation approach provided by ProGuard transforms identifiers of Android apps to meaningless alphabets or strings, which seems one-way secure. But a recent attack [18] shows that attackers can recover a significant portion of the original lexical information leveraging the residual information within the obfuscated apps.

We conjecture that achieving reliable obfuscation is challenging mainly due to the complicated nature of software, and we believe a promising way to handle the challenge is applying the classic idea of *layered security* to software obfuscation. Next, we elaborate on the idea of obfuscation with layered security.

1.1.2 Layered Security for Obfuscation

Layered security is an effective risk management strategy. It mitigates the risks that a threat becomes a reality with several protections from different layers or of various types. The idea has become prevalent for securing information systems after it has been introduced by the Department of Defense in IATF (Information

Assurance Technical Framework[2]). Because information systems are very complicated, there is no silver bullet for avoiding all risks, and layered security is the best practice. In the first level, IATF divides information systems into four areas or layers, which are local computing environment, enclave boundaries, network and infrastructures, and supporting infrastructures. Each of these layers faces a specific group of threats and should be protected correspondingly. Take the area of network and infrastructure as an example, administrators can employ firewalls to deter denial-of-service attacks from the internet, and they can use the SSL/TLS (Secure Socket Layer/Transport Layer Security) gateways [143] to encrypt the traffics from being eavesdropped. The layered security idea integrates different security mechanisms as a whole to protect the security of a system.

Although the software is not as complicated as information systems, its complexity is beyond the capability of any single obfuscation technique. Therefore, we believe employing the idea of layered security for software obfuscation should be a promising way, namely *layered obfuscation*. Different from mainstream obfuscation research which treats software as simple codes, we think practical obfuscation should be based on risk management and should integrate several obfuscation techniques to mitigate different risks.

In practice, layered security has already been employed in protecting real-world digital assets and systems, such as digital watermarking [13] and cloud [190]. Yet, the idea is still very preliminary for software obfuscation. Although some practical obfuscation tools (*e.g.,* DexGuard[3] and DexProtector[4]) already support multiple obfuscation techniques, they do not provide a systematic way regarding how to integrate them concerning layered security.

[2]http://www.dtic.mil/docs/citations/ADA606355

[3]https://www.guardsquare.com/dexguard

[4]https://dexprotector.com/

1.2 Summary of Contributions

This thesis aims to help developers to design obfuscation solutions with layered security. We make contributions on two folds. Firstly, we develop a novel taxonomy of obfuscation techniques which can assist developers in choosing and integrating obfuscation techniques. Secondly, we enrich the taxonomy with three new obfuscation techniques which provide developers with more options for designing better obfuscation solutions. We discuss more details as follows.

1. **Obfuscation taxonomy for layered security**

   When designing layered obfuscation solutions, developers should know present obfuscation techniques as well. Such knowledge is essential for them to choose appropriate techniques and to integrate them efficiently. To meet this need, we develop a taxonomy of obfuscation techniques concerning layered security and systematically analyze the feature of each technique. In the first level of the taxonomy, we categorize obfuscation techniques into four layers based on the obfuscation targets, which are the code-element layer, software-component layer, cross-component layer, and application layer. In the second level, each layer forks into several sub-categories if the obfuscation targets can be further classified. For example, the code-element layer contains data and controls, which are two sub-categories and require different obfuscation techniques. The leaf nodes of the taxonomy hierarchy are specific obfuscation approaches for protecting corresponding targets.

2. **Symbolic opaque predicates for control-flow obfuscation**

   Opaque predicates are essential gadgets for control-flow obfuscation. However, real-world opaque predicates are vulnerable to symbolic execution-based adversaries. To address this problem, we first conduct a systematic study on the challenges faced by symbolic execution and propose a method to benchmark symbolic execution tools in handling these challenges. With the benchmarking method, we have confirmed the prevalence of the challenges with several popular symbolic execution engines (*e.g.,* KLEE [26], Triton [148], and angr [158]). Next, we propose a framework to compose opaque predicates leveraging these challenges. In this way, the opaque predicates can be resilient to symbolic execution-based attacks. A novel characteristic of such opaque predicates is the *bi-opaque property*, which incurs not only false negative issues but also false positive issues to attackers. We have implemented a prototype obfuscation tool based on Obfuscator-LLVM and verified the effectiveness of our approach.

3. **N-version obfuscation for tampering-resilience**

   Nowadays, software tampering attack remains a critical security threat to software systems. None of the existing mechanisms can achieve theoretically tampering-proof without the assistance of trusted hardware. Instead of proposing new tricks against software tampering attacks, N-version obfuscation (NVO) focuses on impeding the replication of tampered software via program diversification. In this way, it can pose a barrier to attackers when they are launching large-scale attacks. This thesis presents a systematic design of a candidate NVO solution for networked apps, which leverages a message authentication code (MAC) mechanism to generate diversities and enforce security features. Our evaluation results show that the time required for breaking such a software system increases linearly to the number of software versions. In this way, attackers would suffer great scalability issues, considering that an app can have millions of users, each using a different version. With minimal NVO costs, effective tampering-resistant security can therefore be established.

4. **Obfuscating deep neural networks**

   We investigate the piracy issue faced by deep learning software. Designing and training a well-performing model is generally expensive. However, when releasing them, attackers may reverse engineer the models and pirate their design. We propose *deep learning obfuscation*, aiming at obstructing attackers from pirating a deep learning model. In particular, we present our work on obfuscating convolutional neural networks (CNN). Our approach obfuscates a CNN model by simulating its feature extractor with a shallow and sequential convolutional block. We have verified the feasibility of our approach with three prevalent CNNs, *i.e.,* GoogLeNet, ResNet, and DenseNet. Although these networks are very deep with tens or hundreds of layers, we can simulate them with a shallow network containing only five or seven convolutional layers. The obfuscated models suffer no accuracy loss and are even more efficient than the original models.

1.3 Thesis Organization

We organize the rest of the thesis as follows:

- *Chapter 2* elaborates on layered obfuscation and discusses the taxonomy of obfuscation techniques.
- *Chapter 3* discusses the threats of symbolic execution to control-flow obfuscation and presents our idea to design symbolic opaque predicates.
- *Chapter 4* introduces the software-tampering threats faced by mobile apps and presents our N-version obfuscation approach.
- *Chapter 5* discusses the piracy issues faced by deep learning software and presents our simulation-based approach for obfuscating deep learning models.

- *Chapter 6* presents related work and discusses the novelty of this thesis.
- *Chapter 7* concludes this thesis and discusses our future work.

□ **End of chapter.**

Chapter 2

Taxonomy of Obfuscation Techniques

This chapter systematizes the knowledge of present obfuscation techniques for layered security. It aims to assist developers in choosing appropriate obfuscation techniques when designing layered obfuscation solutions in practice.

2.1 Rationale

Software obfuscation has been developed for over 30 years. A problem always confusing the communities is what security strength the technique can achieve. Nowadays, this problem becomes even harder as the software economy becomes more diversified. Inspired by the classic idea of layered security for risk management, we propose *layered obfuscation* as a promising way to realize reliable software obfuscation. Our concept is based on the fact that real-world software is usually complicated. Merely applying one or several obfuscation approaches in an ad-hoc way cannot achieve good obscurity. Layered obfuscation, on the other hand, aims to mitigate the risks of reverse software engineering by integrating different obfuscation techniques as a whole solution.

In this chapter, we demonstrate the idea of layered obfuscation and develop a taxonomy of obfuscation techniques to promote the idea. Following our taxonomy hierarchy, the obfuscation strategies under different branches are orthogonal to each other. In this way, it can assist developers in choosing obfuscation techniques and designing layered obfuscation solutions based on their specific requirements.

2.2 Motivating Examples

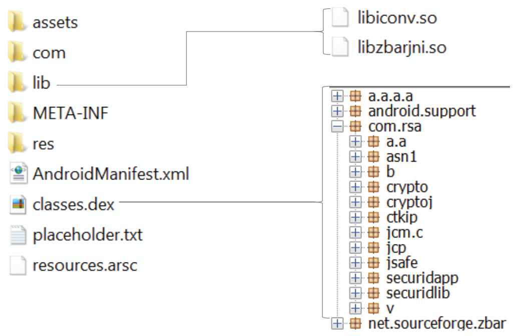
In this section, we discuss the obfuscation requirements of real-world software. We choose two prevalent types of software as our motivating examples, *i.e.,* mobile apps in client-server mode and JavaScript programs in browser-server mode.

2.2.1 Obfuscating Mobile Apps

We choose *RSA SecureID Software Token*[1] as a sample Android app to discuss the requirements of obfuscation. Figure 2.1(a) demonstrates the components of the app installation package. Its major component is `classes.dex` which contains all Java classes coded by developers. It implements user interfaces (UI) based on the APIs of Android framework and JDK. Other UI-related materials (*e.g.,* layout, images, and texts) are mainly within the folder of `res`. Since native codes are advantageous over Java bytecodes when implementing some features, the app also employs native codes which are within the folder of `lib`. Besides, there is a manifest file and other folders to store particular data, such as licenses and fonts.

Because the main feature of the app is to generate one-time passwords, the corresponding password generation codes and seeds should be most critical for protection. However, current mainstream obfuscation techniques (*e.g.,* lexical obfuscation and control-flow obfuscation) mainly focuses on general codes, such as Java codes or

[1]https://play.google.com/store/apps/details?id=com.rsa.securidapp&hl=en_US



(a) The components of a sample Android app, RSA SecureID Software Token.

Type	URL
html	https://magenta.tensorflow.org/demos/performance_rnn/index.html
css	https://fonts.googleapis.com/css?family=Roboto:300,400,500,700
css	https://www.gstatic.com/external_hosted/.../mdl_css-indigo-blue-bundle.css
javascript	https://www.google.com/js/gweb/analytics/autotrack.js
png	https://storage.googleapis.com/.../magenta-logo-bottom-text2.png
javascript	https://storage.googleapis.com/.../performance_rnn/bundle.js
javascript	https://ssl.google-analytics.com/ga.js
woff2	https://fonts.gstatic.com/s/roboto/v18/KFOmCnqEu92Fr1Mu4mxK.woff2
woff2	https://fonts.gstatic.com/s/roboto/v18/KFOlCnqEu92Fr1MmWUlfBBc4.woff2
woff2	https://fonts.gstatic.com/s/roboto/v18/KFOlCnqEu92Fr1MmEU9fBBc4.woff2
mp3	https://storage.googleapis.com/.../demos/SalamanderPiano/A0v1.mp3
mp3	https://storage.googleapis.com/.../demos/SalamanderPiano/C1v1.mp3
...	
json	https://storage.googleapis.com/.../models/performance_rnn/dljs/manifest.json
binary	https://storage.googleapis.com/.../models/performance_rnn/dljs/fully_connected_biases
binary	https://storage.googleapis.com/.../models/performance_rnn/dljs/fully_connected_weights
binary	https://storage.googleapis.com/... /dljs/rnn_multi_rnn_cell_cell_0_basic_lstm_cell_bias
binary	https://storage.googleapis.com/.../dljs/rnn_multi_rnn_cell_cell_0_basic_lstm_cell_kernel

(b) The components of a sample JavaScript application, Performance RNN.

Figure 2.1: Motivating examples for layered obfuscation.

native codes. While these approaches can make the app unreadable in some sense, it is hard to evaluate the resilience of the obfuscated codes to particular reverse-engineering attacks, such as stealing the seeds. We think a promising way to tackle the problem should be based on risk management. If all the risks can be properly mitigated, developers should be confident about the obfuscation solution. Because a risk may exist in any components of the package, the obfuscation solution should integrate different techniques to mitigate corresponding risks.

2.2.2 Obfuscating JavaScript Programs

Our JavaScript example is *Performance RNN*[2] , which is a web application that can play piano automatically based on recurrent neural networks (RNN) implemented with *tensorflow.js*. Figure 2.1(b) demonstrates the components of the web retrieved by a browser to launch the application. Similar to Android apps, these components are heterogeneous. It contains a primary HTML file (`index.html`) as the web entry, a CSS file and related pictures defining the appearance, and a JavaScript file (`bundle.js`) which implements the deep learning algorithms. Besides, there are several binary files that define an RNN model, and dozens of mp3 files to play each note of a piano.

According to the feature of the application, we infer that the key assets of the program should be the RNN model and related algorithms. Therefore, a competent obfuscation solution should at least obfuscate the model files and `bundle.js`. It may further randomize the names of the mp3 files to confuse reverse engineers. However, a better way to obfuscate the application should be based on risk analysis and risk mitigation.

In brief, these two examples demonstrate that practical obfuscation requirements are usually complicated. They also explain why obfuscation cannot be as secure as other security primitives. Furthermore, it indicates that layered security should be a promising way of obfuscating real-world software.

2.3 Our Study Approach

When designing layered obfuscation solutions for specific applications, developers should be knowledgeable about available obfuscation techniques. To meet this need, we develop a taxonomy of obfuscation techniques and survey present obfuscation techniques

[2]https://magenta.tensorflow.org/performance-rnn-browser

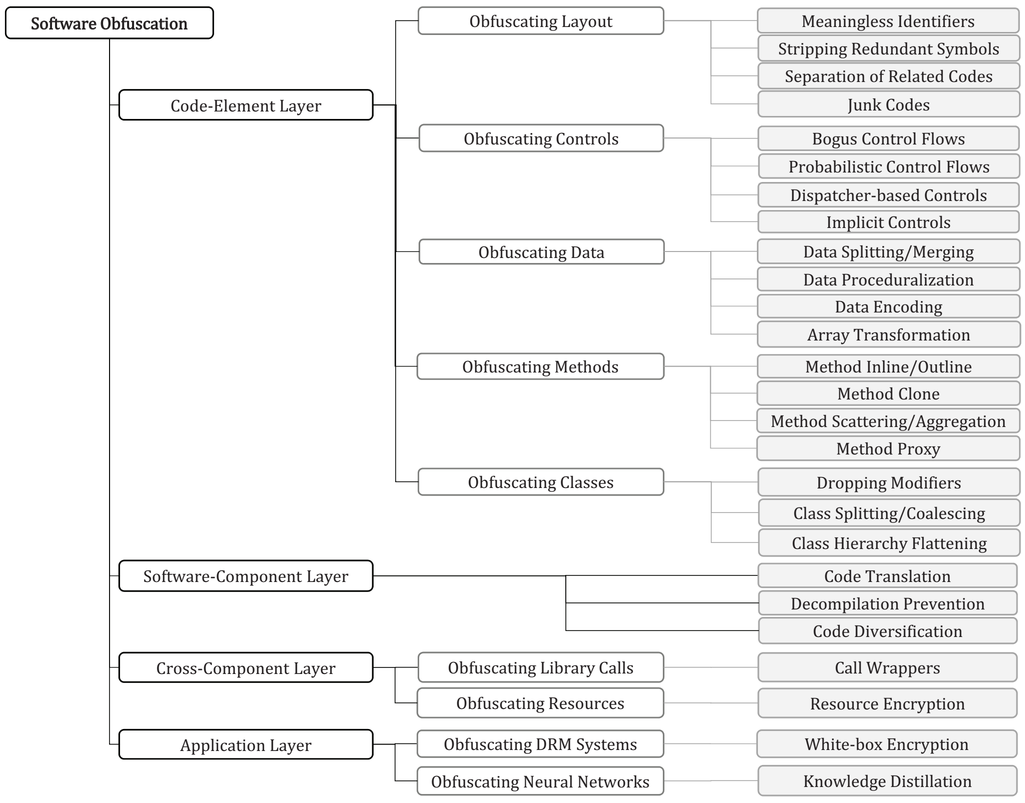


Figure 2.2: The taxonomy of software obfuscation techniques.

under the taxonomy framework. Because we aim to promote the idea of layered security in software obfuscation, the taxonomy developed in this chapter is different from previous ones.

2.3.1 Survey Scope

This work considers all obfuscation techniques that can be adopted at the developers' side, including those obfuscation transformations for source codes, bytecodes, and assembly codes. We do not discuss other obfuscation techniques that require modifying hardware or computing systems, such as address space randomization [17] and instruction set randomization [14].

2.3.2 Survey Approach

Figure 2.2 overviews our proposed obfuscation taxonomy. In the first level, we categorize present obfuscation techniques into four layers according to their obfuscation targets. The first layer is *code-element layer* which obfuscates particular elements of code snippets, including the layouts, controls, data, functions, and classes. The second layer is *software-component layer* which targets on an entire software component, such as a Java library or an ELF (executable file format) file. The third layer is *cross-component layer* which focuses on the interfaces (*e.g.,* JNI) among different components of a software package. Besides, there are unique obfuscation techniques proposed for specific applications, denoted as *application layer*. A famous example of such obfuscation techniques is white-box encryption for DRM (digital right management) systems [36]. In the second level of the taxonomy, we fork each layer into several sub-categories if the obfuscation targets can be further classified in a fine-grained manner. Finally, the leaf nodes of the taxonomy hierarchy are various obfuscation strategies for particular obfuscation targets.

In our taxonomy, the obfuscation strategies under different branches are orthogonal to each other. Therefore, it can assist developers in locating appropriate strategies based on the characteristics of the target software. Then they can choose a combination of several obfuscation techniques by further considering their performance, such as cost, potency, and resilience.

Note that our taxonomy is different from previous work (*e.g.,* [40, 151]) as the taxonomy is target-oriented by considering software packages composed of heterogeneous components.

2.4 Code-Element-Layer Obfuscation

This section discusses the obfuscation techniques for specific code elements. This layer covers most of the publications in software obfuscation area. According to what elements an obfuscation technique targets, we divide this category into five sub-categories: *obfuscating layouts*, *obfuscating controls*, *obfuscating data*, *obfuscating functions*, and *obfuscating classes*.

2.4.1 Obfuscating Layout

Layout obfuscation scrambles the layout of codes or instructions while keeping the original syntax intact. This section discusses four layout obfuscation strategies: meaningless classifiers, stripping redundant symbols, separating related codes, and junk codes.

Meaningless Identifiers

This approach is also known as lexical obfuscation which transforms meaningful identifiers to meaningless ones. For most programming languages, adopting meaningful and uniform naming rules (*e.g.,* Hungarian Notation [160]) is required as a good programming practice. Although such names are specified in source codes, some would remain in the released software by default. For example, the names of global variables and functions in C/C++ are kept in binaries, and all names of Java are reserved in bytecodes. Because such meaningful names can facilitate adversarial program analysis, we should scramble them. To make the obfuscated identifiers more confusing, Chan *et al.* [30] proposed to deliberately employ the same names for objects of different types or within different domains. Such approaches have been adopted by ProGuard as a default obfuscation scheme for Java programs.

Stripping Redundant Symbols

This strategy strips redundant symbolic information from released software, such as the debug information for most propgrams [114]. Besides, there are other redundant symbols for particular formats of programs. For example, ELF files contain symbol tables which record the pairs of identifiers and addresses. When adopting default compilation options to compile C/C++ programs, such as using LLVM [106], the generated binaries contain such symbol tables. To remove such redundant information, developers can employ the `strip` tool of Linux. Another example with redundant information is Android smali codes. By default, the generated smali codes contain information started with `.line` and `.source`, which can be removed for obfuscation purposes [47].

Separating Related Codes

A program is more easy to read if its logically related codes are also physically close [40]. Therefore, separating related codes or instructions can increase the difficulties in reading. It is applicable to both source codes (*e.g.,* reordering variables [114]) and assembly codes (*e.g.,* reordering instructions [178]). In practice, employing unconditional jumps to rewrite a program is a popular approach to achieve this. For example, developers can shuffle the assembly codes and then employ `goto` to reconstruct the original control flow [192]. This approach is popular for assembly codes and Java bytecodes with the availability of `goto` instructions [47].

Junk Codes

This strategy adds junk instructions which are not functional. For binaries, we can add no-operation instructions (`NOP` or `0x00`) [47, 119]. Besides, we can also add junk methods, such as adding defunct methods in Android smali codes [47]. The junk codes can typically change the signatures of the codes, and therefore escape static pattern recognition.

Because layout obfuscation does not tamper with the original code syntax, it is less prone to compatibility issues or bugs. Therefore, such techniques are the most favorite ones in practice. Moreover, the techniques of meaningless identifiers and stripping redundant symbols can reduce the size of programs, which further make them attractive. However, the potency of the layout obfuscation is limited. It is resilient to deobfuscation attacks because some transformations are one-way, which cannot be reversed. However, some layout information can hardly be changed, such as the method identifiers from Java SDK. Such residual information is essential for adversaries to recover the obfuscated information. For example, Bichsel *et al.* [18] tried to deobfuscated ProGuard-obfuscated apps, and they successfully recovered around 80% names.

2.4.2 Obfuscating Controls

Control obfuscation transforms the controls of codes to increase the program complexity. This can be achieved via bogus control flows, probabilistic control flows, dispatcher-based controls, and implicit controls.

Bogus Control Flows

Bogus control flows refer to the control flows that are deliberately added to a program but will never be executed. It can increase the complexity of a program, *e.g.,* in McCabe complexity [121] or Harrison metrics [82]. For example, McCabe complexity [121] is calculated as the number of edges on a control-flow graph minus the number of nodes, and then plus two times of the connected components. To increase the McCabe complexity, we can either introduce new edges or add both new edges and nodes to a connected component.

```
int a, b;
...
if(7a * a – 1 != b){
    //always true
    originalCodes();
} else { //optional
    bogusCodes();
}
```

(a) Opaque constant.

```
int x; //for any x>0
while(x>1){
  if(x%2==1)
    x=x*3+1;
  else x=x/2;
  if(x==1)//always reachable
   originalCodes();
}
```

(b) Collatz conjecture.

```
int *p = &x;
int *q = &x;
if((*p)%2 == 0){
    y = x+1;
} else {
    y = x+1;
    y = y+2;
}
```

```
if((*q)%2 == 0){
    y = y+3;
    x = y+3;
} else {
    x = y+3;
}
```

(c) Dynamic opaque predicate.

Figure 2.3: Control-flow obfuscation with opaque predicates.

To guarantee the unreachability of bogus control flows, Collberg *et al.* [40] suggested employing opaque predicates. They defined opaque predict as the predicate whose outcome is known during obfuscation time but is difficult to deduce by static program analysis. In general, an opaque predicate can be constantly true (P^T), constantly false (P^F), or context-dependent ($P^?$). There are three methods to create opaque predicates: numerical schemes, programming schemes, and contextual schemes.

Numerical Schemes

Numerical schemes compose opaque predicates with mathematical expressions. For example, $7x^2 - 1 \neq y^2$ is constantly true for all integers x and y. We can directly employ such opaque predicates to introduce bogus control flows. Figure 2.3(a) demonstrates an example, in which the opaque predicate guarantees that the bogus control flow (*i.e.,* the else branch) will not be executed. However, attackers would have higher chances to detect them if we employ the same opaque predicates frequently in an obfuscated program. Arboit [3], therefore, proposed to generate a *family* of such opaque predicates automatically, such that an obfuscator can choose a unique opaque predicate each time.

Another mathematical approach with higher security is to employ *crypto functions*, such as hash function $\mathcal{H}$ [156], and homomorphic encryption [198]. For example, we can substitute a predicate $x == c$ with $\mathcal{H}(x) == c_{hash}$ to hide the solution of x for this equation. Note that such an approach is generally employed by malware to evade dynamic program analysis. We may also employ crypto functions to encrypt equations which cannot be satisfied. However, such opaque predicates incur much overhead.

To compose opaque constants resistant to static analysis, Moser *et al.* [123] suggested employing 3-SAT problems, which are NP-hard. This is possible because one can have efficient algorithms to compose such hard problems [153]. For example, Tiella and Ceccato [166] demonstrated how to compose such opaque predicates with k-clique problems.

To compose opaque constants resistant to dynamic analysis, Wang *et al.* [176] proposed to compose opaque predicates with a form of *unsolved conjectures* which loop for many times. Because loops are challenging for dynamic analysis, the approach in nature should be resistant to dynamic analysis. Examples of such conjectures include Collatz conjecture, $5x + 1$ conjecture, Matthews conjecture. Figure 2.3(b) demonstrates how to employ Collatz conjecture to introduce bogus control flows. No matter how we initialize x, the program terminates with $x = 1$, and `originalCodes()` can always be executed.

Programming Schemes

Because adversarial program analysis is a major threat to opaque predicates, we can employ challenging program analysis problems to compose opaque predicates. Collberg *et al.* suggested two classic problems, *pointer analysis* and *concurrent programs*.

In general, pointer analysis refers to determining whether two pointers can or may point to the same address. Some pointer analysis problems can be NP-hard for static analysis or even undecidable [102]. Another advantage is that pointer operations are very efficient during execution. Therefore, developers can compose resilient and efficient opaque predicts with well-designed pointer analysis problems, such as maintaining pointers to some objects with dynamic data structures [42].

Concurrent programs or parallel programs is another challenging issue. In general, a parallel region of n statements has $n!$ different ways of execution. The execution is not only determined by the program, but also by the runtime status of a host computer. Collberg *et al.* [42] proposed to employ concurrent programs to enhance the pointer-based approach by concurrently updating the pointers. Majumdar *et al.* [117] proposed to employ distributed parallel programs to compose opaque predicates.

Besides, some approaches compose opaque predicates with programming tricks, such as leveraging *exception handling mechanisms*. For example, Dolz and Parra [53] proposed to use the `try/catch` mechanism to compose opaque predicates for `.Net` and Java. The exception events include division by zero, null pointer, index out of range, or even particular hardware exceptions [32]. The original program semantics can be achieved via tailored exception handling schemes. However, such opaque predicates have no security basis, and they are vulnerable to advanced handmade attacks.

Contextual Schemes

Contextual schemes can be employed to compose variant opaque predicates(*i.e.,* $\{P^?\}$). The predicates should hold some deterministic properties such that they can be employed to obfuscate programs. For example, Drape [56] proposed to compose such opaque predicates which are invariant under a contextual constraint, *e.g.,* the opaque predicate $x \bmod 3 == 1$ is constantly true if $x \bmod 3 : 1\ ?\ x{+}{+} : x = x + 3$. Palsberg *et al.* [128] proposed dynamic opaque predicates, which include a sequence of correlated predicates. The evaluation result of each predicate may vary in each run. However, as long as the predicates are correlated, the program behavior is deterministic. Figure 2.3(c) demonstrates an example of dynamic opaque predicates. No matter how we initialize $\star$p and $\star$q, the program is equivalent to $y = x + 3, x = y + 3$.

The resistance of bogus control flows mostly depends on the security of opaque predicates. An ideal security property for opaque predicates is that they require worst-case exponential time to break but only polynomial time to construct. Note that some opaque predicates are designed with such security concerns but may be implemented with flaws. For example, the 3-SAT problems proposed by Ogiso *et al.* [127] are based on trivial problem settings which can be easily simplified. If such opaque predicates are implemented properly, they would be promising to be resilient.

Probabilistic Control Flows

Bogus control flows can make troubles to static program analysis. However, they are vulnerable to dynamic program analysis because the bogus control flows are inactive. The idea of probabilistic control flows adopts a different strategy to tackle the threat [132]. It introduces replications of control flows with the same semantics but different syntax. When receiving the same input several times, the program can behave differently for different execution times. The technique is also useful for combating side-channel attacks [44].

Note that the strategy of probabilistic control flows is similar to bogus control flows with contextual opaque predicates. But they are different in nature as contextual opaque predicates introduce dead paths, although they do not introduce junk codes.

```
int a = 1;
int b = 2;
while(a < 10){
    b = a+b;
    if(b>10){
        b--;
    }
    a++;
}
printf("%d", b);
```

(a) Source code.

```
    int a = 1;
    int b = 2;
L1: if(!(a<10))
        goto L3;
    b=a+b;
    if(!(b>10))
        goto L2;
    b--;
L2: a++;
        goto L1;
L3: printf("%d", b);
```

(b) Dismantling while.

```
int swVar = 1;
switch (swVar){
    case 1:
            a = 1;
            b = 2;
            swVar = 2;
            break;
    case 2:
            if(!(a<10))
                swVar = 6;
            else
                swVar = 3;
            break;
    case 3:
            b = b+a;
            if(!(b>10))
                swVar = 5;
            else
                swVar = 4;
            break;
    case 4:
            b--;
            swVar = 5;
            break;
    case 5:
            a++;
            swVar = 2;
            break;
    case 6:
            printf("%d", b);
            break;
}
```

(c) Using switch.

Figure 2.4: Control-flow flattening example.

Dispatcher-Based Controls

A dispatcher-based control determines the next blocks of codes to be executed during runtime. Such controls are essential for control obfuscation because they can hide the original control flows against static program analysis.

One major dispatcher-based obfuscation approach is control-flow flattening, which transforms codes of depth into shallow ones with more complexity. Wang *et al.* [170] firstly proposed the approach. Figure 2.4 demonstrates an example from their paper that transforms a `while` loop into another form with `switch-case`. To realize such transformation, the first step is to transform the code into an equivalent representation with `if-then-goto` statements as shown in Figure 2.4(b); then they modify the `goto` statements with `switch-case` statements as shown in Figure 2.4(c). In this way, the original program semantics is realized implicitly by controlling the data flow of the switch variable. Because the execution order of code blocks is determined by the variable dynamically, one cannot know the control flows without executing the program. Cappaert and Preneel [28] formalized control-flow flattening as employing a dispatcher node (*e.g.,* `switch`) that controls the next code block to be executed; after executing a block, control is transferred back to the dispatcher node. Besides, there are several enhancements to code-flow flattening. For example, to enhance the resistance to static program analysis on the switch variable, Wang *et al.* [169] proposed to introduce pointer analysis problems. To further complicate the program, Chow *et al.* [38] proposed to add bogus code blocks.

László and Kiss [105] proposed a control-flow flattening mechanism to handle the controls of C++ programs, such as `try/catch`, `while/do`, and `continue`. The mechanism is based on abstract syntax tree and employs a fixed pattern of layout. For each block of code to obfuscate, it constructs a `while` statement in the outer loop and a `switch-case` compound inside the loop. The `switch/case` compound implements the original program semantics, and the `switch` variable is also employed to terminate the outer loop. Cappaert and Preneel [28] found that the mechanisms might be vulnerable to local analysis, *i.e.,* the switch variable is immediately assigned such that adversaries can infer the next block to execute by only looking into a current block. They proposed a strengthened approach with several tricks, such as employing reference assignment (*e.g.,* $swVar = swVar + 1$) instead of direct assignment (*e.g.,* $swVar = 3$), replacing the assignment via `if/else` with a uniform assignment expression, and employing one-way functions in calculating the successor of a basic block.

Besides control-flow flattening, there are several other dispatcher-based obfuscation investigations (*e.g.,* [73, 113, 150, 193]). Linn and Debray [113] proposed to obfuscate binaries with branch functions that guide the execution based on the stack information. Similarly, Zhang *et al.* [193] proposed to employ branch functions to obfuscate object-oriented programs, which define a unified method invocation style with an object pool. To enhance the security of such mechanisms, Ge *et al.* [73] proposed to hide the control information in another standalone process and employ inter-process communications. Schrittwieser and Katzenbeisser [150] proposed to employ diversified code blocks which implement the same semantics.

Dispatcher-based obfuscation is resistant against static analysis because it hides the control-flow graph of a software program. However, it is vulnerable to dynamic program analysis or hybrid approaches. For example, Udupa *et al.* [168] proposed a hybrid approach to reveal the hidden control flows with both static analysis and dynamic analysis.

Implicit Controls

This strategy converts explicit control instructions to implicit ones. It can hinder reverse engineers from addressing the correct control flows. For example, we can replace the control instructions of assembly codes (*e.g.,* `jmp` and `jne`) with a combination of `mov` and other instructions which implement the same control semantics [6].

Note that all existing control obfuscation approaches focus on syntactic-level transformation, while the semantic-level protection has rarely been discussed. Although they may demonstrate different strengths of resistance to attacks, their obfuscation effectiveness concerning semantic protection remains unclear.

2.4.3 Obfuscating Data

Present data obfuscation techniques focus on common data types, such as integers, strings, and arrays. We can transform data via splitting, merging, proceduralization, encoding, *etc.*

Data Splitting/Merging

Data splitting distributes the information of one variable into several new variables. For example, a boolean variable can be split into two boolean variables, and performing logical operations on them can get the original value.

Data merging, on the other hand, aggregates several variables into one variable. Collberg *et al.* [41] demonstrated an example that merges two 32-bit integers into one 64-bit integer. Ertaul and Venkatesh [60] proposed another method that packs several variables into one space with discrete logarithms.

Data Proceduralization

Data Proceduralization substitutes static data with procedure calls. Collberg *et al.* [41] proposed to substitute strings with a function which can produce all strings by specifying paticular parameter values. Drape [55] proposed to encode numerical data with two inverse functions f and g. To assign a value v to a variable i, we assign it to an injected variable j as $j = f(v)$. To use i, we invoke $g(j)$ instead.

Data Encoding

Data encoding encodes data with mathematical functions or ciphers. Ertaul and Venkatesh [60] proposed to encode strings with Affine ciphers (*e.g.,* Caser cipher) and employ discrete logarithms to pack words. Fukushima *et al.* [65] proposed to encode the clear numbers with `exclusive or` operations and then decrypt the computation result before output. Kovacheva [98] proposed to encrypt strings with the RC4 cipher and then decrypt them during runtime.

Array Transformation

Array is one most commonly employed data structure. To obfuscate arrays, Collberg *et al.* [41] discussed several transformations, such as splitting one array into several subarrays, merging several arrays into one array, folding an array to increase its dimension, or flattening an array to reduce the dimension. Ertaul and Venkatesh [60] suggested transforming the array indices with composite functions. Zhu *et al.* [199, 200] proposed to employ homomorphic encryption for array transformation, including index change, folding, and flattering. For example, we can shuffle the elements of an array with $i * m \bmod n$, where i is the original index, n is the size of the original array, and m and n are relatively prime.

2.4.4 Obfuscating Methods

Method Inline/Outline

A method is an independent procedure that can be called by other instructions of the program. Method inline replaces the original procedural call with the function body itself. Method outline operates in the opposite way which extracts a sequence of instructions and abstracts a method. They are good companies which can obfuscate the original abstraction of procedures [40].

Method Clone

If a method is heavily invoked, we can create replications of the method and randomly call one of them. To confuse adversarial interpretation, each version of the replication should be unique somehow, such as by adopting different obfuscation transformations [40] or different signatures [59].

Method Aggregation/Scattering

The idea is similar to data obfuscation. We can aggregate irrelevant methods into one method or scattering a method into several methods [40, 114].

Method Proxy

This approach creates proxy methods to confuse reverse engineering. For example, we can create the proxies as public static methods with randomized identifiers. There can be several distinct proxies for the same method [47]. The approach is extremely useful when the method signatures cannot be changed [135].

2.4.5 Obfuscating Classes

Obfuscating classes shares some similar ideas with obfuscating methods, such as splitting and clone [41]. However, since class only exists in object-oriented programming languages, such as `JAVA` and `.NET`, we discuss them as a unique category. Below we present the major strategies for obfuscating classes.

Dropping Modifiers

Object-oriented programs contain modifiers (*e.g.,* public, private) to restrict the access to classes and members of classes. Dropping modifiers removes such restrictions and make all members public [135]. This approach can facilitate the implementation of other class obfuscation methods.

Splitting/Coalescing Class

The idea of coalescing/splitting is to obfuscate the intent of developers when design the classes [162]. When coalescing classes, we can transfer local variables or local instruction groups to another class [66].

Class Hierarchy Flattening

Interface is a powerful tool for object-oriented programs. Similar to method proxy, we can create proxies for classes with interfaces [162]. However, a more potent way is to break the original inheritance relationship among classes with interfaces. By letting each node of a subtree in the class hierarchy implementing the same interface, we can flatten the hierarchy [63].

2.5 Software-Component-Layer Obfuscation

Now we present the obfuscation techniques which do not emphasize particular code syntax or elements, including code translation, decompilation prevention, encoding instructions, and diversification.

2.5.1 Code Translation

Wang *et al.* [174] proposed *translingual obfuscation*, which introduces obscurity by translating the programs written in C into ProLog before compilation. Because ProLog adopts a different program paradigm and execution model from C, the generated binaries should become harder to understand. In an extreme case, Domas [54] considered all high-level instructions should be obfuscated. He proposed *movobfuscation*, which employs only one instruction (*i.e.,* `mov`) to compile the program. The idea is feasible because `mov` is Turing complete [52].

2.5.2 Decompilation Prevention

Preventive obfuscation raises the bar for adversaries to obtain code snippets in readable formats. It is generally designed for non-scripting programming languages, such as C/C++ and Java. For such software, a decompilation or disassembly phase is required to translate machine codes (*e.g.,* binaries) into human readable formats.

Preventive obfuscation, therefore, obstructs this decoding phase by introducing decompilation errors.

Linn and Debray [113] proposed an anti-disassembly approach for binaries. Their approach deters disassembling algorithms by inserting uncompleted instructions after unconditional jumps. In this way, the uncompleted instructions are unreachable as junk codes. If a disassembler cannot handle such uncompleted instructions, they will have troubles when separating instructions. This approach can be further strengthened with some control obfuscation techniques [134]. Chan and Yang [30] proposed several lexical tricks to impede Java decompilation. The idea is to modify bytecodes directly by employing reserved keywords to name variables and functions. This is possible because only the frontend performs the validation check of identifiers. The resulting modified program can still run correctly, but it would cause troubles for decompilation tools.

Moreover, there are some encryption-based approaches which can hide the real instructions from static analysis. A typical application is the class encryption feature for Android apps [177]. By encrypting the `classes.dex`, this feature can hide the Java classes from being decompiled by popular reverse engineering tools, such as Apktool[3] and dex2jar[4]. Besides, encryption-based approaches are widely employed by malware as camouflages [192].

2.5.3 Code Diversification

Previous obfuscation approaches focus on introducing obscurities to one software component, while code diversification generates multiple obfuscated versions of the component simultaneously [104]. Ideally, it can pose equivalent barriers for adversaries to reverse engineer each particular version. Therefore, code diversification can impede large-scale and reproductive attacks to homogeneous software [64, 89]. It is also a technique widely employed by

[3]https://ibotpeaches.github.io/Apktool/
[4]https://github.com/pxb1988/dex2jar

malware camouflage, which creates different copies of malware to evade anti-virus detection [192].

Code diversification generally relies on some randomization mechanisms to introduce variance. Lin *et al.* [112] proposed to generate different layout of data structures during each compilation. In this way, each compiled version contains a unique layout of data objects, such as structures, classes, and stack variables declared in functions. This can be achieved through an algorithm which automatically discovers the potential data objects that can be randomized [182]. By embedding some security designs, code diversification can be resilient to specific attacks [104]. For example, Crane *et al.* [45] proposed to randomize the tables of pointers to deter code-reuse attacks. In Chapter 4, we will present a diversification-based approach for tamper-resilience [187].

2.6 Cross-Component-Layer Obfuscation

Modern software package generally contains several components, such as the components written by developers and other libraries. This phenomenon can facilitate software development and distribution, but it also raises challenging issues for obfuscation. In particular, developers cannot modify the function identifiers implemented in other libraries. To obfuscate such information, Collberg *et al.* [40] suggested substituting common patterns of function invocation with less obvious ones. However, he did not present the details. Recently, Kovacheva [98] investigated the problem for Android apps. He proposed to obfuscate the native calls (*e.g.,* to `libc` libraries) via a proxy, which is an obfuscated class that wraps the native functions. The feature is available in some commercial obfuscation tools, such as DexProtector. Bohannon and Holmes [20] investigated a similar problem for Windows powershell scripts. To obfuscate an invocation command to Windows objects, they proposed to create a nonsense string first and then leverage Windows string operators to transform the string to a valid command during runtime. Besides, some state-of-the-art obfuscation tools (*e.g.,* DexProtector) can encrypt the resource files of software packages and implement functions to decrypt them during runtime.

2.7 Application-Layer Obfuscation

Note that our previously discussed techniques are unrelated to the functionality of the software. In this section, we discuss several obfuscation techniques which are designed for the software with specific features, such as DRM systems and neural networks.

2.7.1 Obfuscating DRM Systems

A DRM system controls the access of users to multimedia files. The favorite solutions of DRM systems are based on content encryption. For such solutions, one critical challenge is to hide decryption keys, especially when attackers can have full access to the decryption software and the computing environment. *White-box encryption* is an obfuscation approach which can withstand key extraction attacks [37].

In high level, a white-box encryption approach pre-evaluates all the operations related to keys and replaces corresponding codes. For example, the original DES[5] algorithm contains 16 rounds of Feistel functions, each `XOR`s the plaintext with a round key, and then employs a lookup table and a permutation box to produce the output. Chow *et al.* [37] proposed to substitute this procedure with a round-key-specific lookup table. In this way, it can hide both the key and round keys. To be resistant to cryptanalysis, Chow *et al.*proposed to further apply bijections and networked encodings for each encryption round [37]. The strategy is also applicable for

[5]FIPS 46, The Data Encryption Standard

AES[6] [36].

2.7.2 Obfuscating Neural Networks

Deep learning has achieved radical developments in the last decade. It is a new paradigm of programming, known as *Software 2.0*[7]. Previous studies show that the structure of neural networks is a critical factor to improve the accuracy of deep learning models. Therefore, the structural information of private machine learning models is a key intellectual property for such software. For example, our JavaScript software in Section 2.2 contains an RNN model and should be protected.

To obfuscate deep learning models, we proposed a simulation-based obfuscation method [184]. The method distills the knowledge of well-trained deep learning models and reloads such knowledge into shallow networks. In this way, the shallow networks retain the same accuracy as the original models, but they have poor learning abilities. Attackers can learn very few useful settings from the simulation networks. More details will be presented in Chapter 5

2.8 Threats to Validity

In this section, we justify the validity of layered obfuscation as a promising way to obfuscate real-world software. A major threat to this idea is whether there are already approaches which can obfuscate all software with security guarantee, *i.e.,* they ensure that the essential program semantics are well protected and demonstrate adequate hardness for adversaries to recover the semantics. However, we cannot find such approaches in the literature. Below, we justify this claim from both the perspectives of practical code obfuscation and theoretical program obfuscation research.

[6]FIPS 197, Advanced Encryption Standard

[7]https://medium.com/@karpathy/software-2-0-a64152b37c35

2.8.1 Practical Obfuscation Techniques

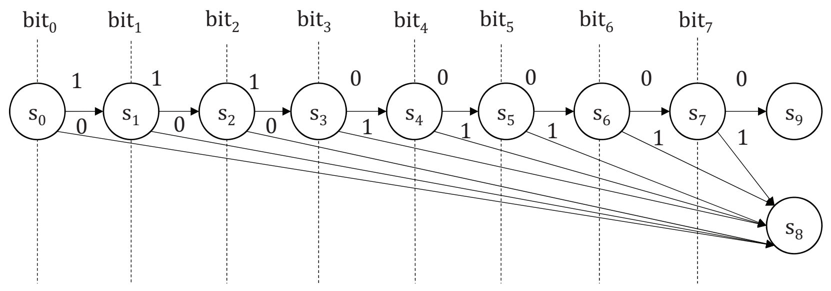
As we have discussed in previous sections, most practical obfuscation techniques focus on obfuscating particular information. They cannot provide guarantee that the obfuscated software is secure against reverse engineering attacks.

Furthermore, real-world obfuscation practice usually adopts one obfuscation technique or combines several techniques in an ad-hoc way. For example, ProGuard is the most popular obfuscation tool for Android apps, and it is the default one embedded in Android Studio for free use. ProGuard can only obfuscate the identifiers of Java programs. Premium obfuscation tools (*e.g.,* DexGuard and DexProtector) are more powerful, but only less than 0.16% of real-world apps employ such premium obfuscation tools [177]. From their official websites, we can find these tools support many obfuscation features, including encryption of strings, encryption of classes, hiding method calls, native code obfuscation, native code encryption, and *etc*. While each of these features is powerful for particular threats, there is little instruction about how to integrate them effectively. The similar situation also exists for iOS app obfuscation [172]. Therefore, the taxonomy developed in this chapter can provide more reference to developers regarding how to select and integrate different obfuscation techniques.

2.8.2 Theoretical Obfuscation Research

From the theoretical perspective, scientists have already found an algorithm (*i.e.,* graded encoding) which can obfuscate all programs with a compelling security property: *indistinguishability* [71, 201, 109]. Since such results may confuse readers, next, we clarify the gaps between such theoretical research and real-world obfuscation problems with a sample graded encoding mechanism.

In general, there are two phases to obfuscate a program with graded encoding: the first phase converts programs to matrix branch-



(a) Branching program (*i.e.,* if x of `int8` equals to 7).

$$\begin{bmatrix}1\\0\\0\\\vdots\\0\\0\\0\end{bmatrix}^T \quad M_0 = \begin{bmatrix}1&0&0&&0&1&0\\0&1&0&\cdots&0&0&0\\0&0&1&&0&0&0\\&\vdots&&\ddots&&\vdots&\\0&0&0&&1&0&0\\0&0&0&\cdots&0&1&0\\0&0&0&&0&0&1\end{bmatrix} \quad M_1 = \begin{bmatrix}1&1&0&&0&0&0\\0&1&0&\cdots&0&0&0\\0&0&1&&0&0&0\\&\vdots&&\ddots&&\vdots&\\0&0&0&&1&0&0\\0&0&0&\cdots&0&1&0\\0&0&0&&0&0&1\end{bmatrix} \quad \begin{bmatrix}0\\0\\0\\\vdots\\0\\0\\1\end{bmatrix}$$

head matrix   matrices for bit$_0$ tail matrix

(b) Matrix branching program.

$$Rand(M_0) = RM_{head}^{-1} \times M_0 \times RM_0 =$$

$$\begin{bmatrix}13&7&3&&3&1&0\\11&8&10&\cdots&13&5&3\\13&5&6&&12&8&3\\&\vdots&&\ddots&&\vdots&\\7&12&0&&1&13&6\\0&12&8&\cdots&0&1&15\\0&0&8&&0&0&1\end{bmatrix} \times \begin{bmatrix}1&0&0&&0&1&0\\0&1&0&\cdots&0&0&0\\0&0&1&&0&0&0\\&\vdots&&\ddots&&\vdots&\\0&0&0&&1&0&0\\0&0&0&\cdots&0&1&0\\0&0&0&&0&0&1\end{bmatrix} \times \begin{bmatrix}1&5&13&&15&3&2\\7&10&0&\cdots&2&7&6\\3&13&8&&6&7&14\\&\vdots&&\ddots&&\vdots&\\9&7&5&&8&14&5\\12&8&0&\cdots&8&13&9\\8&8&0&&0&8&1\end{bmatrix}$$

(c) Randomized matrix branching program.

Figure 2.5: Converting a program to a randomized matrix branching program.

ing programs (MBP) which can be evaluated after encryption; the second phase encrypts MBPs with graded encoding mechanisms. In particular, the first phase determines the limitation of program types that can be supported by theoretical obfuscation research, and the second phase incurs large overhead.

Converting to MBP

An MBP that computes a function f is a tuple

$$MBP_f = (Input, M_{head}, (M_{i,0}, M_{i,1})_{i \in l}, M_{tail})$$

$Input$ selects a matrix $M_{i,0}$ or $M_{i,1}$ for each i according to the corresponding bit of input; M_{head} is a row vector of size w; $(M_{i,0}, M_{i,1})_{i\in l}$ are matrix pairs of size $w \times w$ that encode program semantics; and M_{tail} is a column vector of size w.

Given an input x, the MBP computes an output $MBP_f(x) \in \{0, 1\}$ as follows:

$$MBP_f(x) = M_{head} \times (\prod_{i=1}^{l} M_{i,x_{input(k)}}) \times M_{tail}$$

Suppose the i-th matrix pair corresponds to the k-th bit of the input. If the k-th bit is 0, then $M_{i,0}$ is selected, or *vice versa*. The program output is the matrix multiplication result.

The conversion generally includes two steps: from a circuit P_f to a branching program BP_f, and from BP_f to MBP_f.

$P_f \rightarrow BP_f$: A branching program is a finite state machine. Barrington's Theorem states that we can convert any boolean formula (boolean circuit of fan-in-two, depth d) to a branching program of width 5 and length $\leq 4^d$ [15]. For boolean formulas $P_f \in \{0, 1\}$, the finite state machine has one start state, two stop states ($true$ and $false$), and several intermediate states. Figure 2.5(a) demonstrates an example which converts a boolean program $i == 7$ to a branching program. Suppose i is an integer of eight bits, the boolean formula is $b_0 \wedge b_1 \wedge b_2 \wedge \neg b_3 \wedge \neg b_4 \wedge \neg b_5 \wedge \neg b_6 \wedge \neg b_7$. We need 10 states to model the branching program: eight states (s_0-s_7) that accept each bit of input, and two stop states (s_8 for $false$, and s_9 for $true$).

$BP_f \rightarrow MBP_f$: This step computes each matrix of the MBP_f. In general, M_{head} can be an all-zero row vector except the first position is 1, and M_{tail} can be an all-zero column vector except the last position is 1. $(M_{i,0}, M_{i,1})_{i\in len}$ can be constructed from the adjacency matrices of each state. Figure 2.5(b) demonstrates the matrices corresponding to the first input bit of Figure 2.5(a).

Following such converting approaches, the elements of resulting matrices are either 1 or 0. Kilian [96] proposed that we can randomize these elements while retain its functionality.

$MBP_f \rightarrow RMBP_f$: We first generate $n+1$ random integer matrices RM_i and their inverse RM_i^{-1} of size $w \times w$. Then we multiply the original matrices with such random matrices as follows.

$$RM_{head} = M_{head} \times RM_0$$
$$RM_{0,0} = RM_0^{-1} \times M_{0,0} \times RM_1$$
$$RM_{0,1} = RM_0^{-1} \times M_{0,1} \times RM_1$$
$$\ldots$$
$$RM_{tail} = RM_n^{-1} \times M_{tail}$$

The randomization mechanism ensures that all randomization matrices RM_i would be canceled when evaluating $RMBP_f(x)$.

This phase reveals that the results of theoretical obfuscation research apply to arithmetic programs only. However, real software is more complicated which usually contains many other operations which cannot be converted to MBP directly or efficiently.

Graded Encoding

Although the randomized matrix branching program provides some security, it still suffers three kinds of attacks: partial evaluation, mixed input, and other attacks that do not respect the algebraic structure [72]. Graded encoding is proposed to defeat such attacks.

Graded encoding is based on multilinear maps. In general, a graded encoding scheme includes four components: *setup* that generates the public and private parameters of a system, *encoding* that defines how to encrypt a message with the private parameters, *operations* that declare the supported calculations with encrypted messages, and a *zero-testing function* that evaluates if the plain text of an encrypted message should be 0. *GGH scheme* is the first plausible solution to compose multilinear maps [70]. It is based on ideal lattices which encodes an element e over a quotient ring $R/\mathcal{I}$ as $e + \mathcal{I}$, where $\mathcal{I} = \langle g \rangle \subset R$ is the principal ideal generated by a short vector g. The four components of GGH are defined as follows.

Setup: Suppose the multilinear level is κ. The system generates an ideal-generator g (g and g^{-1} should be short), a large enough modulus q, and denominators $\{z_i\}$ from the ring R_q. Then we publish the zero-testing parameter as $p_{zt} = [h \prod_{i=1}^{\kappa} z_i/g]_q$, where h is a small ring element.

Encoding: The encoding of an element e in set S_{z_i} is computed as : $u := [(e + \mathcal{I})/z_i]_q$.

Operations: If two encodings are in the same set (*e.g.,* $u_1 := [c_1/z_i]_q$ and $u_2 := [c_2/z_i]_q$), then one can add them up $u_1 + u_2$. If the two encodings are from disjoint sets, one can multiply the two encodings $u_1 \cdot u_2$.

Zero-Testing Function: A zero testing function for a level-κ encoding u is defined as

$$IsZero(u) = \begin{cases} 1 & \text{if } ||[u \cdot p_{zt}]_q||_\infty \leq q^{3/4} \\ 0 & \text{otherwise} \end{cases}$$

Note that $u \cdot p_{zt} = h \cdot c/g$. If u is an encoding of 0, c should be a short vector in $\mathcal{I}$ and the product can be smaller than a threshold; otherwise, c should be a short vector in some coset of $\mathcal{I}$ and the product should be very large.

In brief, the scheme is based on noisy multilinear maps as the encoding of a value varies at different times. The only deterministic function is the zero-testing function. However, when a program becomes complex, the noise may overwhelm the signal. The size of q should be as large as possible to overwhelm the noise. This requirement largely limit the efficiency of graded encoding. Note that gradient encoding incurs polynomial overhead. Although the overhead is promising from the theoretical view, it is too large for practical usage. It has been shown that even obfuscating a 16-bit point function would result in a program of several GigBytes [1].

2.8.3 Other Supportive Work

Besides, there are investigations and obfuscation tools which coincide with our proposal of layered obfuscation. For example, Kuzurin *et al.* [100] found that the security properties for obfuscating general programs might be too strong for practical scenarios. They proposed to design specific security properties for particular obfuscation scenarios, such as hiding constants or generating resilient opaque predicates. The idea is consistent with our layered obfuscation approach, *i.e.,* an obfuscation approach cannot be secure-against-all but should only handle particular threats. Moreover, real-world obfuscation tools (*e.g.,* Obfuscator-LLVM [93] and DexGuard) already support combinations of different obfuscation techniques, which is a characteristic of layered obfuscation solutions. However, they are still very preliminary in offering systematic combination strategies. Our work, therefore, develops a novel taxonomy of obfuscation techniques which can assist developers in integrating them systematically.

2.9 Related Work

This chapter is a pilot study to survey obfuscation techniques for layered obfuscation. There are already several obfuscation surveys available, but they do not follow the layered obfuscation idea. The surveys of practical code obfuscation include [7, 56, 118, 151, 146]. Balakrishnan and Schulze [7] surveyed several major obfuscation approaches for both benign codes and malicious codes. Majumdar *et al.* [118] conducted a short survey that summarizes the control-flow obfuscation techniques using opaque predicates and dynamic dispatcher. Drape *et al.* [56] surveyed several obfuscation techniques via layout transformation, control-flow transformation, data transformation, language dependent transformations, *etc*. Roundy *et al.* [146] systematically studied obfuscation techniques for binaries, which have been frequently used by malware packers. Schrittwieser *et al.* [151] surveyed the resilience of obfuscation mechanisms to reverse engineering techniques. There are also surveys of theoretical obfuscation research, including [88] and [11]. Horvath *et al.* [88] studied the history of cryptography obfuscation, with a focus on graded encoding mechanisms. Barak [11] reviewed the importance of indistinguishability obfuscation. To our best knowledge, none of them follows a clear layered obfuscation approach.

2.10 Conclusions

To conclude, this chapter explores layered obfuscation which applies the idea of layered security to software obfuscation. To facilitate the adoption of the idea, we develop a novel obfuscation taxonomy and survey present obfuscation techniques based on the taxonomy. Our taxonomy categorizes present obfuscation techniques into four layers based on the difference of their obfuscation targets. Each layer further contains several sub-categories or obfuscation strategies. The obfuscation strategies under different branches of the taxonomy are orthogonal to each other. In this way, it can provide guidance for users when choosing obfuscation techniques for designing layered obfuscation solutions. We hope this chapter can inspire more investigations on layered obfuscation and encourage the development of new obfuscation techniques, which may not be secure-against-all, but can provide users more options in designing a layered obfuscation solution.

□ **End of chapter.**

Chapter 3

Symbolic Opaque Predicates

Symbolic execution is a program analysis technique which can be employed by attackers to deobfuscate programs. This chapter analyzes the limitations of symbolic execution and proposes a novel obfuscation technique resilient to such attacks.

3.1 Rationale

This chapter focuses on *control-flow obfuscation*, which increases software complexity (*e.g.,* by adding bogus control flows) against reverse engineering. *Opaque predicates* are essential gadgets to achieve such obfuscation transformation. An opaque predicate is a predicate whose value is known before obfuscation time but difficult to be deduced by reverse analysis. Because it holds some deterministic properties, we can employ opaque predicates to transform a program without changing its semantics. For example, we can add a bogus code block after a constantly false opaque predicate and guarantee the code block would never be executed. In practice, *opaque constant* (*e.g.,* $x^2 \neq -1$) is the most prevalent type of opaque predicates adopted by obfuscation tools, such as Obfuscator-LLVM [93]. Although other approaches (*e.g.,* unsolved conjectures [176]) may demonstrate better security, they are not widely adopted due to either implementation or performance issues [151].

Recently, the security of opaque predicates has been greatly challenged due to the development of symbolic execution techniques. Notably, Ming *et al.* have proposed an opaque predicate detection approach based on symbolic execution [122]; Yadegari *et al.* have demonstrated the effectiveness of deobfuscation attacks based on symbolic execution [188]. Symbolic execution is a program analysis approach that models the conditions for executing alternative control flows. It attempts to find test cases that can satisfy such conditions. If a condition cannot be satisfied, it may indicate a bogus control flow or an opaque predicate. Symbolic execution-based attacks may not be new to the research community. But due to the development of symbolic execution techniques, such attacks become practical recently and jeopardize the robustness of obfuscated software.

In this chapter, we propose a novel framework to manufacture *symbolic opaque predicates* which are resistant to symbolic execution-based adversaries. A key procedure in our framework is to introduce challenging problems for symbolic execution to analyze, such as employing symbolic memories and parallel executions [186]. We have conducted a systematic study on the challenges faced by symbolic execution and verified their prevalence among symbolic execution tools. Moreover, we observe a *bi-opaque property* of such opaque predicates, *i.e.,* it may either mislead an attacker into falsely recognizing an opaque predicate as a normal predicate, or to falsely recognizing a normal predicate as an opaque predicate.

We have implemented a prototype tool based on Obfuscator-LLVM [93]. Our tool automatically replaces the opaque predicates generated by Obfuscator-LLVM with symbolic opaque predicates in IR (intermediate representative) level. It employs a repository-based mechanism to manage different templates of symbolic opaque predicates. Currently, we have implemented several templates in the repository, which attack symbolic execution with symbolic memories, floating-point numbers, covert propagation, and parallel programming. The tool is flexible such that users to extend the repository with their own templates.

We have evaluated the resilience of our idea against seveal prevalent symbolic execution engines, including KLEE [26], Triton [148], and Angr [158]. The results demonstrate that symbolic opaque predicates have excellent resilience against symbolic execution-based attacks. Then we evaluate the *cost* of the implemented predicates. Experimental results show that some symbolic opaque predicates incur almost no overhead in comparison with the default opaque predicates adopted in Obfuscator-LLVM, such as those employing symbolic memories and floating-point numbers. Other opaque predicates may incur obvious execution overhead, such as those employing covert propagation and parallel programming. However, this does not degrade the usability of our framework as long as there are some efficient symbolic opaque predicates. The cost issue can be mitigated in practice by allowing users to filter inefficient predicates or to prioritize the predicates according to their preferences. Our approach is thus promising to be adopted by real-world obfuscation tools.

The rest of the chapter is organized as follows. Section 3.2 discusses our motivating examples and defines the adversary model. Section 3.3 presents the preliminary knowledge of symbolic execution. Section 3.4 summarizes the challenges faced by symbolic execution and Section 3.5 benchmarks the prevalence of the challenges with symbolic execution tools. Section 3.6 introduces our framework for composing bi-opaque predicates. Section 3.7 discusses our prototype implementation and performance evaluation results. Section 3.8 discusses the related work. Finally, Section 3.9 concludes the chapter.

```
1  int a, b;
2  ...
3  func() {
4    ...
5    if (a > b) {
6    if (a*(a+1)%2 == 0)
7      fp = A[((fp)()%2)+2];
8    else
9      fp = A[((fp)()%2)+4];
10     ...
11   }
12   ...
13   if ((b-2)*(b-1)*b%6 != 0)
14     fp = A[((fp)()%2)+5];
15   else
16     fp = A[((fp)()%2)+3];
17   ...
18 }
```

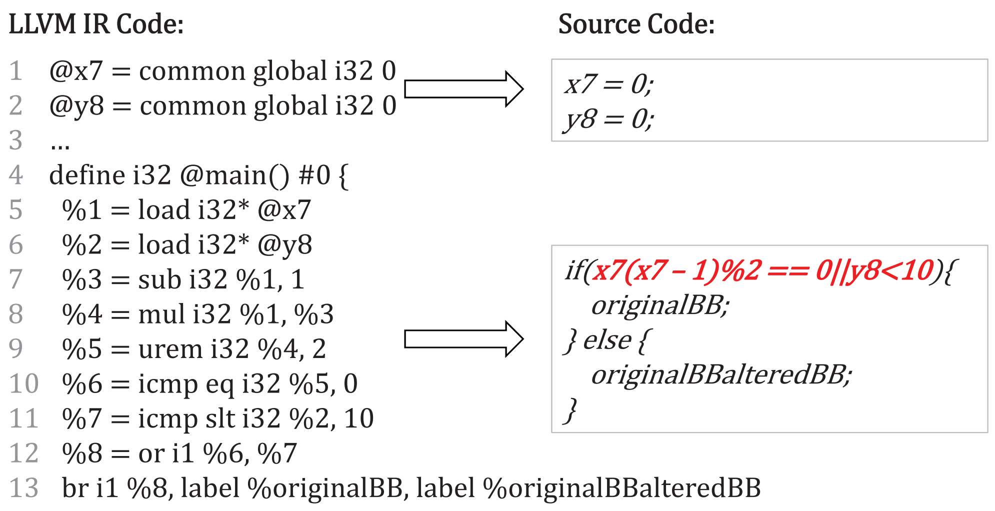
Figure 3.1: Example of vulnerable opaque predicate.

3.2 Motivation

3.2.1 Motivating Examples

Our investigation is mainly motivated by the vulnerability of real-world opaque predicates. Opaque predicates are essential gadgets for control-flow obfuscation. As stated by Collberg *et al.* [40], the security of opaque predicates largely determines the security of control-flow obfuscation. However, we notice that many real-world opaque predicates are not very strong. Below, we use two examples to demonstrate the issue.

The first example is from a highly cited paper [127], which proposes an approach to obfuscate programs with NP-hard security. To compose NP-hard problems, the authors introduce pointer analysis problems and control pointer alignments with opaque predicates. In this way, they can compose 3-SAT problems in the constraint models. However, the underlying opaque predicates in the paper are not strong enough. We demonstrate this in Figure 3.1, which includes two opaque predicates: the first one $a * (a + 1)\%2 == 0$ (line 6) is constantly true for any integer a; the second one $(b-2)*(b-1)*b\%6 \neq 0$ (line 13) is constantly false for any integer b. When such predicates are processed by a symbolic execution engine, the engine would detect that the constraints $a * (a + 1)\%2 \neq 0$ and $(b - 2) * (b - 1) * b\%6 \neq 0$ cannot be satisfied. Such predicates



```
@x7 = common global i32 0
@y8 = common global i32 0
...
define i32 @main() #0 {
 %1 = load i32* @x7
 %2 = load i32* @y8
 %3 = sub i32 %1, 1
 %4 = mul i32 %1, %3
 %5 = urem i32 %4, 2
 %6 = icmp eq i32 %5, 0
 %7 = icmp slt i32 %2, 10
 %8 = or i1 %6, %7
 br i1 %8, label %originalBB, label %originalBBalteredBB
```

```
x7 = 0;
y8 = 0;
```

```
if(x7(x7 - 1)%2 == 0||y8<10){
  originalBB;
} else {
  originalBBalteredBB;
}
```

Figure 3.2: Opaque predicate generated by Obfuscator-LLVM.

would be reported as opaque predicates by symbolic execution-based attackers. As a result, the NP-hard problem can be simplified to a polynomial-time problem.

Figure 3.2 demonstrates another opaque predicate example generated by Obfuscator-LLVM [93]. Obfuscator-LLVM is an open-source obfuscation tool for C programs and has been commercialized recently. In this example, the opaque predicate $x7 * (x7 - 1)\%2 == 0||x8 < 10$ is always true, which can be easily detected by symbolic execution techniques. We have reviewed the source code of Obfuscator-LLVM and found that the opaque predicate is the only supported one. The authors indeed have left comments in the code and stated that the opaque predicate should be improved.

Besides, there are many other investigations relying on such insecure opaque predicates, *e.g.*, [22, 124]. These examples demonstrate a severe vulnerability of current opaque predicates in practice. More resilient opaque predicates are therefore necessary to improve the security of control-flow obfuscation techniques.

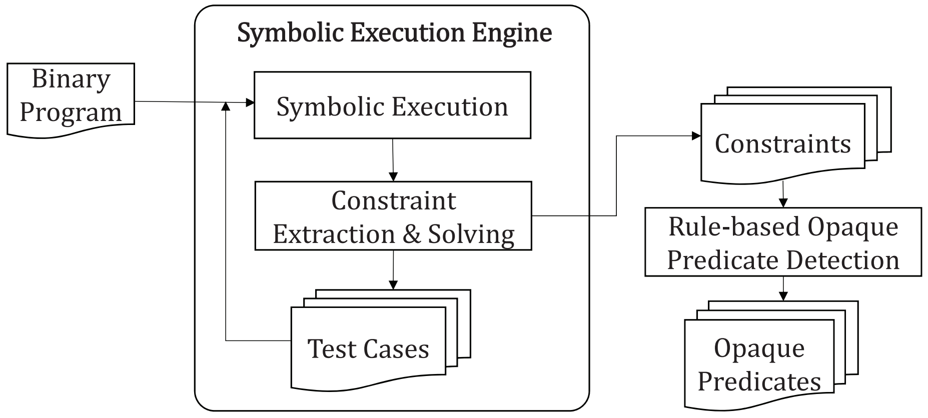

Symbolic Execution Engine
Binary Program
Symbolic Execution
Constraint Extraction & Solving
Test Cases
Constraints
Rule-based Opaque Predicate Detection
Opaque Predicates

Figure 3.3: Opaque predicate detection based on symbolic execution techniques.

3.2.2 Adversary Model

This work considers an adversary model as follows. Suppose an obfuscated program is obtained by an attacker, she can employ symbolic execution techniques to detect opaque predicates from the obfuscated program and further deobfuscate the program.

We demonstrate a framework for such opaque predicate detection attacks in Figure 3.3. Overall, a symbolic execution engine is employed to extract the conditions along control paths as constraint models; then a rule-based detection module is employed to detect opaque predicates from the constraint models.

The constraint model generated by a symbolic execution engine is generally in conjunctive normal form (CNF), *i.e.,* $\lambda_1 \wedge \lambda_2 \wedge ... \wedge \lambda_n$. Each clause λ_i represents a predicate. Then the CNF is processed according to opaque predicate detection rules, such as the rules to detect opaque constants, or contextual opaque predicates [122]. Since the rules are upper-level applications, we do not discuss their details. Instead, we focus on attacking the underlying symbolic execution engines. If the generated CNF is incorrect, it is likely that such attackers would reach false conclusions.

3.3 Preliminary Knowledge about Symbolic Execution

Before discussing our defensive method, the section reviews the underlying techniques of symbolic execution as a prelude.

3.3.1 Theoretical Basis

The core principle of symbolic execution is symbolic reasoning. Informally, given a sequence of instructions along a control path, a symbolic reasoning engine extracts a constraint model and generates a test case for the path by solving the model.

Formally, we can use Hoare Logic [87] to model the symbolic reasoning problem. Hoare Logic is composed of basic triples $\{S_1\}I\{S_2\}$, where $\{S_1\}$ and $\{S_2\}$ are the assertions of variable states and I is an instruction. The Hoare triple tells if a precondition $\{S_1\}$ is met, when executing I, it will terminate with the postcondition $\{S_2\}$. Using Hoare Logic, we can model the semantics of instructions along a control path as:

$$\{S_0\}I_0\{S_1, \Delta_1\}I_1...\{S_{n-1}, \Delta_{n-1}\}I_{n-1}\{S_n\}$$

$\{S_0\}$ is the initial symbolic state of the program; $\{S_1\}$ is the symbolic state before the first conditional branch associated with symbolic variables; Δ_i is the corresponding constraint for executing the following instructions, and $\{S_i\}$ satisfies Δ_i. A symbolic execution engine can compute an initial state $\{S_0'\}$, *i.e.,* the concrete values for symbolic variables, which can trigger the same control path. This can be achieved by computing the weakest precondition (*aka* wp) backward using Hoare Logic:

$$\{S_{n-2}\} = wp(I_{n-2}\{S_{n-1}\}), \quad s.t. \ \{S_{n-1}\} \ sat \ \Delta_{n-1}$$

$$\{S_{n-3}\} = wp(I_{n-3}\{S_{n-2}\}), \quad s.t. \ \{S_{n-2}\} \ sat \ \Delta_{n-2}$$

$$\cdots$$

$$\{S_1\} = wp(I_1\{S_2\}), \quad s.t. \ \{S_2\} \ sat \ \Delta_2$$
$$\{S_0\} = wp(I_0\{S_1\}), \quad s.t. \ \{S_1\} \ sat \ \Delta_1$$

Combining the constraints in each line, we can get a constraint model in conjunction normal form: $\Delta_1 \wedge \Delta_2 \wedge ... \wedge \Delta_{n-1}$. The solution to the constraint model is a test case $\{S_0'\}$ that can trigger the same control path.

Finally, while sampling $\{I_i\}$, not all instructions may be found to be useful. We only keep the instructions whose parameter values depend on the symbolic variables. We can demonstrate the correctness by expending any irrelevant instruction I_i to $X := E$, which manipulates the value of a variable X with an expression E. If E does not depend on any symbolic value, X would be a constant, and should not be included in the weakest preconditions. In practice, it can be realized by symbolic execution tools (*e.g.,* Mayhem [29] and FuzzBALL [120]) using taint analysis techniques [152].

3.3.2 Symbolic Execution Framework

Figure 3.4 demonstrates the conceptual framework of a symbolic execution tool. It involves inputting a program and outputting test cases for the program. The framework includes a core symbolic reasoning engine and a path selection engine.

The symbolic reasoning engine analyzes the instructions along a path and generates test cases that can trigger the path. Based on the symbolic reasoning, we can identify four stages: symbolic variable declaration, instruction tracing, semantic interpretation, and constraint modeling and solving. The details are as follows:

- *Symbolic variable declaration* (S_{var}): In this stage, we have to declare symbolic variables which will be employed in the following symbolic analysis process. If some symbolic variables are missing from declaration, insufficient constraints can be generated for triggering a control path.

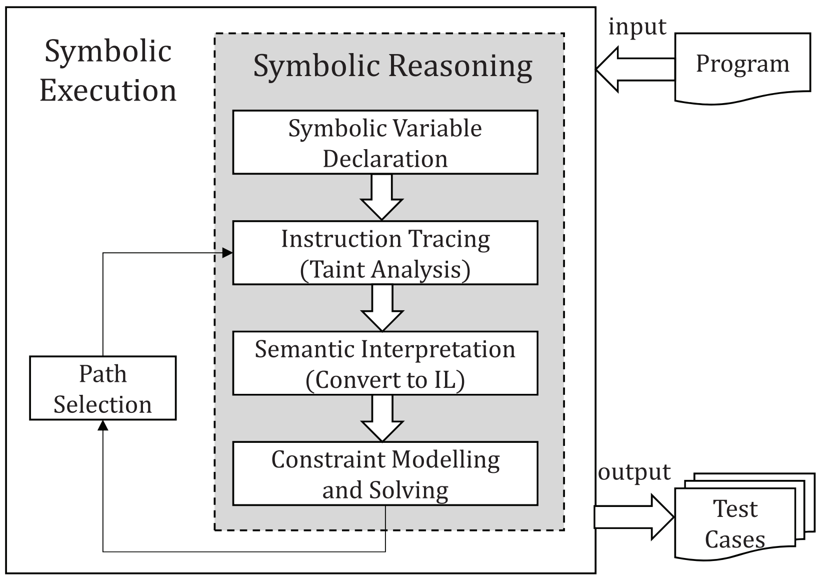

Symbolic Execution
Symbolic Reasoning
Symbolic Variable Declaration
Instruction Tracing (Taint Analysis)
Semantic Interpretation (Convert to IL)
Constraint Modelling and Solving
Path Selection
input
Program
output
Test Cases

Figure 3.4: Conceptual framework for symbolic execution.

- *Instruction tracing* (S_{inst}): This stage collects the instructions along control paths. If some instructions are missing, or the syntax is not supported, the symbolic reasoning process would be inconsistent.

- *Semantic interpretation* (S_{sem}): This stage translates the semantics of collected instructions with an intermediate language (IL). If some instructions are incorrectly interpreted, or the data propagation are incorrectly modeled, the symbolic execution engine would generate inconsistent constraint models consequently.

- *Constraint modeling and solving* (S_{model}): This stage generates constraint models from IL, and then solves them. If the required satisfiability modulo theory is unsupported, errors are likely.

The path selection engine determines which path should be analyzed in the next round of symbolic reasoning. The favorited strategies include depth-first search, width-first search, random search, *etc*. [8].

3.3.3 Implementation Variations

According to the different ways of instruction tracing, we can classify symbolic execution tools into static symbolic execution (*e.g.,* KLEE [26, 163]) and dynamic symbolic execution (*e.g.,* Triton [148]). Static symbolic execution loads a whole program first before extracting instructions along with a path on the program control-flow graph (CFG). Dynamic symbolic execution is also known as concolic (concrete and symbolic) execution. It collects instructions which have been actually executed. In each round, the concolic execution engine executes the program with concrete values to generate instructions [181].

We may also classify symbolic execution tools into source-code-based symbolic execution and binary-code-based symbolic execution. In general, we do not perform symbolic reasoning on source codes or binaries directly. A prior step is to interpret the semantics of the program with an intermediate language (IL). Therefore, the main difference between the two implementation methods lies in the translation process. Regarding source codes, we can translate the code directly with the compiler's frontend. As for binaries, we have to lift the assembly codes into IL, which is error-prone due to the complicated features of modern CPUs [97]. The lifting process is challenging and remains as an active research area.

3.4 Challenges of Symbolic Execution

Based on whether a challenge is associated with the symbolic reasoning process, we can categorize the challenges of symbolic execution into *symbolic-reasoning challenges* and *path-explosion challenges*. A symbolic-reasoning challenge attacks the symbolic

Table 3.1: List of challenges faced by symbolic execution.

Challenge		**Stage of Error**		
		S_{var}	S_{inst} & S_{sem}	S_{model}
Symbolic -reasoning Challenges	Sym. Var. Declaration	✓	✓	✓
	Covert Propagations	-	✓	✓
	Buffer Overflows	-	✓	✓
	Parallel Executions	-	✓	✓
	Symbolic Memories	-	✓	✓
	Contextual Symbolic Values	-	✓	✓
	Symbolic Jumps	-	-	✓
	Floating-point Numbers	-	-	✓
	Arithmetic Overflows	-	-	✓
Path-explosion Challenges	Loops	-	-	-
	Crypto Functions	-	-	-
	External Function Calls	-	-	-

reasoning process and leads to incorrect test cases being generated. A path-explosion challenge happens when there are too many paths to analyze. It does not attack a single symbolic reasoning process, but may get starved of computational resources or require a very long time for symbolic execution.

Table 3.1 lists the challenges that we have investigated in this work. We collected the challenges via a careful survey of existing papers. The survey coverred several survey papers realated to symbolic execution techniques (*e.g.,* [8, 27, 152]), several investigations that focus on systemizing the challenges of symbolic execution (*e.g.,* [46, 94]), and other important papers related to symbolic execution (*e.g.,* [24, 29, 48, 74, 78, 140, 181, 188]).

3.4.1 Symbolic-Reasoning Challenges

We now discuss nine challenges that may incur errors to symbolic reasoning.

```
int logic_bomb() {
  int pid = (int) getpid();
  printf ("current pid is %d\n%", pid);
  if(pid == 4096)
    return BOMB_ENDING;
  else
    return NORMAL_ENDING;
}
```

(a) Symbolic variable declarations.

```
int logic_bomb(char* symvar) {
  int flag = 0;
  char buf[8];
  strcpy(buf, symvar);
  if(flag == 1){
    return BOMB_ENDING;
  }
  return NORMAL_ENDING;
}
```

(b) Buffer overflows.

```
char* shell(const char* cmd){
  char* ret = "";
  FILE *f = popen(cmd, "r");
  char buf[1024];
  memset(buf,'\0',sizeof(buf));
  while(fgets(buf,1024-1,f)!=NULL)
    ret = buf;
  pclose(f);
  return ret;
}
```

```
int logic_bomb(char* symvar) {
  int i=symvar[0]-48;
  char cmd[256];
  sprintf(cmd, "echo %d\n", i);
  char* ret = shell(cmd);
  if(atoi(ret) == 7){
    return BOMB_ENDING;
  }
  return NORMAL_ENDING;
}
```

(c) Covert symbolic propagations.

```
int threadprop(int in){
  pthread_t tid[2];
  int rc1 = pthread_create(&tid[0], NULL, Inc, (void *) &in);
  int rc2 = pthread_create(&tid[1], NULL, Mult, (void *) &in);
  rc1 = pthread_join(tid[0], NULL);
  rc2 = pthread_join(tid[1], NULL);
  int out = in;
  return out;
}
```

```
int logic_bomb(char* symvar) {
  int i=symvar[0]-48;
  int j=threadprop(i);
  if(j == 50){
    return  BOMB_ENDING;
  }
  return  NORMAL_ENDING;
}
```

(d) Parallel executions.

Figure 3.5: Sample logic bombs with symbolic-reasoning challenges: part I.

Symbolic Variable Declarations

Since the test cases are the solutions of symbolic variables subject to constrain models, symbolic variables should be declared before a symbolic reasoning process. For example, in source-code-based symbolic execution tools (*e.g.,* KLEE [26]), users can manually declare symbolic variables in the source codes. Binary-code-based concolic execution tools (*e.g.,* Triton [148]) generally assume a fixed length of program arguments from stdin as the symbolic variable. If some symbolic variables are missing from the declaration, the generated test cases would be insufficient for triggering particular control paths. Since the root cause occurs before symbolic execution, the challenge attacks S_{var}.

Figure 3.5(a) is a sample with a symbolic variable declaration problem. It returns a `BOMB_ENDING` only when being executed with a particular process id. To explore the path, a symbolic execution tool should treat `pid` as a symbolic variable and then solve the constraint with respect to `pid`. Otherwise, it cannot find test cases that can trigger the path.

To declare symbolic variables precisely, a user should know target programs well. However, the task is impossible when analyzing programs on a large scale, *e.g.,* when performing malware analysis. In an ideal case, a symbolic execution tool may automatically detect such variables which can control program behaviors and report the solutions accordingly. To our best knowledge, very few tools have implemented the ideal feature, except DART [74]. Instead, present papers (*e.g.,* [8, 35]) generally discuss the challenge together with other problems related to the computing environment, such as libraries, kernels, and drivers. In reality, there are several challenges of this work referring to the computing environment, such as contextual symbolic variables, covert propagations, parallel executions, external function calls. We demonstrate that these challenges are different.

Buffer Overflows

Buffer overflow is a typical software bug that can bring security issues. Due to insufficient boundary checking, the input data may overwrite adjacent memories. Adversaries can employ such bugs to inject data and intentionally and tamper with the semantics of the original codes. Buffer overflows can happen in either stack or heap regions. If a symbolic execution tool cannot detect the overflow issues arising, it would fail to track the propagation of symbolic values. Therefore, buffer overflow involves a particular covert propagation issue. Source-code-based symbolic execution tools are prone to buffer overflows because the stack layout of a program exists only in the assembly codes, depending on the particular platforms. Therefore, such tools cannot model stack information using source codes only. In contrast, binary-code-based symbolic execution tools should be more potent in handling buffer overflow issues because they can simulate actual memory layouts. However, even if these tools can precisely track propagation, they suffer from difficulties in automatically analyzing the unexpected program behaviors caused by overflow. Otherwise, they would be powerful enough to generate exploits for bugs, which is a problem still requiring solution [4].

Figure 3.5(b) presents an example of buffer overflows. The program returns a `BOMB_ENDING` if the value of `flag` equals one, which is unlikely because the value is zero and should remain unchanged without explicit modification. However, the program has a buffer overflow bug. It has a buffer `buf` of eight bytes and employs no boundary check when copying symbolic values to the buffer with `strcpy`. We can change the value of `flag` to one leveraging the bug, *e.g.,* when `symvar` is “ANYSTRIN\x01\x00\x00\x00”.

Covert Propagations

Some data propagation ways are covert because they cannot be traced easily by data-flow analysis tools. For example, if the symbolic values are propagated via other media (*e.g.,* files) outside of the process memory, the propagation would be untraceable. Such propagation methods are undecidable and can be beyond the capability of pure program analysis. Symbolic execution tools have to handle such cases using ad hoc methods. There are also some propagations challenging only to certain implementations. For example, propagating symbolic values via embedded assembly codes can be a problem for source-code-based symbolic execution tools only. If a symbolic execution tool fails to detect certain propagations, the instructions related to the propagated values would be missed from the following analysis. This results in the challenge attacking the stages of S_{inst} and S_{sem}.

Figure 3.5(c) shows a covert propagation sample. We define an integer `i` and initiate it with the value of a symbolic variable `symvar`. So `i` is also a symbolic variable. We then propagate the value of `i` to another variable `ret` through a shell command `echo`, and let `ret` control the return value. To find a test case which can return the corresponding `BOMB_ENDING`, a symbolic execution tool should properly track or model the propagation incurred by the shell command.

Parallel Executions

Classic symbolic execution is effective for sequential programs. We can draw an explicit CFG for sequential programs and let a symbolic execution engine traverse the CFG. However, if the program processes symbolic variables in parallel, classic symbolic execution techniques would face problems. Parallel programs can be undecidable because the execution order of parallel codes does not only depend on the program but may also depend on the execution context. A parallel program may exhibit different behaviors even with the same test case. This poses a problem for symbolic execution to generate test cases for triggering corresponding control flows. If a symbolic execution tool directly ignores the parallel syntax or addresses the syntax improperly, errors would happen during S_{inst} and S_{sem}.

Figure 3.5(d) demonstrates an example with parallel codes. The symbolic variable `i` is processed by another two additional threads in parallel, and the result is assigned to `j`. Then the value of `j` determines whether the program should return a `BOMB_ENDING`.

To handle parallel codes, the symbolic execution tool has to interpret the semantics and track parallel executions, *e.g.,* by introducing extra symbolic variables [62]. However, such an approach may not

```
int logic_bomb(char* symvar) {
   int i=symvar[0]-48;
   int array[] ={1,2,3,4,5};
   if(array[i%5] == 5){
      return BOMB_ENDING;
   }
   else
      return NORMAL_ENDING;
}
```

(a) Symbolic memories.

```
int f0() {return 0;}   ...   int f6() {return 6;}
int logic_bomb(char* symvar) {
   int (*func[7])() = {f0, f1, f2, f3, f4, f5, f6};
   int ret = func[(symvar[0] - 48)%7]();
   if(ret == 5){
         return BOMB_ENDING;
   }
   return NORMAL_ENDING;
}
```

(b) Symbolic jumps.

```
int logic_bomb(char* symvar) {
   FILE *fp = fopen(symvar, "r");
   if(fp != NULL){
      fclose(fp);
      return BOMB_ENDING;
   }else{
      return NORMAL_ENDING;
   }
}
```

(c) Contextual symbolic values.

```
int logic_bomb(char* symvar) {
   int i=symvar[0]-48;
   float a = i/70.0;
   float b = 0.1;
   if(a != 0.1 & a - b == 0){
         return BOMB_ENDING;
   }
   return NORMAL_ENDING;
}
```

(d) Floating-point numbers.

```
int logic_bomb(char* symvar) {
    int i = symvar[0] - 48;
    if (254748364 * i < 0 && i > 0){
       return BOMB_ENDING;
    }
    return NORMAL_ENDING;
}
```

(e) Arithmetic overflows.

Figure 3.6: Sample logic bombs with symbolic-reasoning challenges: part II.

be scalable because the possibility of parallel execution can be a large number. In practice, there are several heuristic approaches that can be used to improve the efficiency. For example, we may restrict the exploration time of concurrent regions with a threshold [62]; we may conduct symbolic execution with arbitrary contexts and convert multi-thread programs into equivalent sequential ones [16]; or we can prune unimportant paths leveraging some program codes, such as assertion [79].

Symbolic Memories

Symbolic memory is a situation whereas symbolic variables serve as the offsets or pointers to retrieve values from the memory, such as array indexes. While handling symbolic memories, the symbolic execution engine should take advantage of the memory layout for analysis. For example, we can convert an array selection operation to a `switch/case` clause in which the number of possible cases equals the length of the array. However, the number of possible combinations would grow exponentially when there are several such operations along a control flow. In practice, a symbolic execution tool may directly employ the feature of array operations implemented by some constraint solvers, such as STP [68] and Z3 [49]. It may also analyze the alignment of some pointers in advance, such as CUTE [154]. However, the power of pointer analysis is limited because the problem can be NP-hard or even undecidable for static analysis [102]. If a symbolic execution tool cannot model symbolic memories properly, errors would occur during S_{inst} and S_{sem}.

Figure 3.6(a) presents a sample of symbolic memories. In this example, the symbolic variable `i` serves as an offset to retrieve an element from the array. The retrieved element then determines whether the program returns a `BOMB_ENDING`.

Symbolic Jumps

In general, symbolic execution only extracts constraint models when encountering conditional jumps, such as `var<0` in source codes, or `jle 0x400fda` in assembly codes. However, we may also employ unconditional jumps to achieve the same effects as conditional jumps. The idea is to jump to an address controlled by symbolic values. If a symbolic execution engine is not tailored to handle such unconditional jumps, it would fail to extract corresponding constraint models and miss some available control flows. Therefore, the challenge attacks the constraint modeling stage S_{model}.

Figure 3.6(b) presents an example of symbolic jumps. The program contains an array of function pointers, and each function returns an integer value. The symbolic variable serves as an offset to determine which function should be called during execution. If `f5()` is called, the program would return a `BOMB_ENDING`.

Contextual Symbolic Values

This challenge is similar to symbolic memories but is more complicated. Other than retrieving values from the memory like symbolic memories, symbolic values can also serve as the parameters to retrieve values from the environment, such as loading the contents of a file pointed by symbolic values. By default, this contextual information is unavailable to the program or process, and the analysis is more complicated. Moreover, since the contextual information can be changed any time without informing the program, the problem is undecidable. A symbolic tool that does not support such operations would cause errors during S_{inst} and S_{sem}.

Figure 3.6(c) is an example of contextual symbolic values. If `symvar` points to an existing file on the local disk, the program returns a `BOMB_ENDING`.

Floating-Point Numbers

A floating-point number ($f \in \mathbb{F}$) approximates a real number ($r \in \mathbb{R}$) with a fixed number of digits in the form of $f = sign \times base^{exp}$. For example, the 32-bit float type compliant to IEEE-754 has 1-bit for $sign$, 23-bit for $base$, and 8-bit for exp. This representation is essential for computers, as the memory spaces are limited in comparison with the infinity of $\mathbb{R}$. As a tradeoff, floating-point numbers have limited precision, which turns some unsatisfiable constraints over $\mathbb{R}$ into satisfiable ones over $\mathbb{F}$ with a rounding mode. In order to support reasoning over $\mathbb{F}$, a symbolic execution tool should consider such approximations when extracting and solving constraint models. However, recent studies (*e.g.,* [161, 139, 110, 111]) show that there is still no silver bullet for the problem. Floating-point numbers continue to pose a challenge for symbolic execution tools, and the challenge attacks S_{model}.

Figure 3.6(d) demonstrates an example with floating-point operations. Because we cannot represent 0.1 with float type precisely, the first predicate `a != 1` is always true. If the second condition `a == b` can be satisfied, the program would return a `BOMB_ENDING`. Therefore, one test case to returning a `BOMB_ENDING` is `symvar` equals `'7'`.

Arithmetic Overflows

Arithmetic overflow happens when the result of an arithmetic operation is outside the range of an integer type. For example, the range of a 64-bit signed integer is $[-2^{64}, 2^{64} - 1]$. In this case, a constraint model (*e.g.,* the result of a positive integer plus another positive integer is negative) may have no solutions over $\mathbb{R}$; but it can have solutions when we consider arithmetic overflow. Handling such arithmetic overflow issues is not as difficult as in the case of the previous challenges. However, some preliminary symbolic execution tools may fail to consider these cases and suffer errors when extracting and solving the constraint models.

Figure 3.6(e) shows a sample with an arithmetic overflow problem. To meet the first condition `254748364 * i < 0`, `i` should be a negative value. However, the second condition requires `i` to be a positive value. Therefore, it has no solutions in the domain of real numbers. But the conditions can be satisfied when `254748364 * i` exceeds the max value that the integer type can represent.

3.4.2 Path-Explosion Challenges

Now we discuss three path-explosion challenges existing in small-size programs.

External Function Calls

Shared libraries, such as `libc` and `libm` (*i.e.,* a maths library), provide some basic function implementations to facilitate software development. An efficient way to employ the functions is via

```
int logic_bomb(char* symvar) {
    int i = symvar[0] - 48;
    float v = sin(i * PI / 30);
    if(v > 0.5){
        return BOMB_ENDING;
    }
    return NORMAL_ENDING;
}
```

(a) External function calls.

```
int f(int x){
    if (x % 2 == 0)
        return x / 2;
    return 3 * x + 1;
}
int logic_bomb(char* symvar) {
    int i = symvar[0]-48+94;
    int j = f(i);
    int loopcount = 1;
    while(j != 1){
        j = f(j);
        loopcount ++;
    }
    if(loopcount == 25)
        return BOMB_ENDING;
    else
        return NORMAL_ENDING;
}
```

(b) Loops.

```
int logic_bomb(char* symvar) {
    int plaintext = symvar[0] - 48;
    unsigned cipher[5];
    cipher[0] = 0X77de68da;
    cipher[1] = 0Xecd823ba;
    cipher[2] = 0Xbbb58edb;
    cipher[3] = 0X1c8e14d7;
    cipher[4] = 0X106e83bb;
    if(SHA1_COMP(plaintext,cipher)==0){
        return BOMB_ENDING;
    }else{
        return NORMAL_ENDING;
    }
}
```

(c) Crypto functions.

Figure 3.7: Sample bomb samples with path-explosion challenges.

dynamic linkage, which does not pack the function body to the program but only links with the functions dynamically during execution. Therefore, such external functions do not enlarge the size of a program; they just enlarge code complexity.

When an external function call is related to the propagation of symbolic values, the control flows within the function body should be analyzed by default. There are two situations. A simple situation is that the external function does not affect the program behaviors after executing it, such as simply printing symbolic values with `printf`. In this case, we may ignore the path alternatives within the function. However, if the function execution affects the follow-up program behaviors, we should not ignore them. Otherwise, the symbolic execution would be based on the wrong assumption that the new test case generated for an alternative path can always trigger the same control flow within the external function. If a small program contains several such function calls, the complexity of external functions may cause path explosion issues. In practice, there are different strategies (*e.g.,* abstraction [26], strict consistency and local consistency [35]) that symbolic execution tools may adopt to handle the challenge with a trade-off between consistency and efficiency.

Figure 3.7(a) demonstrates a sample with an external function call. It computes the sine of a symbolic variable via an external function call (*i.e.,* `sin`), and the result is used to determine whether the program should return a `BOMB_ENDING`.

Loops

Loop statements, such as `for` and `while`, are widely employed in real-world programs. Even a very small program with loops can include many or even an infinite number of paths. By default, a symbolic execution tool should explore all available paths of a program, which can be beyond the capability of the tool if there are too many paths. In practice, a symbolic execution tool may employ a search strategy favoring unexplored branches on a program CFG [5, 25], or introduce new symbolic variables as the counters for each loop [149]. Because loop can incur numerous paths, we can hardly have a perfect solution for this problem.

Figure 3.7(b) shows a sample with a loop. The loop function is implemented with the Collaz conjecture [101]. No matter what is the initial value of `i`, the loop will terminate with `j` equals 1.

Crypto Functions

Crypto functions generally involve some computationally complex problems to ensure security. For a hash function, the complexity guarantees that adversaries cannot efficiently compute the plaintext of a hash value. For a symmetric encryption function, it promises that one cannot efficiently compute the key when given several pairs of plaintext and ciphertext. Therefore, such programs should also be resistant to symbolic execution attacks. From a program analysis view, the number of possible control paths for the crypto functions can be substantial. For example, the body of the SHA1 algorithm is a loop that iterates 80 rounds with each round containing several bit-level operations.

Figure 3.7(c) demonstrates a code snippet which employs a SHA1 function. If the hash result of the symbolic value is equivalent to a predefined value, the program would return a `BOMB_ENDING`. However, this is difficult since SHA1 cannot be reversely calculated.

In general, symbolic execution tools cannot handle such crypto programs. Malware may employ the technique to deter symbolic execution-based program analysis [156]. When analyzing programs with crypto functions, a common way is to avoid exploring the function internals (*e.g.,*[43, 175]). For example, TaintScope [175] first discriminates the symbolic variables corresponding to crypto functions from other variables, and then employs a fuzzy-based approach to search solutions for such symbolic variables rather than solving the problem via symbolic reasoning.

So far, we have discussed 12 different challenges. Note that we do not intend to propose a complete list of challenges for symbolic execution. Instead, we collect all the challenging issues that have been mentioned in the literature and systematically analyze them. This analysis is essential while designing the dataset of logic bombs in Section 3.5.2.

3.5 Benchmarking Symbolic Execution Tools

In this section, we introduce our methodology and a framework to benchmark the capability of real-world symbolic execution tools.

3.5.1 Objective and Challenges

Before describing our approach, we first discuss our design goal and the challenges to overcome.

We aim to design an approach that can benchmark the capabilities of symbolic execution tools. Our purpose is critical and valid in several aspects. As we have discussed, some challenging issues are only engineering issues, such as arithmetic overflows. With enough engineering effort, a symbolic execution tool should be able to handle these issues. On the other hand, some challenges such as loops are hard from a theoretical viewpoint. However, some heuristic approaches can tackle certain easy cases. Symbolic execution tools may adopt different heuristics and demonstrate different capabilities in handling them. Therefore, it is worth benchmarking their performances in handling particular challenging issues. Developers generally do not provide much information concerning the limitations of their tools to users.

A useful benchmarking approach should be accurate and efficient. However, this is challenging to benchmark symbolic execution tools accurately and efficiently with real-world programs. Firstly, a real-world program contains many instructions or lines of codes. When a symbolic execution failure occurs, locating the root cause requires much domain knowledge and effort. Since errors may propagate, it is challenging to conjecture whether a symbolic execution tool fails in handling a particular issue. Secondly, the symbolic execution itself is inefficient. Benchmarking a symbolic execution tool generally implies performing several designated symbolic execution tasks, which would be time-consuming. Note that existing symbolic execution papers (*e.g.,* [26, 83, 99, 158]) generally evaluate the performance of their tools by conducting symbolic execution experiments with real programs. This process usually takes several hours or even days. They demonstrate the effectiveness of their work using the achieved code coverage and number of bugs detected, but they do not analyze the root causes of uncovered codes in detail.

Algorithm 1: Method to design evaluation samples.

```
// Create a function with a symbolic variable
LogicBomb($symvar$) // $symvar2$ is a value computed from a
    challenging problem related to $symvar$
$symvar2 \leftarrow$ Challenge($symvar$);
// If $symvar2$ satisfies a condition
if Condition($symvar2$) then
    // Trigger the bomb
    Bomb();
end
```

3.5.2 Approach based on Logic Bombs

To tackle the challenges of benchmarking symbolic execution tools concerning accuracy and efficiency, we propose an approach based on logic bombs. Below, we discuss our detailed design.

Evaluation with Logic Bombs

A logic bomb is a code snippet that can only be executed when certain conditions have been met. To evaluate whether a symbolic execution tool can handle a challenge, we can design a logic bomb guarded by a particular issue with the challenge. Then we can perform symbolic execution on the program embedded with the logic bomb. If a symbolic execution tool can generate a test case that can trigger the logic bomb, it indicates that the tool can handle the challenging issue, or *vice versa*.

Algorithm 1 demonstrates a general framework for designing such logic bombs. It includes four steps: the first step is to create a function with a parameter $symvar$ as the symbolic variable; the second step is to design a challenging problem related to the symbolic variable and save the result to another variable $symvar2$; the third step is to design a condition related to the new variable $symvar2$; the final step is to design a bomb (*e.g.,* return a specific value) which indicates that the condition has been satisfied. Note that because the value of $symvar2$ is propagated from $symvar$,

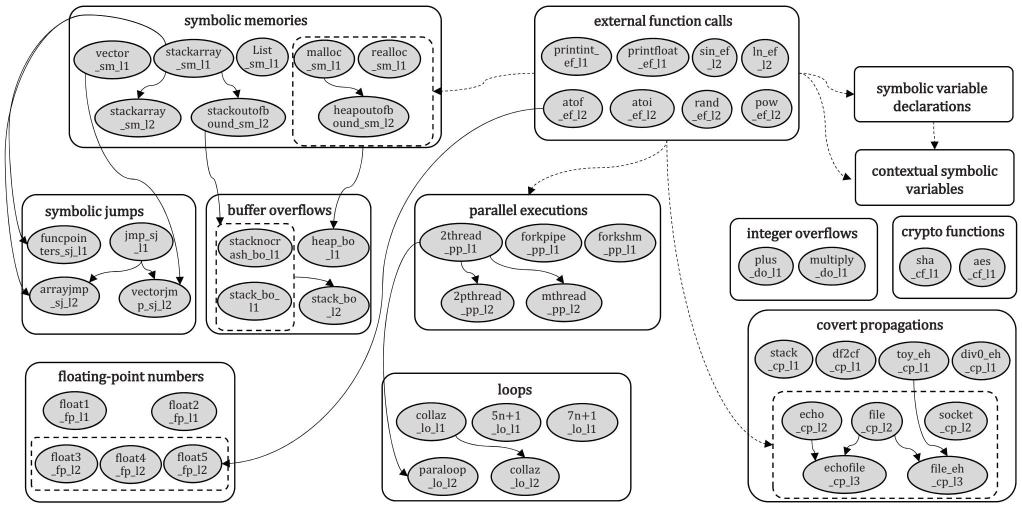


Figure 3.8: Dataset of logic bombs and the challenge propagation relationships among them.

$symvar2$ is also a symbolic variable and should be considered in the symbolic analysis process.

The magic of the logic bomb idea enables us to make the evaluation much precise and efficient. We can create several such small programs, each containing only a challenging issue and a logic bomb that tells the evaluation result. Because the object programs for symbolic execution are usually small, we can easily avoid unexpected issues that may also cause failures via a careful design. Also, because the programs are small, performing symbolic execution on them generally requires a short time. For the programs that unavoidably incur path explosion issues, we can restrict the symbolic execution time either by controlling the problem complexity or by employing a timeout setting.

Logic Bomb Dataset

Following Algorithm 1, we have designed a dataset of logic bombs to evaluate the capability of symbolic execution tools. Some of the logic bombs are already shown in Figure 3.5, 3.6, and 3.7. Our full dataset is available on GitHub[1]. The dataset contains over 60 logic bombs for 64-bit Linux platform, which covers all the challenges discussed in Section 3.4. For each challenge, we implement several logic bombs. Either each bomb involves a unique challenging issue (*e.g.,* covert propagation via file write/read or via system calls), or introduces a problem with a different complexity setting (*e.g.,* one-leveled arrays or two-leveled arrays).

When designing logic bombs, we carefully avoid trivial test cases (*e.g.,* `\x00`) that can trigger the bombs. Moreover, we try to employ straightforward implementations, and we hope to ensure that the results would not be affected by other unexpected failures. For example, we avoid using `atoi` to convert `argv[1]` to integers because some tools cannot support `atoi`. However, fully avoiding external function calls is impossible for some logic bombs. For example, we should employ external function calls to create threads when designing parallel codes. Surely the result might be affected if a symbolic execution tool cannot handle external functions. To tackle the interference of challenges, we draw a challenge propagation chart among the logic bombs as shown in Figure 3.8. There are two kinds of challenge propagation relationships: $should$ in solid lines, and may in dashed lines. A $should$ relationship means that a logic bomb contains a similar challenging issue in another logic bomb; if a tool cannot solve the precedent logic bomb, it should not be able to solve the later one. For example, the `stackarray_sm_l1` is precedent to `stackarray_sm_l2`. A may relationship means a challenge type may be a precedent to other logic bombs, but it is not the determining one. For example, a parallel program generally involves external function calls. However, although a tool is unable to solve the external functions well, it might be able to solve some logic bombs with parallel issues as sequential programs.

[1]https://github.com/hxuhack/logic_bombs

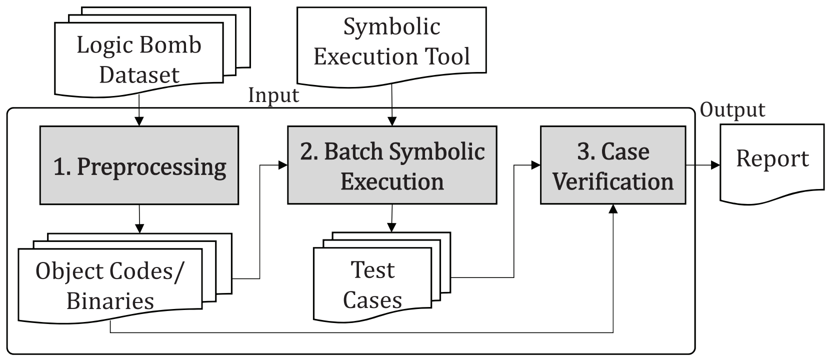


Figure 3.9: Framework to benchmark symbolic execution tools.

3.5.3 Automated Benchmarking Framework

Based on the evaluation idea with logic bombs, we design a benchmarking framework as shown in Figure 3.9. The framework inputs a dataset of carefully designed logic bombs and outputs the benchmarking result for a particular symbolic execution tool. There are three critical steps in the framework: dataset preprocessing, batch symbolic execution, and case verification.

In the preprocessing step, we parse the logic bombs and compile them into object codes or binaries such that a target symbolic execution tool can process them. The parsing process pads each code snippet of a logic bomb with a main function and makes it a self-contained program. By default, we employ `argv[1]` as the symbolic variables. If a target symbolic execution tool requires adding extra instructions to launch tasks, the parser should add such required instructions automatically. For example, we can add symbolic variable declaration codes when benchmarking KLEE. The compilation process compiles the processed source codes into binaries or other formats that a target symbolic execution tool supports. Symbolic execution is generally performed based on intermediate codes. When benchmarking source-code-based symbolic execution tools such as KLEE, we have to compile the source codes into the supported intermediate codes. When benchmarking binary-code-based symbolic execution tools, we can directly compile them into binaries, and the tool will lift binary codes into intermediate codes automatically.

In the second step, we direct the symbolic execution tool to analyze the compiled logic bombs in a batch mode. This step outputs a set of test cases for each program. Some dynamic symbolic execution tools (*e.g.,* Triton) can directly tell which test case can trigger a logic bomb during runtime. However, other static symbolic execution tools may only output test cases by default, so we need to replay the generated test cases to examine the results further. Besides, some tools may falsely report that a test case can trigger the logic bomb. Therefore, we need a third step to verify the test cases.

In the third step, we replay the test cases with the corresponding programs of logic bombs. If a logic bomb can be triggered, it indicates that the challenging case has been solved by the tool. Finally, we can generate a benchmarking report based on the case verification results.

3.5.4 Benchmarking Results

Experimental Setting

We choose three popular symbolic execution tools for benchmarking: KLEE [26], Angr [158], and Triton [148]. Because our dataset of logic bombs are written in C/C++, we only choose symbolic execution tools for C/C++ programs or binaries. The three tools have all been released as open source and have a high community impact. Moreover, they adopt different implementation techniques for symbolic execution. By supporting variant tools, we show that our approach is compatible with different symbolic execution implementations.

KLEE[26] is a source-code-based symbolic execution tool implemented based on LLVM [106]. It supports programs written in C. By default, our benchmarking script uses a `klee_make_symbolic` function to declare the symbolic variables of logic bombs in the source-code level. Then, it compiles the source codes into intermediate codes for symbolic execution. The symbolic execution process outputs a set of test cases. Our script finally examines the test cases by replaying them with the binaries. The whole process is automated with our benchmarking script. The version of KLEE we benchmark is 1.3.0. Note that because our experiment does not intend to find the best tool for particular challenges, we do not consider the patches provided by other parties before they are merged into the master branch.

Triton [148] is a dynamic symbolic execution tool based on binaries. It automatically accepts symbolic variables from the standard input. During symbolic execution, it firstly runs the programs with concrete values and leverages Intel PinTool [115] to trace related instructions; then it lifts the traced instructions into the SSA (single static assignment) form and performs symbolic analysis. If there are alternative paths found in the trace, Triton generates new test cases via symbolic reasoning and employs them as the concrete values in the following rounds of concrete execution. This symbolic execution process continues until no alternative path can be found. The version of Triton we adopted is the one released on GitHub on Jul 6, 2017.

Angr [158] is also a tool for binaries but employs different implementations. Before performing any symbolic analysis, Angr firstly lifts the binary program into VEX IR [125]. Then it employs a symbolic analysis engine (SimuVEX) to analyze the program based on the IR. Angr does not provide ready-to-use symbolic execution script for users but only some APIs. Therefore, we have to implement our own symbolic execution script for Angr. Our script collects all the paths to the CFG leaf nodes and then solves the corresponding path constraints. Angr provides all the critical features via APIs, and we only assemble them. Finally, we check whether the generated test cases can trigger the logic bombs. In our experiment, we employ Angr version 7.7.9.21.

Note that all our benchmarking scripts for these tools follow the framework proposed in Figure 3.9. During the experiments, we employ our logic bomb dataset for evaluation. A tool can pass a test only if the solution generated can correctly trigger a logic bomb. We finally report which logic bombs can be triggered by the tools.

We conduct our experiments on an Ubuntu 14.04 X86_64 system with Intel i5 CPU and 8G RAM. Because some symbolic execution tasks may take very long time, our tool allows users to configure a timeout threshold which ensures benchmarking efficiency. However, the timeout mechanism may incur some false results if it is too short. To mitigate the side effects, we adopt two timeout settings (60 seconds and 300 seconds) for each tool. In this way, we can observe the influence of the timeout settings and decide whether we should conduct more experiments with an increased timeout value.

Result Overview

Table 3.2 presents our experimental results. We label the results with four options: pass, fail, timeout, and inapplicable. While 'pass' and 'fail' imply the symbolic execution has finished, 'timeout' implies our benchmarking script has terminated the symbolic execution process when a timeout threshold is triggered. We label several results as inapplicable because they contain C++ or assembly codes, which KLEE does not support.

We can observe that Angr has achieved the best performance with 22 cases solved when the timeout was 300 seconds. Comparatively, it only solved 16 cases when the timeout is 60 seconds. KLEE solved nine cases and the result remains the same with different timeout settings. Triton performed much worse with just three cases being solved. To further verify the correctness of our benchmarking results, we compared our experimental results with the previously declared challenge propagation relationships in Figure 3.8. The results were all consistent, showing that our dataset can distinguish

Table 3.2: Benchmarking results of symbolic execution tools.

Challenge	Case ID	KLEE		Triton		Angr	
		t = 60s	t = 300s	t = 60s	t = 300s	t = 60s	t = 300s
Buf. Overflows	stacknocrash_bo_l1	fail	fail	fail	fail	pass	pass
	stack_bo_l1	fail	fail	fail	fail	pass	pass
	heap_bo_l1	fail	fail	fail	fail	fail	fail
	stack_bo_l2	fail	fail	fail	fail	fail	fail
Covert Prop.	df2cf_cp	pass	pass	fail	fail	pass	pass
	echo_cp	fail	fail	timeout	timeout	timeout	timeout
	echofile_cp	fail	fail	fail	fail	timeout	timeout
	file_cp	fail	fail	timeout	timeout	fail	fail
	socket_cp	fail	fail	fail	fail	fail	fail
	stack_cp	inapp.	inapp.	pass	pass	pass	pass
	file_eh_cp	inapp.	inapp.	fail	fail	timeout	pass
	div0_eh_cp	inapp.	inapp.	fail	fail	timeout	pass
	file_eh_cp	inapp.	inapp.	fail	fail	timeout	fail
Sym. Memories	malloc_sm_l1	pass	pass	timeout	fail	pass	pass
	realloc_sm_l1	pass	pass	fail	fail	pass	pass
	stackarray_sm_l1	pass	pass	fail	fail	pass	pass
	list_sm_l1	inapp.	inapp.	fail	fail	timeout	pass
	vector_sm_l1	inapp.	inapp.	fail	fail	timeout	pass
	stackarray_sm_l2	pass	pass	fail	fail	fail	fail
	stackoutofbound_sm_l2	pass	pass	fail	fail	pass	pass
	heapoutofbound_sm_l2	fail	fail	timeout	fail	pass	pass
Sym. Jumps	funcpointer_sj_l1	pass	pass	fail	fail	fail	fail
	jmp_sj_l1	inapp.	inapp.	fail	fail	pass	pass
	arrayjmp_sj_l2	inapp.	inapp.	fail	fail	fail	fail
	vectorjmp_sj_l2	inapp.	inapp.	fail	fail	timeout	pass
Float. Num.	float1(2)_fp_l1	fail	fail	fail	fail	pass	pass
	float3(4)(5)_fp_l2	fail	fail	fail	fail	timeout	timeout
Arith. Overflows	plus_do	pass	pass	pass	pass	pass	pass
	multiply_do	pass	pass	fail	fail	pass	pass
Ext. Func. Calls	printint_ef_l1	fail	fail	pass	pass	pass	pass
	printfloat_ef_l1	fail	fail	fail	fail	fail	fail
	atoi_ef_l2	fail	fail	fail	fail	pass	pass
	atof_ef_l2	fail	fail	fail	fail	timeout	timeout
	ln_ef_l2	fail	fail	fail	fail	timeout	fail
	pow_ef_l2	fail	fail	fail	fail	pass	pass
	rand_ef_l2	fail	fail	timeout	timeout	fail	fail
	sin_ef_l2	fail	fail	fail	fail	timeout	timeout
Others	23 cases, no pass						
pass #	63 cases	9	9	3	3	17	22

the capability of different symbolic-execution tools accurately.

The efficiency of our benchmarking approach largely depends on the timeout setting. Note that Table 3.2 includes some timeout results; they account for most of our experimental time. Although we try to keep each logic bomb as succinct as possible, our dataset still contains some complex but unavoidable problems or path explosion issues. When the timeout value is 60 seconds, our benchmarking process for each tool takes only dozens of minutes. When extending the timeout value to 300 seconds, the benchmark takes a bit longer time. However, the benefit is not very obvious, and only Angr can solve 5 more cases. Can the result get further improved by allowing more time? We have tried another group of experiments with 1,800 seconds timeout. But the results remain unchanged. Therefore, 300 seconds should be a marginal timeout setting for our benchmarking experiment. Considering that symbolic execution is computationally expensive, which may take several hours or even several days to test a program, our benchmarking process is very efficient. We may further improve the efficiency by employing a parallel mode, such as assigning several logic bombs for each process.

Case Study

We now discuss the detailed benchmarking results for each challenge. Firstly, there are several challenges that none of the tools can trigger even one logic bomb, including symbolic variable declarations, parallel executions, contextual symbolic values, loops, and crypto functions. As for symbolic variable declaration challenge, they fail in modeling the conditions to trigger the logic bombs because all the tools cannot recognize the expected symbolic variables automatically. The challenges of contextual symbolic values and crypto functions involve tough problems, so it can be expected that all the tools fail in handling them. However, it is a bit surprising that none of the tools can handle parallel executions and loops.

Covert Propagations: Angr passed four test cases: `df2cf_cp`, `stack_cp`, and two exception handling cases. `df2cf_cp` propagates the symbolic values indirectly by substituting a data assignment operation with equivalent control-flow operations. KLEE also solved the case, but Triton failed. `stack_cp` propagates symbolic values via direct assembly instructions `push` and `pop`. Triton also solved the case. Besides, Angr also passed two test cases that propagate symbolic values via the C++ exception handling mechanism, which Triton failed. We further break down the details of an exception handling program (see Figure 3.10). As shown

```
double division(int num, int denominator) {
  if(denominator == 0 ) {
    throw "Division by zero!";
  }
  return (num/denominator);
}
```

```
int logic_bomb(char* s) {
  int symvar = s[0] - 48;
  try {
    division(10, symvar-7);
    return NORMAL_ENDING;
  }catch (const char* msg) {
    return BOMB_ENDING;
}}
```

(a) Source codes.

```
0x0000000000400aa9 <+41>:     callq  0x400a10 <_Z8divisionii>
0x0000000000400aae <+46>:     movsd  %xmm0,-0x40(%rbp)
0x0000000000400ab3 <+51>:     jmpq   0x400ab8 <_Z10logic_bombPc+56>
0x0000000000400ab8 <+56>:     movl   $0x0,-0x4(%rbp)
0x0000000000400abf <+63>:     jmpq   0x400afd <_Z10logic_bombPc+125>
0x0000000000400ac4 <+68>:     mov    %edx,%ecx
......                        ......
0x0000000000400ae1 <+97>:     callq  0x4008a0 <__cxa_begin_catch@plt>
0x0000000000400ae6 <+102>:    mov    %rax,-0x30(%rbp)
0x0000000000400aea <+106>:    movl   $0x1,-0x4(%rbp)
0x0000000000400af1 <+113>:    movl   $0x1,-0x34(%rbp)
0x0000000000400af8 <+120>:    callq  0x400890 <__cxa_end_catch@plt>
0x0000000000400afd <+125>:    mov    -0x4(%rbp),%eax
0x0000000000400b00 <+128>:    add    $0x40,%rsp
0x0000000000400b04 <+132>:    pop    %rbp
0x0000000000400b05 <+133>:    retq
0x0000000000400b06 <+134>:    mov    -0x20(%rbp),%rdi
0x0000000000400b0a <+138>:    callq  0x4008c0 <_Unwind_Resume@plt>
```

(b) Assembly codes.

Figure 3.10: Example of exception handling (division by zero).

in the box region of Figure 3.10(b), the mechanism relies on two function calls, which might be the problem that fails Triton. All the tools failed other covert propagation cases that propagate values via `fread/fwrite`, echo, socket, *etc*. Note that KLEE supports modeling file operations in POSIX standard such as `read/write`, but it cannot support C libraries directly.

Buffer Overflows: Only Angr could solve two easy buffer overflow problems: `stacknocrash_bo_l1` and `stack_bo_l1`. The cases share a simple stack overflow issue. Their solutions require modifying the value of the stack that might be illegal. However, Angr could not solve the heap overflow issue `heap_bo_l1`. It

```
int logic_bomb(char* s) {              if(l2_ary[l1_ary[x]] == 9){
   int symvar = s[0] - 48;                 return BOMB_ENDING;
   int l1_ary[] ={1,2,3,4,5};            }
   int l2_ary[] ={6,7,8,9,10};           else
                                           return NORMAL_ENDING;
   int x = symvar%5;                   }
```

(a) Source codes.

```
(gdb) break *logic_bomb+97
(gdb) run
(gdb) x/20xw $rsp-80
0x7fffffffe2d0: 0x00000006    0x00000007    0x00000008    0x00000009
0x7fffffffe2e0: 0x0000000a    0x00000000    0x004003e5    0x00000000
0x7fffffffe2f0:  0x00000001   0x00000002    0x00000003    0x00000004
0x7fffffffe300: 0x00000005    0x00007fff    0xf7fe2000    0x00000001
0x7fffffffe310: 0xffffe6db    0x00007fff    0x00000000    0x00000000
```

(b) Memory layout after array initialization.

```
text: 0x000000000040060d <+29>:      mov   0x4007c0,%rdi
text: 0x0000000000400615 <+37>:      mov   %rdi,-0x30(%rbp)
text: 0x0000000000400619 <+41>:      mov   0x4007c8,%rdi
text: 0x0000000000400621 <+49>:      mov   %rdi,-0x28(%rbp)
text: 0x0000000000400625 <+53>:      mov   0x4007d0,%ecx
text: 0x000000000040062c <+60>:      mov   %ecx,-0x20(%rbp)
text: 0x000000000040062f <+63>:      mov   0x4007e0,%rdi
text: 0x0000000000400637 <+71>:      mov   %rdi,-0x50(%rbp)
text: 0x000000000040063b <+75>:      mov   0x4007e8,%rdi
text: 0x0000000000400643 <+83>:      mov   %rdi,-0x48(%rbp)
text: 0x0000000000400647 <+87>:      mov   0x4007f0,%ecx
text: 0x000000000040064e <+94>:      mov   %ecx,-0x40(%rbp)
......                              ......
.rodata:00000000004007C0    dq 200000001h
.rodata:00000000004007C8    dq 400000003h
.rodata:00000000004007D0    dd 5
.rodata:00000000004007D4    align 20h
.rodata:00000000004007E0    dq 700000006h
.rodata:00000000004007E8    dq 900000008h
.rodata:00000000004007F0    dd 0Ah
```

(c) Assembly codes.

Figure 3.11: Example of the stack layout for array.

also failed on another harder stack overflow issue `stack_bo_l2`, which requires composing sophisticated payload, such as employing return-oriented programming methods [144]. We are surprised that Triton failed all the tests because binary-code-based symbolic execution tools should be resilient to buffer overflows in nature.

Symbolic Memories: The results show that Triton does not support symbolic memory, but KLEE and Angr provide very good support. Angr has solved seven cases out of eight. It only failed in handling the case depicted in Figure 3.11(a) with a two-leveled array `stackarray_sm_l2`. This implies that Angr would fail when there are multi-leveled pointers. In comparison, KLEE is able to solve the two-leveled array problem because it is based on STP [68], which is designed for solving such problems related to arrays. Figure 3.11(c) presents the assembly codes that initialize the arrays, while Figure 3.11(b) presents the stack layout after initialization. We note that the information about array size or boundary does not exist in assembly codes. This explains why binary-code-based symbolic execution tools do not suffer from problems when a challenge requires an out-of-boundary access, *e.g.*, `stackoutofbound_sm_l2`.

Symbolic Jumps: Since symbolic jump demonstrates no explicit conditional branches in the CFG, it should be a hard problem for symbolic execution. However, KLEE and Angr are not likely to be affected much by the trick. KLEE tackled the problem which has an array of function pointers `funcpointer_sj_l1`. Angr successfully handled two cases with assembly `jmp`, but it failed `funcpointer_sj_l1`.

Floating-point Numbers: The results indicate that KLEE and Triton do not support floating-point operations, and Angr can support some. During our tests, Triton directly reported that it could not interpret such floating-point instructions. Angr has solved two out of the five designated cases. The two passed cases are easier ones, which only require integer values as the solution. All the failed cases require decimal values as the solution, and they employ the `atof` function to convert `argv[1]` to decimals. Since Angr has also failed the test in handling `atof` in `atof_ef_l2`, the failures are likely to be caused by the `atof` function.

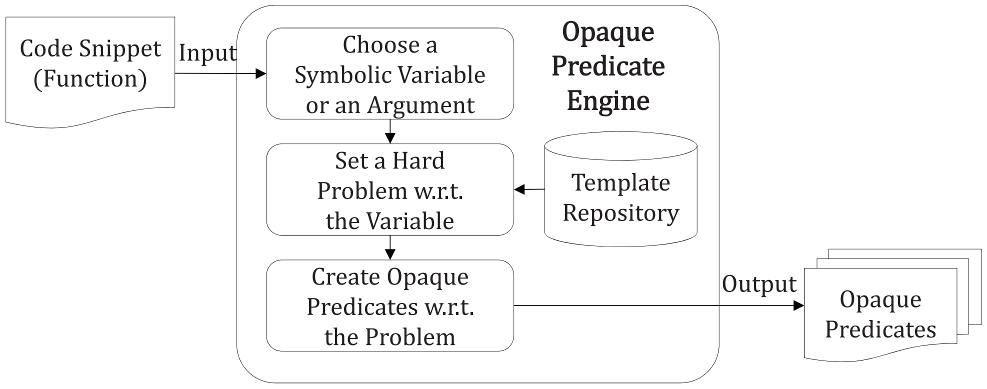

Code Snippet (Function)
Input
Opaque Predicate Engine
Choose a Symbolic Variable or an Argument
Set a Hard Problem w.r.t. the Variable
Template Repository
Create Opaque Predicates w.r.t. the Problem
Output
Opaque Predicates

Figure 3.12: Framework for composing opaque predicates.

Arithmetic Overflows: Arithmetic overflow is not a very hard problem since it only requires symbolic execution tools to handle such cases carefully. In our test, KLEE and Angr have been able to solve all the cases. However, Triton failed in handling the integer overflow case in Figure 3.6(e). The result shows there is still much room for Triton to improve for this problem.

External Function Calls: In this group of logic bombs, each case only contains one external function call. However, this result is very disappointing. Triton only passed a very simple case that print (with `printf`) a symbolic value of integer type. It does not even support printing out floating-point values. Angr has solved the `printf` cases and two more complicated cases, `atoi_ef_l2` and `pow_ef_l2`. It failed the `atof_ef_l2` and other cases. The results show that we should be cautious when designing logic bombs. Even when involving straightforward external function calls, the results could be affected.

```
int func(int symvar){
    int j = symvar;
    if(j == 7){
        Foo();
    }
}
```

(a) Original toy program.

```
int func(int symvar){
    int j = symvar;
    int l1_ary[] = {1,2,3,4,5,6,7};
    int l2_ary[] = {j,1,2,3,4,5,6,7};
    int i = l2_ary[l1_ary[j%7]];
    if(i == j)
        Bogus();
    if(i == 1 && j == 7)
        Foo();
}
```

(b) Symbolic memories.

```
int func(int symvar){
    int j = symvar;
    float f = j/1000000.0;
    if(f==0.1){
        Bogus();
    }
    if(1024+f == 1024 && f>0 && j==7){
        Foo();
    }
}
```

(c) Floating-point numbers.

```
int func(int symvar){
    int i, j = symvar;
    FILE *fp = fopen("covp.tmp", "w");
    if(fp == NULL) {
        exit(1);
    }
    fprintf(fp,"%d",j);
    fclose(fp);
    fp = fopen("covp.tmp", "r");
    fscanf(fp,"%d",&i);
    fclose(fp);
    if(i == 7){
        Foo();
    }
    if(i != j){
        Bogus();
    }
}
```

(d) Covert symbolic propagations.

```
int64_t ThreadProp(int64_t in){
    int64_t out = in;
    pthread_t tid;
    int rc1 = pthread_create(&tid,
                 NULL, Inc, (void *) &out);
    int rc2 = pthread_create(&tid,
                 NULL, Dec, (void *) &out);
    pthread_join(tid, NULL);
    return out;
}

int func(int symvar){
    int i, j = symvar;
    i = ThreadProp(j);
    if(i == 6){
        Foo();
    }
    if(i == j){
        Bogus();
    }
}
```

(e) Parallel executions.

Figure 3.13: Bi-opaque predicate examples.

3.6 Designing Bi-Opaque Predicates

3.6.1 Idea in a Nutshell

Intuitively, we can employ the weakness of symbolic execution to compose opaque predicates such that they can evade detection from symbolic execution-based adversaries. This is feasible because symbolic execution faces some challenges, and real-world symbolic execution tools have to adopt heuristic methods to handle them. Introducing such challenging problems into a program may incur error for symbolic execution.

Figure 3.12 demonstrates a general framework to compose such opaque predicates. Suppose the input is a code snippet or a function which contains arguments. Then we can choose an argument as the symbolic variable and create a challenging problem related to the variable. The challenging problem is selected from a repository of predefined templates. We may create hundreds of such templates by attacking different challenges of symbolic execution or employ different problem settings. Finally, we can create opaque predicates based on the symbolic variable protected by the problem.

Note that at least one symbolic variable should get involved in a challenging problem. Because only such problems matter to symbolic execution. If a problem does not include any symbolic value, all the problem-related instructions would be pruned by the symbolic execution engine. This can be proved with Hore Logic [87] following the principle of symbolic execution [186]. Because involving symbolic variables is a prerequisite for composing such opaque predicates, we name our opaque predicates as *symbolic opaque predicates*. If a function has no argument, then we have to introduce fake arguments or employ global symbolic variables.

3.6.2 Bi-Opaque Property

Traditional opaque predicates aim to evade from detection, such that the obfuscated control-flow graph cannot be easily simplified. In other words, they try to mislead adversaries into falsely recognizing them as normal predicates. Failing to detect them would cause *false negative* issues for adversaries. With symbolic opaque predicates, an interesting observation is that we may also introduce *false positive* issues, *i.e.,* we may mislead adversaries into falsely recognizing normal predicates as opaque predicates.

In this way, a predicate can be opaque in either a way, which is the novel *bi-opaque property* of our approach. Specifically, we name the two types of opaque predicates: *type I opaque predicate* which intends to introduce false negatives and *type II opaque* *predicate* which intends to introduce false positives. Next, we use several examples to demonstrate how to compose symbolic opaque predicates with the bi-opaque property.

3.6.3 Demonstration

Suppose Figure 3.13(a) is a function to obfuscate, then Figure 3.13(b) demonstrates how to obfuscate it with symbolic opaque predicates. Specifically, the predicates employ the challenge of symbolic memory.

Symbolic memories are difficult for program analysis because it involves pointer analysis issues, which can be NP-hard or even undecidable [102]. In this example, we compose two integer arrays. The symbolic value $j\%7$ points to an element within the first array, and the element serves as an offset of the second array. The selected element from the second array is assigned to a new variable i. In this way, i is a symbolic value protected by the challenging problem, and we can compose symbolic opaque predicates with i.

For example, we can compose a type I opaque predicate that cannot be satisfied, such as $i == j$. With the opaque predicate, we can insert a bogus code block (*i.e.,* `Bogus()`) which would never be executed. The security of the predicate depends on the capability of symbolic execution engines. If a symbolic execution engine employs no mechanism to handle symbolic memory, it would generate incorrect constraint models and falsely recognize the predicate as a normal predicate.

To compose a type II opaque predicate, we first select an ordinary predicate, $j == 7$. Then we modify the predicate by introducing a new condition related to i, such as $i == 1 \&\& j == 7$. The modification does not change the semantics of the original predicate because $i == 1$ is always true when j equals 7. Such condition can be easily generated because the value of i can be calculated from any j. In assembly codes, the new predicate will be dissembled

Algorithm 2: LLVM template corresponding to the example of symbolic memories in Figure 3.13(b).

```
/* Input an icmp instruction; output 2 opaque
   predicates                                                   */
input  : inst
output: type1Opq,type2Opq
/* Parse the icmp instruction                                   */
Value* left ← inst->getOperand(0) ;
Value* right ← inst->getOperand(1) ;
Value* symVar ;
ConstantInt* ciObj ;
if isa<ConstantInt> (*left) then
    ciObj ← left ;
    symVar ← right ;
end
else if isa<ConstantInt> (*right) then
    symVar ← left ;
    ciObj ← right ;
end
if !symVar->getType()->isIntegerTy() then
    return;
end
/* Define the size of the two arrays.                           */
ArrayType* ar1AT ← ArrayType::get(intType, 7) ;
ArrayType* ar2AT ← ArrayType::get(intType, 8) ;
/* Allocate storage for the arrays                              */
AllocaInst* ar1AI ← new AllocaInst(ar1AT, "", inst) ;
AllocaInst* ar2AI ← new AllocaInst(ar2AT, "", inst) ;
/* ...                                                          */
/* Here we omit several lines of codes that
   initialize the elements of each array.                       */
/* ...                                                          */
/* Create a new variable $j$ that equals to $symVar$,
   and then load $j$.                                           */
AllocaInst* jAI ← new AllocaInst(varType, "", inst) ;
StoreInst* jSI ← new StoreInst(symVar, jAI, inst) ;
LoadInst* jLI ← new LoadInst(jAI, "", inst) ;
/* Compute $j\%7$.                                              */
BinaryOperator* remBO ← BinaryOperator::Create(SRem, jLI,
   cInt7, "", inst);
/* Get an element from the array $ar1AI$ with an index
   $remBO$; load its value to $l1LI$.                           */
std::vector<Value*> l1Vec, l2Vec;
l1Vec.push_back(cInt0);
l1Vec.push_back(remBO) ;
ArrayRef<Value*> l1AR(l1Vec);
Instruction* l1EPI ← GetElementPtrInst::CreateInBounds(
   ar1AI, l1AR,"", inst);
LoadInst* l1LI ← new LoadInst(l1EPI,"", false, inst);
/* Get an element from the array $ar2AI$ with an index
   $l1LI$; load its value to $iLI$.                             */
l2Vec.push_back(cInt0);
l2Vec.push_back(l1LI);
ArrayRef<Value*> l2AR(l2Vec);
Instruction* l2EPI ← GetElementPtrInst::CreateInBounds(
   ar2AI, l2AR,"", inst);
LoadInst* iLI ← new LoadInst(l2EPI, "", false, inst);
/* Compose a type I opaque predicate, $i == j$.                 */
ICmpInst* type1Opq ← new ICmpInst(inst, ICMP_EQ, iLI,
   jLI, "");
/* Compose a type II opaque predicate,
   $i == j\%7 + 1 \&\& inst$ .                                  */
BinaryOperator* addBO ← BinaryOperator::Create(ADD,
   remBO, cInt1,"", inst);
ICmpInst* leftOpq ← new ICmpInst(inst, ICMP_EQ, iLI,
   cInt1, "");
BinaryOperator* andBO ← BinaryOperator::Create(AND,
   leftOpq, inst,"", inst);
ICmpInst* type2Opq ← new ICmpInst(inst, ICMP_EQ,
   cInt1,andBO,"");
```

into two predicates $i == 1$ and $j == 7$. The second predicate $j == 7$ will only be evaluated if the first predicate is true. If a symbolic execution engine does not support symbolic memory, it cannot solve the constraint of $i == 1$ and cannot reach the ordinary predicate $j == 7$.

3.6.4 Template Generalization

With the above example, we have demonstrated how our idea works in practice. Now we discuss how to implement the challenging problem as a template.

In general, a template is a code fragment in a compiler pass, which inserts, deletes, or modifies the program to be compiled. Algorithm 2 demonstrates such a template which implements the challenging symbolic memory problem in Figure 3.13(b). The algorithm inputs an `icmp` instruction and outputs symbolic opaque predicates. Suppose the `icmp` compares if a symbolic variable equals to an integer, the template first parses the instruction and get a symbolic variable $symVar$ and a constant $ciObj$. Then, we define the types of the two arrays and initialize them. Next, we can create an integer variable i and initialize it with the value $l2\_ary[l1\_ary[j\%7]]$.

Based on the protected symbolic variable i, we can directly create a type I opaque predicate with a comparison instruction $i == j$. To compose a type II opaque predicate, we have to introduce one more `icmp` instruction. The new instruction compares if i equals to a value, and it should be true if the original `icmp` (*i.e.,* $inst$) is true. In this example, according to the array setting, when j equals to a constant value, the value of i can be determined as $j\%7 + 1$.

3.6.5 Template Enrichment

Employing only one template is vulnerable to pattern recognition. We have to create different opaque predicates to increase the security level. This can be achieved in two ways. Firstly, we may create more templates by employing different problem settings. Secondly, we may create new templates by employing new challenges.

Employing New Settings

For each challenge that symbolic execution is faced with, we may compose a great many templates. Take the symbolic memory as an example, one can create arrays with different elements, employ a different modular, use three arrays instead of two arrays, store the array with heap instead of stack. All such methods ensure that the resulting symbolic opaque predicates are different in binaries or assembly codes.

Employing New Challenges

Another orthogonal approach is to employ new challenges, such as *floating-point numbers*, *covert propagations*, and *parallel executions*.

Figure 3.13(c) is an example that composes opaque predicates based on the challenge of floating-point numbers. A floating-point number is an approximation of a real number with a fixed length of digits in the form of $significant \times b^e$. It enables the computer to handle very large numbers or very small numbers with only limited memory space. As a trade off, floating-point numbers sacrifice the precision. Floating-point numbers may incur troubles to symbolic execution because reasoning over rational numbers and real numbers may lead to inconsistencies [23, 75]. In this example, because the float type cannot represent 0.1 precisely, no matter which value we assign to `symvar`, $f == 0.1$ cannot be satisfied. To compose a type II predicate, we can change the predicate $j == 7$ to $(1024 + f == 1024) \&\& (f > 0) \&\& (j == 7)$. The new predicate aims to fool symbolic execution engines that the constraint $(1024 + f == 1024) \&\& (f > 0)$ cannot be satisfied, which is true in the domain of real numbers. However, it can be satisfied in the domain of floating-point numbers. For example, $f = 0.000007$ is a solution. In this way, the type II opaque predicate can be satisfied when $j = 7$, which preserves the semantics. If a symbolic execution engine cannot handle such floating-point numbers, it may falsely regarded $f == 0.1$ as a normal predicate, and the type II predicate as an opaque predicate.

Figure 3.13(d) demonstrates how to compose opaque predicates by attacking the challenge of covert propagation. Symbolic execution requires precise tracking on the propagation of the symbolic values. However, symbolic values may be propagated in many ways via I/O (input/output) operations. In this example, the symbolic value j is propagated via a file on the disk and then assigned to

i. We can compose a type I opaque constant $i! = j$, which will always be false. If a symbolic execution engine cannot track the propagation, it would treat i as a constant and regard the opaque predicate as a normal one. To compose a type II opaque predicate, we can change the predicate $j == 7$ to $i == 7$, where i equals to j. This modification keeps the original semantics of the program. However, a symbolic executor may consider i as a constant and reach false conclusions.

Figure 3.13(e) is another example that introduces a simple parallel computing problem. Parallel executions are difficult to handle for symbolic executions because the execution order is not only determined by the programs, but also by the host computer. Therefore, we cannot generate a static control-flow graph for the program, which is a basis for classic symbolic execution to work. In this example, we create two more threads that modify the value of a symbolic variable j: one thread increases in to $in + 1$, and another decreases in to $in - 1$. Due to parallel execution, the two threads compute on the same value of in simultaneously. The value of i is determined by the thread that terminates late, which should be the second thread in our example. Finally, the return value of the `ThreadProp` function should equal to $j - 1$. Based on the protected symbolic variable i, we can compose a type I opaque predicate as $i == j$, and a type II opaque predicate as $i == 6$.

Similar to Algorithm 2, we can extract templates based on such examples. Note that this work does not intend to enumerate all such templates to create symbolic opaque predicates. Rather, we would like to show a general framework and demonstrate how it works. This can shed light to more types of symbolic opaque predicates.

3.7 Performance Evaluation

3.7.1 Evaluation Criteria

According to Collberg *et al.* [42], the evaluation criteria for assessing software obfuscation quality include *potency*, *resilience*, *stealth*, and *cost*. However, not all of the criteria are applicable to our work. We mainly evaluate symbolic opaque predicates with resilience, stealth, and cost.

Resilience evaluates how the obfuscation technique can hold up against automatic attacks. In this work, we assume the attackers are symbolic execution-based adversaries, which are automatic attacks. Section 3.5 already demonstrated the effectiveness of our approaches against symbolic execution engines.

Stealth assesses whether an obfuscation technique is suspicious to human attackers. A stealthy opaque predicate should not incur abnormal instruction patterns or obvious statistical difference with normal predicates.

Cost measures the overhead incurred by obfuscation. Opaque predicates may incur overhead in both program size and execution time. We should evaluate such overhead when obfuscating real programs with symbolic opaque predicates and compare the overhead with existing opaque predicates.

We do not evaluate *potency* because it is not applicable to opaque predicates. *Potency* measures how much obscurity can be added to the program. This is the major objective of general obfuscation or control-flow obfuscation, rather than opaque predicates.

3.7.2 Prototype Implementation

We have implemented a prototype obfuscation tool based on Obfuscator-LLVM [93]. Obfuscator-LLVM is an obfuscation tool for C programs based on LLVM compiler [106]. We adopt LLVM as our compiler basis because it is open-source released and has achieved

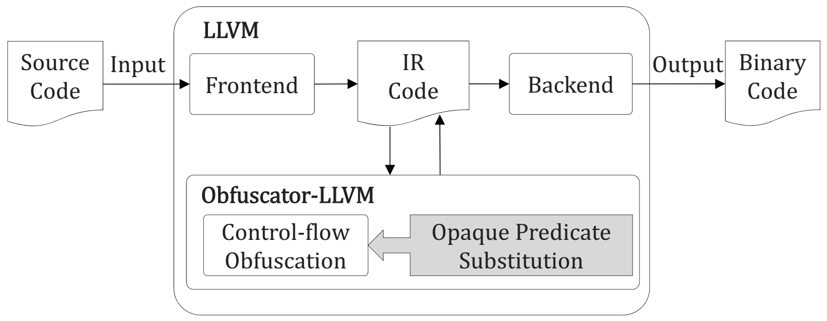


Figure 3.14: Prototype implementation based on Obfuscator-LLVM.

wide usage in both research and industrial fields.

Figure 3.14 describes the framework of our prototype. The source code of a program is firstly processed by an LLVM frontend, which transforms the source code to intermediate representatives (IR). For C programs, the frontend is Clang. IR is the core object processed in LLVM. LLVM provides a basic framework for performing program analysis tasks based on IR. It allows users to customize their own compilation passes for specific program analysis tasks, such as optimization and obfuscation. Obfuscator-LLVM in nature applies several compilation passes to obfuscate programs in IR level. Finally, the IR will be compiled to binaries by a corresponding backend (*e.g.,* for X86_64 system).

Based on the framework of LLVM, we implement the feature of symbolic opaque predicates as a compiler pass. The pass can substitute the opaque predicates generated by Obfuscator-LLVM with resilient ones. We have implemented all the challenging problems discussed in Section 3.6.5. Users can decide which opaque predicates will be employed during obfuscation.

Our prototype supports two methods to customize new templates of symbolic opaque predicates. The first one is to write a native LLVM pass which can insert IR (as shown in Algorithm 2) during compilation. To this end, users should be familiar with the IR syntax and LLVM APIs, which impose a steep learning curve. The

```
mov     edi, edx
sub     edi, 1
imul    edx, edi
and     edx, 1
cmp     edx, 0
setz    r8b
cmp     esi, 0Ah
setl    r9b
or      r8b, r9b
test    r8b, 1
jnz     loc_100002440
jmp     loc_1000045D1
```

(a) Obfuscator-LLVM.

```
cvtsi2sd   xmm0, dword ptr [rsi]
movsd      xmm1, [rbp+var_28]
divsd      xmm0, xmm1
cvtsd2ss   xmm0, xmm0
movss      dword ptr [rdx], xmm0
cvtss2sd   xmm0, dword ptr [rdx]
movsd      xmm2, [rbp+var_20]
ucomisd    xmm0, xmm2
mov        [rbp+var_51], r10b
jnz        loc_100001282
jp         loc_100001282
jmp        loc_1000014BF
```

(b) Floating-point numbers.

```
movsxd   rdx, dword ptr [rdx]
mov      rdi, rdx
mov      [rbp+var_41], r10b
mov      [rbp+var_50], rdx
call     _ThreadProp
mov      rcx, [rbp+var_50]
cmp      rcx, rax
jnz      loc_10000113C
mov      al, [rbp+var_41]
test     al, 1
jnz      loc_100000F98
jmp      loc_100001004
```

(c) Parallel executions.

Figure 3.15: The assembly codes of symbolic opaque predicates.

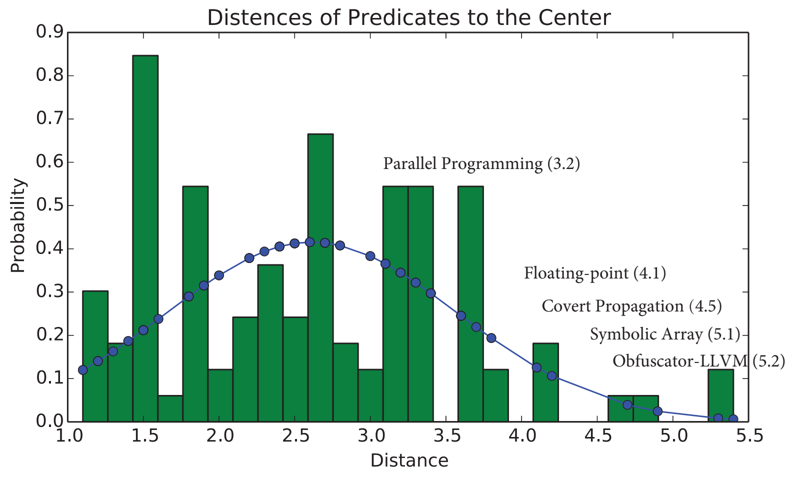


Figure 3.16: Comparing the stealth of symbolic opaque predicates with ordinary predicates.

second method requires only very little knowledge about LLVM development. Users can create new templates in source code level. Then they can compile the source code to object code and link it with the original program via static linkage. The second approach is somehow limited but it can facilitate the development process.

3.7.3 Stealth

Currently, there is no standard evaluation method for stealth. Existing methods (*e.g.,*[174]) generally measure the statistical dif-

Table 3.3: Categorization of Instructions.

Category	Instructions
Arithmetic Instructions	imul, inc, sub, add, idiv, divsd, sbb
Logical Instructions	and, sar, xor, test, shr, shl, or, xorps
Instructions for Data Transfer	movaps, movsd, movabs, movzx, mov, movss, movsx, movsxd, stosd
Instructions Converting Data Dimension	cvtss2sd, cvtsi2sd, cvtsd2ss, cqo, cdq
Pointer Instructions	lea
Comparison Instructions	cmp, ucomisd
Jump Instructions	jle, jne, jge, jae, jl, je,jg, jp, ja, jbe, jno, jmp
Stack-related Instructions	pop, push, call, ret
Instructions Creating Boolean Variable	setge, setne, setg, seta, setb, setl, sete
Other Instructions	nop

ference of instructions between obfuscated programs and ordinary programs. The less difference an obfuscation approach incurs, the stealthier it is.

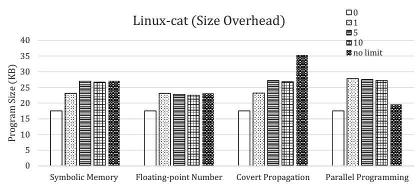
To apply the idea on evaluating symbolic opaque predicates, we should measure the difference between a symbolic opaque predicate and ordinary predicates. In general, the difference depends on which challenging problem that a predicate employs. Different problems will generate different codes and corresponding assembly instructions. Figure 3.15 demonstrates the assembly codes of several opaque predicates. Figure 3.15(a) is the default opaque predicate generated by Obfuscator-LLVM, which is mainly composed of arithmetic operations. Figure 3.15(b) is the symbolic opaque predicate with floating-point numbers, which is mainly composed of floating-point operations. The two figures demonstrate obvious difference; however, all such instructions are widely used in ordinary programs.

In our experiment, we use a similarity-based approach to measure the difference between symbolic opaque predicates and ordinary predicates. To this end, we randomly select 100 ordinary predicates from the unobfuscated binaries. For each predicate, we arbitrarily select the 10 instructions before its conditional jump because such instructions would serve as essential information for reverse analysis. Then we categorize such instructions into several types with a categorization approach employed for malware detection [95]. Table 3.3 lists the categories and corresponding instructions in each category. Considering the space where each dimension is an instruction category, a predicate can be represented as a vector in that space. Then we can compute the center of the 100 ordinary opaque predicates, and compute the euclidean distance from each predicate to the center. Figure 3.16 shows the distribution of such distances. In our experiment, the average distance is 2.6, and the max distance is 5.4. For comparison, we also compute the distances from our symbolic opaque predicates to the center, which are between 3.2, 4.1, 4.5, and 5.1. The distances are smaller than the max distance of ordinary predicate. Moreover, they are slightly better than the distance of the default opaque predicate employed in Obfuscator-LLVM, which is 5.2.

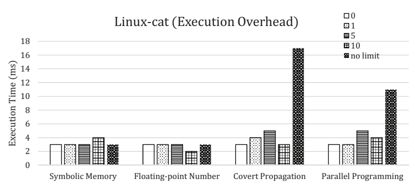
The opaque predicates based on parallel programming has the best performance in stealth. The main reason is that we have employed a call-based approach to implement the predicate. As shown in Figure 3.5(d), we implement the symbolic analysis problem in another function and only employ the return value in the main routine. In its binary code shown in Figure 3.15(c), only a `call` instruction is artificially added before the unconditional jump, and the rest instructions are mostly from the original program. By simply reading the instructions nearby a conditional jump, it would be difficult to discover the tricks of symbolic opaque predicates.

3.7.4 Cost

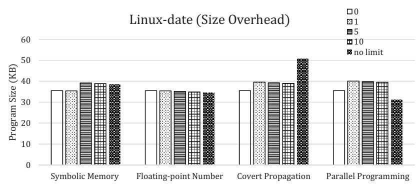
To evaluate the cost of symbolic opaque predicates, we obfuscate several general programs (*e.g.,* Linux commands such as `cat`, `ls`,



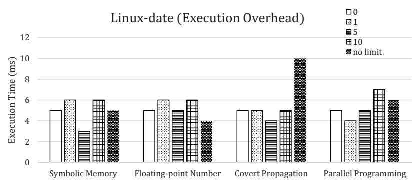
(a) Size overhead when obfuscating Linux command cat.



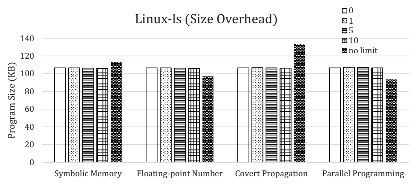
(b) Execution overhead when obfuscating Linux command cat.



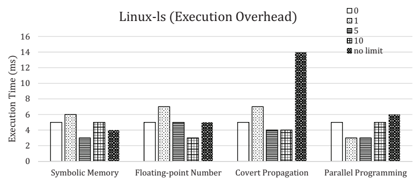
(c) Size overhead when obfuscating Linux command date.



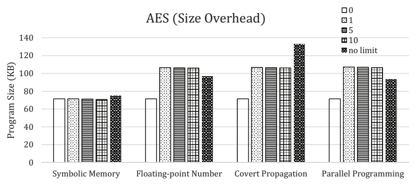
(d) Execution overhead when obfuscating Linux command date.



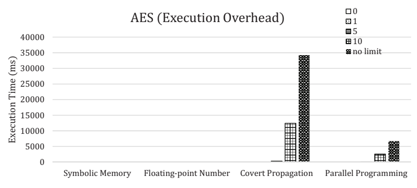
(e) Size overhead when obfuscating Linux command ls.



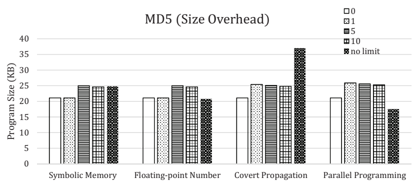
(f) Execution overhead when obfuscating Linux command ls.



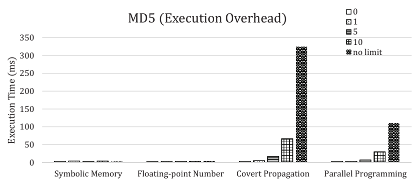
(g) Size overhead when obfuscating AES.



(h) Execution overhead when obfuscating AES.



(i) Size overhead when obfuscating MD5.



(j) Execution overhead when obfuscating MD5.

Figure 3.17: Cost of symbolic opaque predicates.

`date`) and several encryption programs (*e.g.,* MD5 and AES). We choose encryption programs because they generally have higher security requirements, and therefore obfuscation is more needed. When obfuscating the programs, we employ 80% obfuscation rate (*i.e.,* a configuration of LLVM-Obfuscator) as the baseline. Then for each program, we replace a certain number (1, 5, 10, and no limit) of opaque predicates from the obfuscated software with the symbolic opaque predicates. We watch the performance variations with different numbers of symbolic opaque predicates.

Figure 3.17 shows our evaluation results. We measure the performance of obfuscation with both program size and execution time. From the result, we observe that the size overhead is not a big issue. The symbolic opaque predicates based on symbolic memories and floating-point numbers both incur similar size overhead in comparison with the default opaque predicate employed by Obfuscator-LLVM. The sample of covert propagation involves more instructions and therefore incurs more overhead. However, such cost can be mitigated by employing a call-based implementation. For example, although the sample of parallel execution also involves many instructions, the resulting obfuscated program is even smaller than the program obfuscated by the original Obfuscator-LLVM.

Some symbolic opaque predicates are also very efficient in execution time, such as those based on symbolic memories and floating-point numbers. Their costs are similar to the default opaque predicates employed in Obfuscator-LLVM. However, some symbolic opaque predicates incur much cost during execution. As shown in *e.g.,* Figure 3.17(h) and Figure 3.17(j), the execution overhead may be thousands of times when employing covert propagation and parallel programming to obfuscate encryption programs. Such predicates involve heavy operations (*e.g.,* file read/write, thread creation/execution) and incur nontrivial execution cost. The overhead seems acceptable for general Linux programs, but it can be amplified for encryption programs because the symbolic opaque predicates are nested in loops in such programs.

In a word, the cost of symbolic opaque predicates depends on the employed challenging problems and their implementation mechanisms. Some symbolic opaque predicates can be very promising with trivial costs. But we should be careful when employing other opaque predicates with heavy cost, especially when using them with loops. In practice, we may prioritize the cost of symbolic opaque predicates and preemptively employ more efficient ones. Note that there is still a large room to improve the usability issue, which is beyond the scope of this work.

3.8 Related Work

In this section, we first survey the recent achievement of software deobfuscation with symbolic execution techniques, which illustrates the importance of our research problem; then we elaborate the novelty of our research by comparing our work with existing opaque predicates which might also be resilient to symbolic execution.

3.8.1 Symbolic Execution for Deobfuscation

Recently, the development of symbolic execution techniques has bred several important attempts to deobfuscation(*e.g.,* [122, 183, 188, 189]). Ming *et al.* [122] proposed LOOP, which is a logic-oriented tool for opaque predicate detection. LOOP is made up of a symbolic execution engine and a rule-based predicate analyzer. The rule can detect three types of opaque predicates, including invariant opaque predicates, contextual opaque predicates, and dynamic opaque predicates. Another work [183] from the same group employs symbolic execution techniques to detect malware camouflage from obfuscated binaries. Yadegari *et al.* [189] proposed a generic framework to deobfuscate binaries based on symbolic execution. Their framework collects traces generated by a symbolic execution engine and then employs the traces to simplify the obfuscated control-flow graph. Their work is based on an enhanced symbolic execution engine (*i.e.,* ConcoLynx [188]). However, the tool is not available for public evaluation.

Besides, there are several other investigations that attack obfuscated software with symbolic execution techniques, such as [10, 19, 77]. Because the underlying techniques are similar, we do not discuss each of them in detail.

3.8.2 Comparison with Existing Opaque Predicates

Before this work, Wang *et al.* [176] have conducted another investigation that has a similar purpose with us. They propose to compose resilient opaque predicates by attacking the weakness of symbolic execution in handling loops. Specifically, they create opaque predicates with *unsolved conjectures*, which is a form of looped codes. A common characteristic of such *unsolved conjectures* is that they would eventually exit the loops with some convergence properties. For example, the Collatz conjecture takes an input $x \in \mathbb{N}^+$, and iteratively calculates $x = x/2$ if x is even, otherwise calculates $x = 3x + 1$. No matter what value x has bee initialized with, the loop always terminates with x equals to 1. Besides, there are other predicates that maybe secure against symbolic execution, such as the opaque predicate with *one-way function* [156], and the predicate involving dynamic updated objects [42]. Note that all such opaque predicates are secure because they attack some weakness of symbolic execution. Such approaches also comply with our framework, and we may extend our template repository with them.

In a word, our work is different from previous work in that our framework is more general. We emphasize the importance of employing symbolic variables rather than leveraging specific tricks. In other words, we highlight the common properties for an opaque predicate to be secure against symbolic execution.

3.9 Conclusion

To conclude, this chapter studies the security issue of control-flow obfuscation with respect to symbolic execution-based attacks. To combat such attacks. we have proposed symbolic opaque predicates and demonstrated a general framework to compose such predicates. A novel characteristic of symbolic opaque predicates is the bi-opaque property, which can incur either false negative or false positive issues to symbolic execution-based attackers. To demonstrate the usability of our approach, we have implemented a prototype obfuscation tool based on Obfuscator-LLVM and conducted real-world experiments. We have evaluated the resilience, stealth, and cost of some symbolic opaque predicates. Evaluation results show that symbolic opaque predicates exhibit good resistance against prevalent symbolic execution engines. Some opaque predicate examples are also stealthy and efficient. Therefore, symbolic opaque predicates can serve as a promising way for practical obfuscation tools to improve their resilience to symbolic execution-based attacks.

□ **End of chapter.**

Chapter 4

N-Version Obfuscation

Software tampering attack is a notorious threat to current systems, such as Android. However, there is still no silver bullet for this problem with pure software techniques. This chapter introduces a novel obfuscation approach to combat such attacks. Rather than proposing new tricks against tampering attacks, we focus on impeding the replication of software tampering via program diversification and thus pose a scalability barrier against the attacks.

4.1 Rationale

Software is vulnerable to tampering attacks after release. Attackers may bypass its license checking mechanism and use restricted features, or they may pack malicious payloads into the software and disseminate infected packages [196]. Although there are already security mechanisms (*e.g.,* obfuscation and self-checksumming) to protect software from being reverse engineered, skillful attackers can bypass the protections with enough determination. It is often believed that software cannot achieve theoretically tampering-resilience without trusted hardware circuits [34]. However, hardware-assisted approaches suffer compatibility issues with current PC or smartphone taxonomy as they require specifically tailored hardware. Hence, investigating purely software-based approach is critical.

In this chapter, we propose a tampering-resilient solution which does not rely on hardware. Our approach is inspired by the lifecycle of tampering attacks, which often contains a replication phase to affect more hosts and gain as much benefit as possible. Intuitively, we may not guarantee a software instance to be fully tampering-resilient, but we can nullify the applicability of the tampering solution on other machines. Such an idea is inspired by the existing program diversification approach [64], which prevents the spreading of attacks by making intrusions much harder to replicate. If an attacker tries to launch attacks on another machine, she has to work on it specifically. In this way, we can disarm the ability of automated contagion and control the scope of potential damages.

As a first attempt, we propose to deliver the same featured, but functionally non-equivalent software versions to different machines. We name the approach as N-version obfuscation (NVO), and succinctly describe its major properties: *metamorphic*, *homomorphic*, and *automated*. The *metamorphic* property requires each version of the software to be unique in functionality so as to avoid the replication of tampered software; the *homomorphic* property enables a universal handler to handle the variance among different versions; the *automated* property automates the program compilation and delivery process concerning the N versions. We further provide a candidate solution for client-server applications. Our solution integrates a MAC (Message Authentication Code) mechanism with functionally non-equivalent SHA1 algorithms[1] into the original software. Our security analysis result shows that the attacking complexity incurred by NVO increases almost linearly with the number of functionally non-equivalent software versions, which would pose a scalability barrier against tampering attacks considering that an application can have millions of versions used by different users. It is worth noting that NVO itself provides no protection against tampering; however, it can be applied seamlessly

[1]FIPS PUB 180, Secure Hash Standard

to other existing tampering-resilient approaches, and hence equip them with the replication-resistant property.

The rest of this chapter is organized as follows. We first give more details about the motivation and background in Section 4.2. We then introduce our approach with a candidate solution in Section 4.3. Section 4.4 evaluates the effectiveness of our approach. The related work is discussed in Section 4.5. Finally, Section 4.6 concludes this chapter.

4.2 Motivation and Background

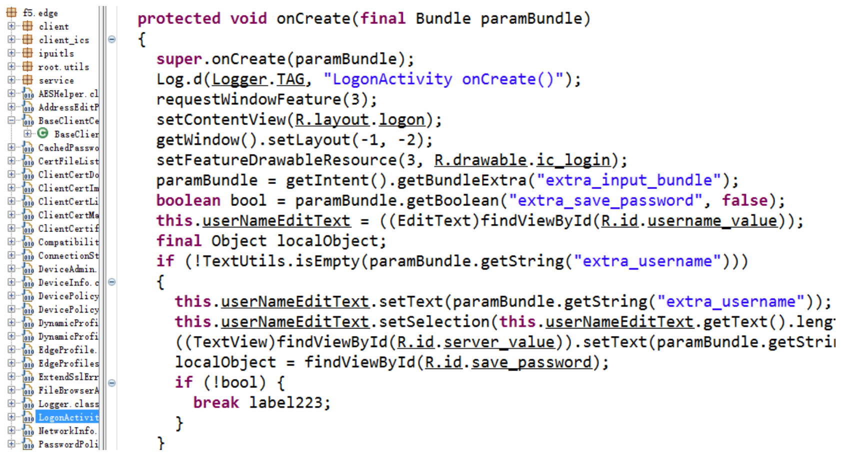
4.2.1 Adversary Model

This chapter assumes the hostile host model [147] whereas attackers can use a malicious host to analyze the software and inspect its execution step by step. In general, adversarial software tampering can be achieved in two ways: software repacking and dynamic injection. We discuss these two approaches as follows.

Software Repacking

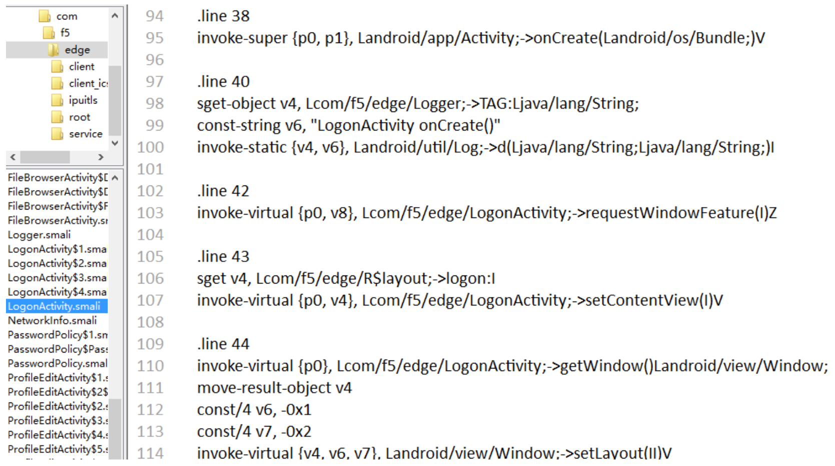
For many reasons, software installation packages downloaded by the users may not be the original ones. Take Android apps as an example, adversaries may replace the original advertisement module of the software for extra profits or plant malicious payloads for remote control. Such repacked apps are very common in either Google Play or third-party markets [195].

One important reason for the widespread of Android app repacking is that repacking Android apps is generally much easier than repacking traditional PC software written in C/C++, or iOS apps written in Objective C. Android program is mainly written in Java, and its installation package is delivered to end users in a compressed file with an `apk` extension. One major component of the installation package is the `classes.dex`, which wraps all the java classes.



```
protected void onCreate(final Bundle paramBundle)
{
  super.onCreate(paramBundle);
  Log.d(Logger.TAG, "LogonActivity onCreate()");
  requestWindowFeature(3);
  setContentView(R.layout.logon);
  getWindow().setLayout(-1, -2);
  setFeatureDrawableResource(3, R.drawable.ic_login);
  paramBundle = getIntent().getBundleExtra("extra_input_bundle");
  boolean bool = paramBundle.getBoolean("extra_save_password", false);
  this.userNameEditText = ((EditText)findViewById(R.id.username_value));
  final Object localObject;
  if (!TextUtils.isEmpty(paramBundle.getString("extra_username")))
  {
    this.userNameEditText.setText(paramBundle.getString("extra_username"));
    this.userNameEditText.setSelection(this.userNameEditText.getText().leng
    ((TextView)findViewById(R.id.server_value)).setText(paramBundle.getStri
    localObject = findViewById(R.id.save_password);
    if (!bool) {
      break label223;
    }
  }
```

(a) Java bytecodes opened with jd-jui.



```
.line 38
invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

.line 40
sget-object v4, Lcom/f5/edge/Logger;->TAG:Ljava/lang/String;
const-string v6, "LogonActivity onCreate()"
invoke-static {v4, v6}, Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I

.line 42
invoke-virtual {p0, v8}, Lcom/f5/edge/LogonActivity;->requestWindowFeature(I)Z

.line 43
sget v4, Lcom/f5/edge/R$layout;->logon:I
invoke-virtual {p0, v4}, Lcom/f5/edge/LogonActivity;->setContentView(I)V

.line 44
invoke-virtual {p0}, Lcom/f5/edge/LogonActivity;->getWindow()Landroid/view/Window;
move-result-object v4
const/4 v6, -0x1
const/4 v7, -0x2
invoke-virtual {v4, v6, v7}, Landroid/view/Window;->setLayout(II)V
```

(b) Corresponding smali code.

Figure 4.1: Example of disassembling Android apk.

To interpret the program logic of the app, one may convert the `classes.dex` to either Java bytecode (*i.e.*, `jar` file) or Dalvik bytecode (*i.e.*, smali code) with corresponding tools (*e.g.*, dex2jar, Apktool). Unlike assembly codes, the bytecodes are much easier to read. Figure 4.1 shows some examples of such bytecode snippets. Attackers can modify an application by rewriting the target bytecodes.

Note that Android provides an official package integrity checking mechanism based on the digital signature. However, many developers sign the APK with a self-signed digital certificate, which cannot be verified. For usability reasons, Android generally does not strictly forbid the installation of such repacked apps signed by an untrusted digital certificate.

Dynamic Injection

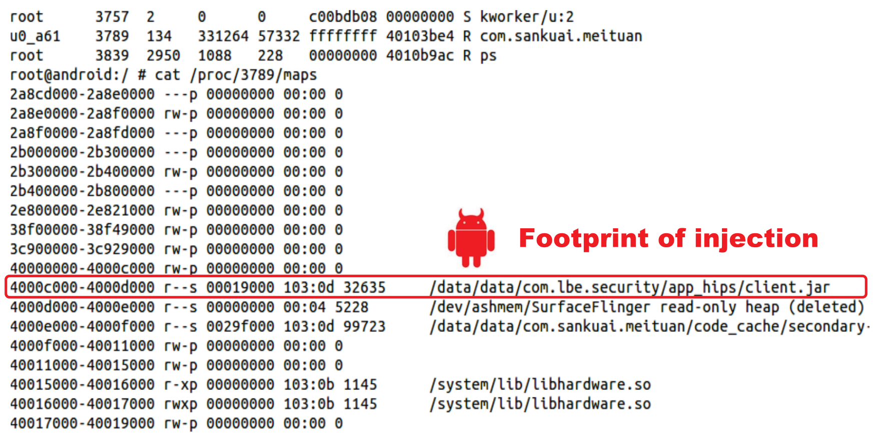
Attackers may also manipulate an app during its execution, *e.g.,* by injecting a library into the app process using Linux `ptrace` tool. This approach is popular among viruses, which place backdoors to control the program or monitor its execution. Besides, powerful anti-virus software also employs the same way to 'protect' the security of their clients. Figure 4.2(a) demonstrates an example of such dynamic injection attacks. The injected payloads can be very powerful and cause severe security issues. For example, we can obtain login credentials by injecting a payload into the process. Figure 4.2(b) demonstrates the result of our credential leakage experiment by injecting the VPN client of a famous vendor. Our another experiment shows that over 90% of apps can be dynamically injected [185].

4.2.2 Tampering-Resilience Background

Now, we overview the techniques of reverse engineering and review several major approaches to combat reverse engineering attacks.

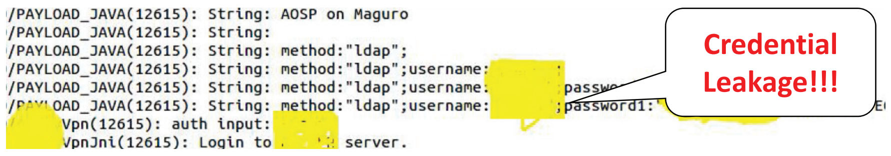
Reverse Engineering Overview

Software tampering involves a process to analyze and manipulate a software package based on its executables, *e.g.,* in an executable and linkable format (ELF). Anti-reversing techniques impede such a process by placing tricks in the executables to fool the analyzer. General reverse engineering on ELF files involves two phases: a disassembly phase and an analyzing phase. The disassembly phase



```
root      3757  2     0      0     c00bdb08 00000000 S kworker/u:2
u0_a61    3789  134   331264 57332 ffffffff 40103be4 R com.sankuai.meituan
root      3839  2950  1088   228   00000000 4010b9ac R ps
root@android:/ # cat /proc/3789/maps
2a8cd000-2a8e0000 ---p 00000000 00:00 0
2a8e0000-2a8f0000 rw-p 00000000 00:00 0
2a8f0000-2a8fd000 ---p 00000000 00:00 0
2b000000-2b300000 ---p 00000000 00:00 0
2b300000-2b400000 rw-p 00000000 00:00 0
2b400000-2b800000 ---p 00000000 00:00 0
2e800000-2e821000 rw-p 00000000 00:00 0
38f00000-38f49000 rw-p 00000000 00:00 0
3c900000-3c929000 rw-p 00000000 00:00 0
40000000-4000c000 rw-p 00000000 00:00 0
4000c000-4000d000 r--s 00019000 103:0d 32635     /data/data/com.lbe.security/app_hips/client.jar
4000d000-4000e000 r--s 00000000 00:04 5228      /dev/ashmem/SurfaceFlinger read-only heap (deleted)
4000e000-4000f000 r--s 0029f000 103:0d 99723     /data/data/com.sankuai.meituan/code_cache/secondary-
4000f000-40011000 rw-p 00000000 00:00 0
40011000-40015000 rw-p 00000000 00:00 0
40015000-40016000 r-xp 00000000 103:0b 1145      /system/lib/libhardware.so
40016000-40017000 rwxp 00000000 103:0b 1145      /system/lib/libhardware.so
40017000-40019000 rw-p 00000000 00:00 0
```

(a) App (pid:3789) has been injected by LBE.



```
/PAYLOAD_JAVA(12615): String: AOSP on Maguro
/PAYLOAD_JAVA(12615): String:
/PAYLOAD_JAVA(12615): String: method:"ldap";
/PAYLOAD_JAVA(12615): String: method:"ldap";username:[redacted]
/PAYLOAD_JAVA(12615): String: method:"ldap";username:[redacted]passw
/PAYLOAD_JAVA(12615): String: method:"ldap";username:[redacted];password1:'
[redacted]Vpn(12615): auth input:[redacted]
VpnJni(12615): Login to[redacted] server.
```

(b) Experiment to steal credentials by dynamically hijacking the function `java...tostring()`.

Figure 4.2: Example of dynamic injection.

decodes the ELF binaries to assembly code, which can be performed automatically by some tools (*e.g.,* IDA [2]). We can hardly impede the decoding because ordinary processors should be able to decode the program. Therefore, the reverse engineering and corresponding anti-reversing efforts are mainly related to the analyzing phase.

There are two general ways to do a reverse analysis, *i.e.,* the static approach and the dynamic approach. The static approach does not execute the assembly codes but directly analyzes them using reverse engineering tools such as IDA. Although there are many static analysis tools are available off-the-shelf, the power of pure static approaches is very limited. For example, they cannot detect runtime unpacking which has been widely used by malware to escape static analysis. A more powerful approach is dynamic analysis, which analyzes a program via real executions [58, 136].

[2]https://www.hex-rays.com/products/ida/

There are many approaches which can obstruct the analyzing phase, such as obfuscation, anti-debugging, and self-checksumming.

Anti-Debugging

Researchers have suggested setting traps with anti-debugging code to hinder debugging. For example, one may simply check the debug register to detect if a debugger is present, or he can count the execution time of a code block to detect if it has been paused, and then penalize the debugger [67, 157]. If the trick of anti-debugging code can be recognized, adversaries may suppress the checking by patching the binaries or switching to another debugger.

Self-Checksumming

When deriving enough understanding about the code, adversaries can manipulate the binaries by adding or deleting some code according to a specific purpose while preserving its ability of execution. A possible way to detect such code patching is to use self-checksumming code. The basic idea is to pre-calculate relative addresses (*i.e.,* the checksum), and let the program fetch instructions during execution according to such addresses. If the checksum governed regions have been manipulated, the instruction would not be correct, and the program would likely to suspend [179]. Using overlapped self-checksumming code can further increase the strength of protection. However, it can be defeated by carefully detecting and removing them [137] or exploring the vulnerabilities [179] of the execution environment.

4.2.3 Challenge of Tampering-Resilient Apps

Apps can be very complex. They can involve classes written in Java, native code written in C or C++, and other third-party libraries, all of which are vulnerable to tampering. Therefore, a universal safeguard is required to protect the integrity of each component. Since no general tools can be applied for such heterogeneous codes, the implementation of tampering-resilience for the whole program would be labor intensive. Moreover, mobile apps are usually upgraded more frequently than general PC software, so that their testing strategy and releasing criteria cannot be as rigorous as PC. Consequently, the laborious tampering-resilient implementation and testing would likely to slow down the releasing speed, or insufficient testing on such low-level code tends to cause more bugs.

Besides, we should also consider the overhead incurred by security mechanisms. Traditional anti-tampering approaches usually work by adding extra code to the original program, which can complicate the control flow of the original program, or by performing some integrity checking. Such approaches inevitably incur overhead, and a trade-off between the effectiveness and the overhead should be considered. Note that for some resource-constrained mobile devices cannot afford much overhead.

Finally, according to the literature [34], it is impossible for software to be absolutely secure against analysis without specific hardware protection. Although there are some existing solutions for Android apps, such as the ProGuard offered by Google, DexGuard, and AppInk [195], there are no general criteria regarding the acceptable tampering-resilient strength. It is desirable to develop an anti-tampering solution whose security can be proved or quantified, and referenced by the developers.

4.3 Our Proposed Approach

While achieving theoretically tampering-proof is hardly possible, our idea aims to pose the tampering attack unscalable. In this section, we formally define the idea of NVO and then introduce a candidate solution for networked apps.

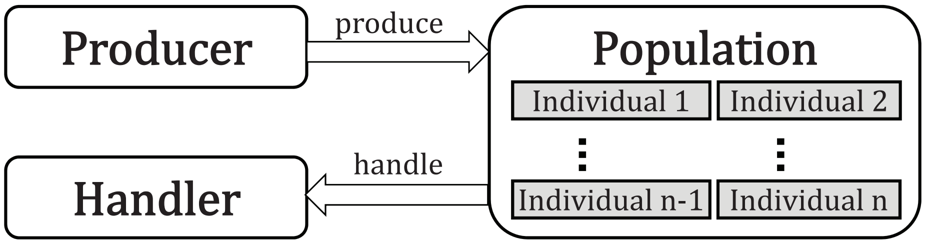


Figure 4.3: Conceptual framework of NVO.

4.3.1 General Idea of NVO

We formally define the NVO problem as the following: Given an algorithm A, how to automatically generate a large set of functionally non-equivalent algorithms $\{C_1, ...C_n\}$, which are similar to A, and their parent algorithm P, so that they meet the following three properties:

Homomorphic: When performing on the same task, P can output the same result as C_i, if the gene vector $\{g_1, ...g_n\}$ of C_i is known to P.

Metamorphic: When performing on the same task, C_i and C_j generally output different results.

Automated: The generation and delivery of $\{C_1, ...C_n\}$ can be automated.

Figure 4.3 demonstrates the conceptual framework of NVO. The producer generates a set of functionally non-equivalent individuals, *i.e.,* $\{C_1, ...C_n\}$. The handler communicates with each individual and processes their requests leveraging a parent algorithm P. Suppose the software architecture is in client-server mode, we can deploy the handler at the server side, and deliver the individuals to the client side. In this way, the client can have functionally non-equivalent diversities according to the metamorphic property, and the homomorphic property enables the server to handle such diversities. To make the idea practical, we should automate the generation and delivery of such diverse software versions.

4.3.2 Our Candidate Solution

To apply NVO on apps, one major issue to address is regarding which part of an app can have effective functionally non-equivalent diversities. Intuitively, there are two possible ways: we can either find the candidate code snippet in the original program or add some extra code to the original program. Generally, the first approach is program dependent, and can hardly be generalized. Hence, our approach is to add extra code which can achieve the intended diversities.

A possible way is to add MAC to the original program. MAC is a popular mechanism adopted by client-server computing architecture to check the integrity and authenticity of messages. When a client sends a request to the server, it calculates the MAC of the request and appends it to the original request. The server validates the MAC first and then processes the request. We can leverage the MAC to create clients with functionally non-equivalent diversities. More specifically, the diversity can be introduced based on the hash algorithm (*e.g.,* SHA1), which is one major component of a MAC algorithm. Figure 4.4 illustrates such a mechanism. Each client is embedded with a unique SHA1-based MAC calculation algorithm. To successfully perform a request to the server, it has to send the identification (such as machine serial number or user id), the request, and the MAC together to the server. The server queries the genes of a client from its local N-version database according to the identification of the client and then verifies the MAC. The distribution of such diverse programs can be achieved by implementing the MAC in mobile code (*i.e.,* a dynamic library), and delivering it by the server upon request. In other words, the client software can be launched without the library for the first time and then requests the server for the library. The server randomly chooses a library from a pool of pre-compiled libraries and delivers it to the client; in the meanwhile, the server records the mapping

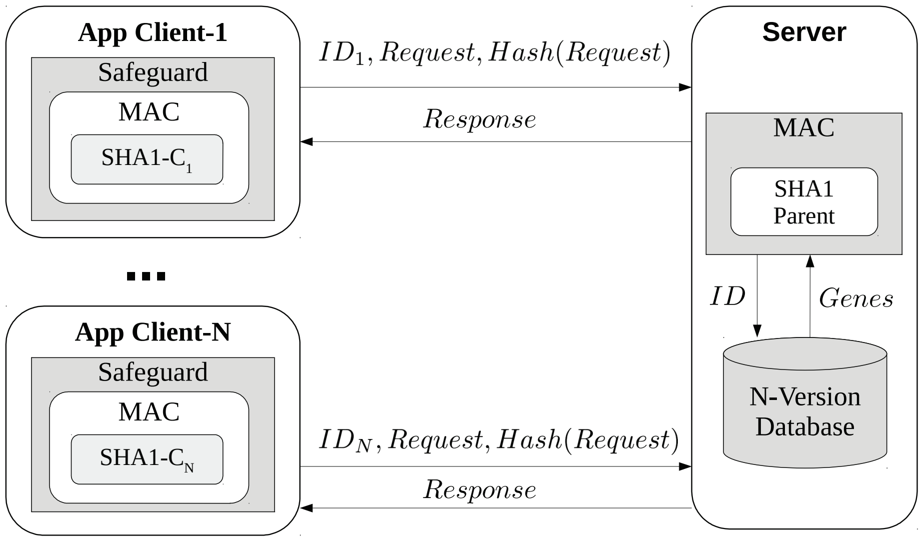


Figure 4.4: Sample of NVO for tampering-resilient apps.

between the genes of the client and its unique identification in the N-version obfuscation database. Figure 4.5 demonstrates a detailed safeguard delivery and initialization process.

In the following paragraphs, we first show a viable means to solve the NVO problem with the SHA1 algorithm, and then discuss the security measurements which can be built on the mechanism.

N-version Obfuscated SHA1

Our approach leverages the iterations of calculations needed by SHA1 to generate functionally non-equivalent diversities. The main loop of original SHA1 (Algorithm 3) includes 80 rounds of iterations. Each iteration takes one plaintext block ($w[i]$) into calculation. For every twenty rounds, the calculation (the equation for generating f and the value of k) switches to another one. Even though there are some security considerations of choosing a specific calculation for each round, to our best knowledge, no evidence shows the programs would suffer great security degradation if we switch them with each other. Therefore, we can diversify the original SHA1 algorithm by choosing different sequences of equations for generating f and

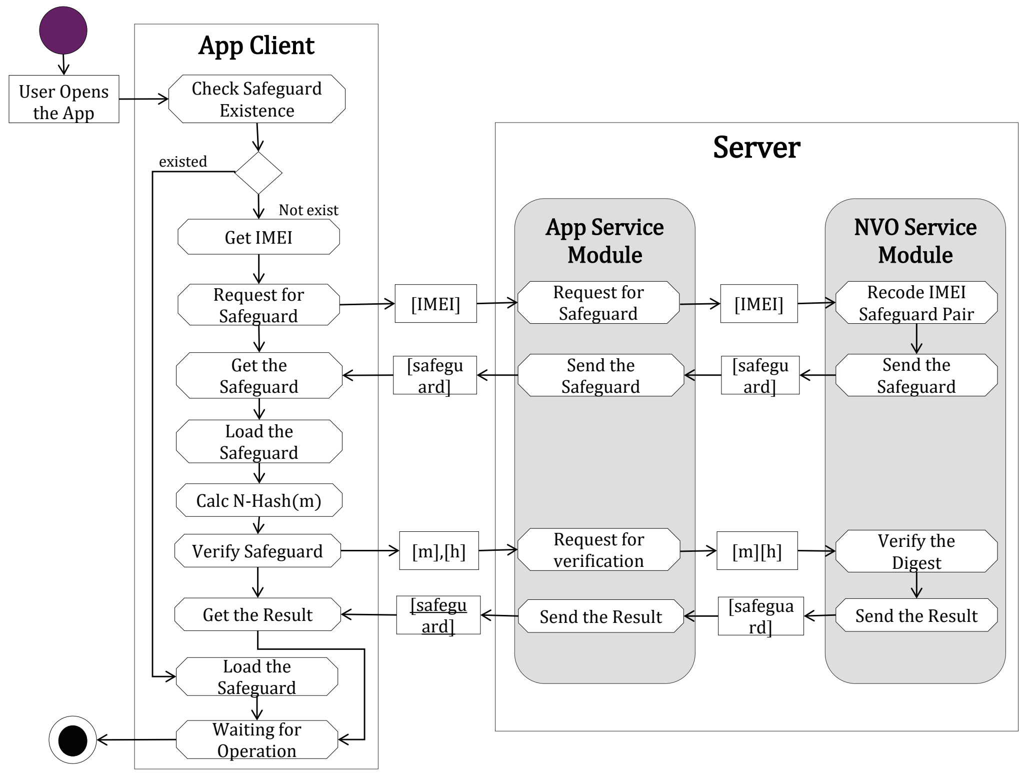


Figure 4.5: Activity diagram to automate the process of safeguard delivery and initialization.

sequences of values of k, which are the genes of individuals. We can also design a parent algorithm which can receive the genes of an individual, and process data input according to the setting of genes. Algorithm 4 shows such a parent algorithm we designed. In Algorithm 4, the pointer array of equations ($f\_genes[80]$) for generating f and the value array of k ($k_{gen}[80]$) for the 80 rounds of iterations are passed to the algorithm as the genes of a child. Obviously, given the same input $w[80]$, the parent algorithm can compute the same result as a child when $f_{gen}[80]$ and $k_{gen}[80]$ are properly set.

Algorithm 3: The main loop of SHA1.

```
Data: $w[80]$
// blocks of plaintext
for $i = 0; i < 80; i++$ do
    if $0 \leq i \leq 19$ then
        $f \leftarrow (b$ AND $c)$ OR $(($NOT $b)$ AND $d)$;
        $k \leftarrow$ 0X5A827999;
    end
    if $20 \leq i \leq 39$ then
        $f \leftarrow b$ XOR $c$ XOR $d$;
        $k \leftarrow$ 0X6ED9EBA1;
    end
    if $40 \leq i \leq 59$ then
        $f \leftarrow (b$ AND $c)$ OR $(b$ AND $d)$ OR $(c$ AND $d)$;
        $k \leftarrow$ 0X8F1BBCDC;
    end
    if $60 \leq i \leq 79$ then
        $f \leftarrow b$ XOR $c$ XOR $d$;
        $k \leftarrow$ 0XCA62C1D6;
    end
    $temp \leftarrow (a$ LEFTROTATE $5) + f + e + k + w[i]$;
    $e \leftarrow d$;
    $d \leftarrow c$;
    $c \leftarrow b$ LEFTROTATE $30$ ;
    $b \leftarrow a$;
    $a \leftarrow temp$;
end
```

Security based on MAC

The N-version obfuscated SHA1 program itself provides little effectiveness against software tampering attack. However, it is resistant to replication, because the server cannot verify the MAC of replicated programs. Therefore, it can serve as a basis for software integrity checking, and equip programs with a replication-resistance property. In this section, we discuss one possible way to build such security features on top of the MAC.

A viable means is to implement an integrity checking function aligning with the MAC in the safeguard so that it can serve as

Algorithm 4: A parent algorithm for SHA1

```
Data: $f_{gen}[80], k_{gen}[80], w[80]$
for $i = 0; i < 80; i++$ do
    Call $f_{gen}[i]$;
    // Pointer to F0, F1, F2 or F3
    F_TAIL($k_{gen}[i], w[i]$);
end
F0() $f \leftarrow (b$ AND $c)$ OR ((NOT $b$) AND $d)$;
F1() $f \leftarrow b$ XOR $c$ XOR $d$;
F2() $f \leftarrow (b$ AND $c)$ OR $(b$ AND $d)$ OR $(c$ AND $d)$;
F3() $f \leftarrow b$ XOR $c$ XOR $d$;
F_TAIL($k, w$) $temp \leftarrow (a$ LEFTROTATE $5) + f + e + k + w$;
$e \leftarrow d$;
$d \leftarrow c$;
$c \leftarrow b$ LEFTROTATE 30;
$b \leftarrow a$;
$a \leftarrow temp$;
```

a safeguard for the whole app. By interleaving the code of the integrity checking function with the MAC algorithm, the integrity checking can be triggered when calculating a MAC. Algorithm 5 shows an exemplary integrity checking function for the apps of Android operating system. The function navigates the maps file of the app process itself, which records the program segments and their addresses in the memory. It then compares the record with a previously defined standard dictionary by the developers. If there is any abnormal segment in the maps, *i.e.,* the integrity has been violated, a responsive mechanism can be triggered. Such an approach is effective in detecting either software repacking or dynamic injection attacks as we have discussed in Section 4.2. For example, Algorithm 5 can detect the tampering in Figure 4.2(a) by finding that `com.lbe.../client.jar` is an abnormal segment.

If an attacker has successfully tampered one copy of the safeguard (*e.g.,* removing the integrity checking function) and replicated it on other machines, the server can detect the replication because of an incorrect MAC, *i.e.,* inconsistent mapping between the identifica-

Algorithm 5: Example of integrity checking function.

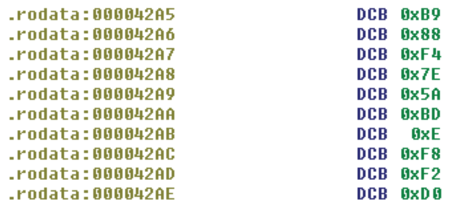
```
Data: $dict < segment >$
// A list of predefined segment with name and size
IntegrityChk() $pid \leftarrow$ getpid();
$file \leftarrow$ open (/proc/pid/maps);
while $line \leftarrow$ readline($file$) != EOF do
    $segName \leftarrow$ GetSegName($line$);
    $segSize \leftarrow$ GetSegSize($line$);
    if !dict.contains($segNmae$) then
        Reaction();
    else
        if dict.getsize($segNmae$)!=$segSize$ then
            Reaction();
        end
    end
end
```

tion and the genes. We may further implement a reaction mechanism to renew the safeguard or crash the client software directly.

Protecting the Genes

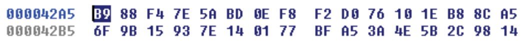
Genes are the secrets of the diversity and should be resilient to adversarial extraction. Without protection, the N-version software executables are generated in plain ELF binaries. Adversaries may find the gene sections by comparing several versions of the software, and extract the genes manually, or even automatically. Figure 4.6 demonstrates the genes of the safeguard located using IDA. To provide protections for the secrets from being extracted, we randomly change the meaning of the genes, *i.e.,* the same value of $f_{gen}[i]$ for different versions may trigger different operations. We further adopt two methods to protect the meaning of genes from being reasoned: functional obfuscation, and control-flow obfuscation with opaque constants.

a) *Functional obfuscation*: Adversaries may reason the meaning of the gene by checking the call relationship with some functions (*e.g.,* $F0, F1$ in Algorithm 4). We hence obfuscate the functions



```
.rodata:000042A5                DCB 0xB9
.rodata:000042A6                DCB 0x88
.rodata:000042A7                DCB 0xF4
.rodata:000042A8                DCB 0x7E
.rodata:000042A9                DCB 0x5A
.rodata:000042AA                DCB 0xBD
.rodata:000042AB                DCB  0xE
.rodata:000042AC                DCB 0xF8
.rodata:000042AD                DCB 0xF2
.rodata:000042AE                DCB 0xD0
```

(a) Genes in the .rodata section of the safeguard using IDA View.



```
000042A5  B9 88 F4 7E 5A BD 0E F8  F2 D0 76 10 1E B8 8C A5
000042B5  6F 9B 15 93 7E 14 01 77  BF A5 3A 4E 5B 2C 98 14
```

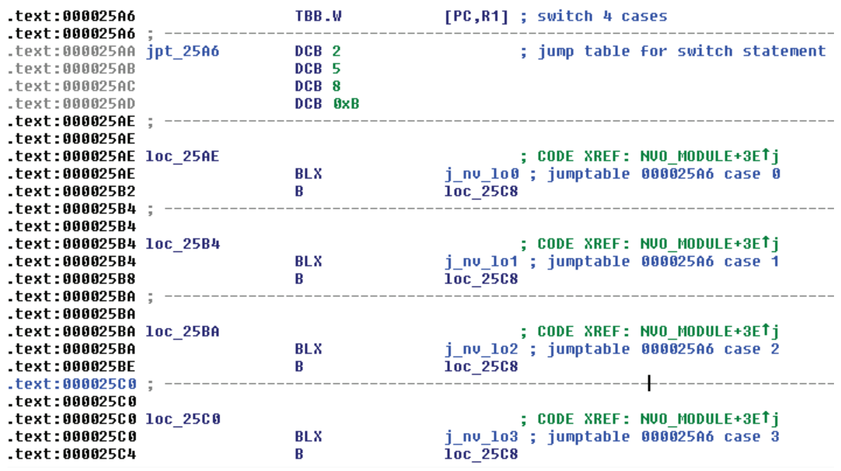
(b) The bits of genes using HexView.

Figure 4.6: Locating the genes with IDA.

in each version from being located. Firstly, we change the function names to random strings, so that attackers cannot locate the functions easily located by their names. Secondly, we change the order of those functions, so that they appear in different positions of the executables. In this way, even when adversaries have extracted the genes, they still have trouble in mapping the genes with the functions.

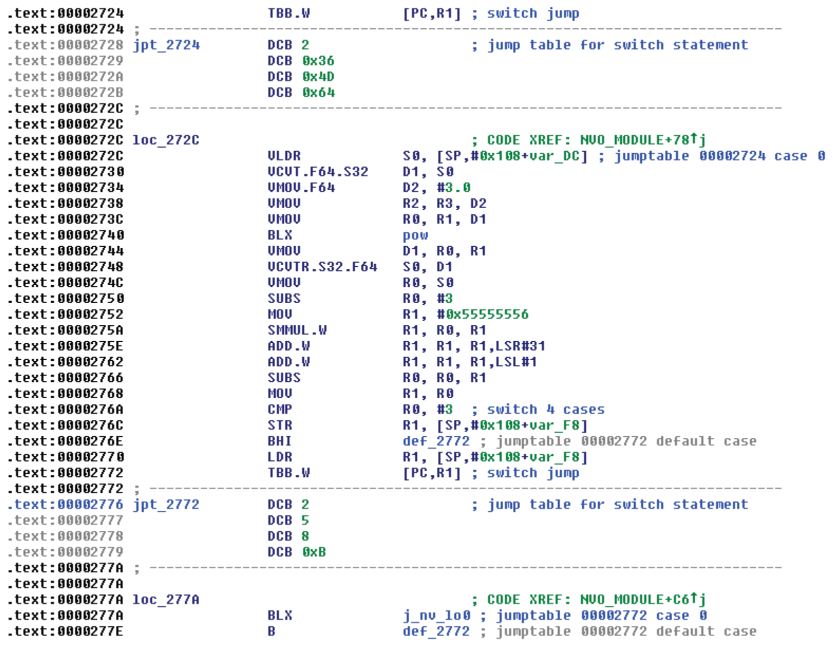
b) *CFG obfuscation*: In this step, we obfuscate the CFG, so that even the functional obfuscation can be penetrated, the calling relationship between the genes and the functions would not be easily solved. To this end, we adopt the obfuscation approaches proposed in [127], which composes NP-hard problems with function pointers and opaque constants. A comparison of the instructions before and after the CFG obfuscation is shown in Figure 4.3.2.

Finally, it is worth noting that the obfuscation protections we adopt to protect the secrets in this section are all functional equivalent transformations. The NVO approach itself does not provide any resistance to reverse engineering. However, our approach can be seamlessly integrated with other anti-reverse-engineering protections, such as anti-checksumming. We may use them together to provide better tamper-resistant capabilities.



```
.text:000025A6                 TBB.W           [PC,R1] ; switch 4 cases
.text:000025A6 ; ---------------------------------------------------------------------------
.text:000025AA jpt_25A6        DCB 2                   ; jump table for switch statement
.text:000025AB                 DCB 5
.text:000025AC                 DCB 8
.text:000025AD                 DCB 0xB
.text:000025AE ; ---------------------------------------------------------------------------
.text:000025AE
.text:000025AE loc_25AE                                ; CODE XREF: NVO_MODULE+3E↑j
.text:000025AE                 BLX             j_nv_lo0 ; jumptable 000025A6 case 0
.text:000025B2                 B               loc_25C8
.text:000025B4 ; ---------------------------------------------------------------------------
.text:000025B4
.text:000025B4 loc_25B4                                ; CODE XREF: NVO_MODULE+3E↑j
.text:000025B4                 BLX             j_nv_lo1 ; jumptable 000025A6 case 1
.text:000025B8                 B               loc_25C8
.text:000025BA ; ---------------------------------------------------------------------------
.text:000025BA
.text:000025BA loc_25BA                                ; CODE XREF: NVO_MODULE+3E↑j
.text:000025BA                 BLX             j_nv_lo2 ; jumptable 000025A6 case 2
.text:000025BE                 B               loc_25C8
.text:000025C0 ; ---------------------------------------------------------------------------
.text:000025C0
.text:000025C0 loc_25C0                                ; CODE XREF: NVO_MODULE+3E↑j
.text:000025C0                 BLX             j_nv_lo3 ; jumptable 000025A6 case 3
.text:000025C4                 B               loc_25C8
```

(a) Before CFG obfuscation, the function calling can be easily mapped with the genes in the jump table.



```
.text:00002724                 TBB.W           [PC,R1] ; switch jump
.text:00002724 ; ---------------------------------------------------------------------------
.text:00002728 jpt_2724        DCB 2                   ; jump table for switch statement
.text:00002729                 DCB 0x36
.text:0000272A                 DCB 0x4D
.text:0000272B                 DCB 0x64
.text:0000272C ; ---------------------------------------------------------------------------
.text:0000272C
.text:0000272C loc_272C                                ; CODE XREF: NVO_MODULE+78↑j
.text:0000272C                 VLDR            S0, [SP,#0x108+var_DC] ; jumptable 00002724 case 0
.text:00002730                 VCVT.F64.S32    D1, S0
.text:00002734                 VMOV.F64        D2, #3.0
.text:00002738                 VMOV            R2, R3, D2
.text:0000273C                 VMOV            R0, R1, D1
.text:00002740                 BLX             pow
.text:00002744                 VMOV            D1, R0, R1
.text:00002748                 VCVTR.S32.F64   S0, D1
.text:0000274C                 VMOV            R0, S0
.text:00002750                 SUBS            R0, #3
.text:00002752                 MOV             R1, #0x55555556
.text:0000275A                 SMMUL.W         R1, R0, R1
.text:0000275E                 ADD.W           R1, R1, R1,LSR#31
.text:00002762                 ADD.W           R1, R1, R1,LSL#1
.text:00002766                 SUBS            R0, R0, R1
.text:00002768                 MOV             R1, R0
.text:0000276A                 CMP             R0, #3  ; switch 4 cases
.text:0000276C                 STR             R1, [SP,#0x108+var_F8]
.text:0000276E                 BHI             def_2772 ; jumptable 00002772 default case
.text:00002770                 LDR             R1, [SP,#0x108+var_F8]
.text:00002772                 TBB.W           [PC,R1] ; switch jump
.text:00002772 ; ---------------------------------------------------------------------------
.text:00002776 jpt_2772        DCB 2                   ; jump table for switch statement
.text:00002777                 DCB 5
.text:00002778                 DCB 8
.text:00002779                 DCB 0xB
.text:0000277A ; ---------------------------------------------------------------------------
.text:0000277A
.text:0000277A loc_277A                                ; CODE XREF: NVO_MODULE+C6↑j
.text:0000277A                 BLX             j_nv_lo0 ; jumptable 00002772 case 0
.text:0000277E                 B               def_2772 ; jumptable 00002772 default case
```

(b) After CFG obfuscation, the function calling related to the genes has been obfuscated using opaque constants and sub jump tables.

Figure 4.7: Example of obfuscation for `switch/case`.

Generating N-versions

We automate the process of generating N-version SHA1 algorithms based on LLVM, which is a widely used open-source compiler that supports extensions. LLVM first represents the source code with Abstract Syntax Tree (AST) and then transfers it into intermediate code (IR), which would finally be compiled into executables according to a specific platform. The automation can be achieved in two ways: in AST level by customizing a `libtooling` (*i.e.,* an LLVM tool that can manipulate the source code of a target AST branch during the compilation process), or in IR level by adding extra N-version obfuscation passes to the compiler. We suggest the second way because IR is machine independent and provides better adaptability.

According to Algorithm 4, each gene (either $fp[i]$ or $k[i]$) has four possibilities, and we can use two bits to represent a gene. During each compilation, we first randomly generate two 160-bit long sequences: one as the chromosome for the equation function pointer (*i.e.,* $fp[80]$) and the other as the chromosome for the value option of k (*i.e.,* $k[80]$). We then replace the corresponding code with hardcoded genes. Similarly, we can implement the obfuscation approaches by adding obfuscation passes for protecting the genes in a similar way as that in [93].

4.3.3 Approach Discussion

Several ideas proposed in the literature are very close to NVO, such as white-box encryption, and N-version programming (NVP). In this section, we compare NVO with these ideas and clarify why NVO is a unique approach to security.

White-box Encryption

NVO creates functionally non-equivalent diversities among versions in the level of program logic. A question to ask is why we do not simply use different keys to compose diversities. For example, we may use a keyed-hash message authentication code (HMAC) algorithm and hardcode a unique symmetric key into each version. Note that such an approach is also effective, but it is

Table 4.1: Comparison of NVP and NVO.

	NVO	**NVP**
Purpose	Security:tampering resistant	Reliability: fault tolerant
Fault	Malicious faults	Accidental faults
Assumption	Independent obfuscation	Independent programming
Program	Functionally non-equivalent	Functionally equivalent
Generation	Automatically generated	Independently designed
Population	Very large	Very small
Effectiveness	$O(N)$ security	$1-(1-R)^N$ reliability
Cost	$O(1)$	$O(N)$

more vulnerable than our proposed NVO approach because hiding a key (*i.e.,* white-box cryptography) is more difficult than hiding the program logic [36]. White-box cryptography can be viewed as an extreme circumstance of NVO with only key diversities. Besides, white-box cryptography does not stress on producing diversities, which is the major focus of NVO. Essentially these two approaches are two orthogonal frameworks, each with its own objectives and algorithms. Nevertheless, our approach may incorporate white-box cryptography for a hybrid security mechanism.

N-version Programming

NVO improves software security by automatically generating different versions of the software. It is inspired by the classical N-version Programming (NVP) approach, which improves software reliability by independently designing different versions of software, so that the same bug may not happen in all versions [33, 116]. Although the two ideas are similar, they target in solving different problems, and they are very different in several key aspects. Table 4.1 presents a detailed comparison of NVO and NVP.

4.4 Evaluation

NVO aims to impede the replication of attack by creating diversified software instances and increase the complexity for intruding multiple clients. In this evaluation section, we first discuss the effectiveness of NVO in thwarting tampering replication and then evaluate the complexity incurred by NVO for intruding multiple software clients. Note that our evaluation only considers the software tampering attack, and we do not consider other types of attacks, such as side-channel attacks.

4.4.1 Security Effectiveness

Suppose a program has adopted the protection mechanism discussed in Section 4.3.2. If a decent attacker wants to manipulate the program through software repacking or dynamic injection, she has to disarm the security safeguard by removing or modifying the security code discussed in Algorithm 5. According to the adversary analysis, the safeguard cannot be simply removed or disabled from the app, because the MAC mechanism rested in the safeguard needs to be executed. However, in a hostile host environment, the software can be fully inspected. Through careful analysis, the attacker may discern that the protection lies in the integrity checking function of Algorithm 5. If she is skillful and spends enough efforts, she can further disable the checking by carefully modifying the function, such as suppressing the reaction. If there is no NVO protection, the attacker may replicate the repacked app, or apply her dynamic injection scripts on other machines, and the whole software ecosystem would be contaminated. However, NVO can impede such replication of tampering attack, with detailed discussions in what follows.

If the attacking type is app repacking, then the repacked app replicated on multiple machines would have the same genes for the MAC algorithm. Suppose the app (*e.g.,* ebank) uses `UserID` as the corresponding unique `ID` for the genes (as we have discussed in Figure 4.4), then the server would receive mismatching MAC from the app that has been logged on, and thus can detect that the client app has been tampered. In this way, we may take advantage of the user's credential, which cannot be easily faked. But what if the app mainly provides services to anonymous users that do not require logging on? Generally, such kind of apps does not have strong security requirements. Having said that, NVO still works for such apps by employing other information as the `ID`, such as the International Mobile Equipment Identity (IMEI). The major difference is that IMEI can be faked much easier. For example, the repacked app can hardcode the faked IMEI corresponding to the genes of the app. However, when replicating on multiple machines, the server would detect the abnormality that multiple clients are using the same IMEI. In a nutshell, due to the divergence property of NVO, the server can detect app repacking attack when the app is communicating with the server. Such a detection condition is trivial because, for many apps, pure clients are useless unless they can interact with the server (*e.g.,* ebank, shopping), or obtain rich contents from the server (*e.g.,* news, videos).

If the attacking type is dynamic injection, the sample integrity checking function (*e.g.,* Algorithm 5) is effective in detecting the tampering. Although it relies little on the NVO mechanism, the NVO hardens the security of the integrity checking function against being suppressed. For example, if the attacker seeks to suppress the security checking in Algorithm 5, an intuitive way is to disable the `Reaction()` function. To locate the function within the ELF file, the attacker may either check the related ELF table (*e.g.,* .dynsym and .rel.plt) dynamically or hardcode the address of the function into the malicious code. However, our NVO implementation transforms such self-defined function names to a random alphabet combination for each version, so that the dynamic approach cannot know which symbol designates the target function. Besides, the hardcoded address also cannot work because the function would appear at different positions of the binaries for each version, due to our functional obfuscation implementation.

4.4.2 Security Strength

To replicate tampering attacks, attackers have to bypass our NVO settings. Intuitively, they may either suppress the security checking in each version dynamically or create a library which is similar to the parent algorithm and extract the genes of each version. We discuss the complexity of these two kinds of attacks in what follows.

Suppressing Security Checking

To suppress the security checking of a software instance, attackers should obtain the safeguard on that machine and then remove the checking instructions within the safeguard. If the safeguard is protected with interleaved self-checksumming code [31], a successful tampering requires removing all the self-checksumming code at the same time, of which the chance is very low without sophisticated analysis. Existing approaches to identify such code generally require dynamic taint analysis and debugging [137]. Empirically, the time required to tamper each safeguard is not negligible.

Let t_0 denote the time needed for analyzing one software copy and tampering it on the attacker's own hostile host. The time complexity is $O(1)$, which equals to tampering one software copy without NVO. Let t_1 denote the time needed to fetch the safeguard on another machine, so as to replace it, and t_2 denote the time needed to tamper it. If the attacker wishes to tamper the software on n machines, the total time can be estimated as $t_0 + n * (t_1 + t_2)$. Because of the interleaved self-checksumming code, t_2 should not be negligible [136]; hence the complexity can be approximated to $O(n)$.

Universal Attacker

Another possible tampering approach is to build an algorithm similar to the parent, which calculates the hash value according to the genes of a specific child. Such an approach requires the attacker to be able to extract the genes from each safeguard. To this end, the attacker may compare the difference between the two implementations, and locate the genes. If the attacker has derived enough knowledge on our NVO theory and implementation, such kind of attack is theoretically possible. However, in our NVO implementation, the meanings of the genes differ in different versions, because they have been obfuscated with opaque constants [127]. To our best knowledge, existing work on breaking such obfuscated programs requires either symbolic execution with sophisticated constraint solvers or complicated taint analysis [189], which is computationally intensive and time-consuming. Let t_3 denote the time needed to extract the genes of a safeguard. The time needed to tamper the software ecosystem can be estimated to $t_0 + n * (t_1 + t_3)$. Because efficient automatic deobfuscation is hard to achieve, t_3 should not be negligible [2, 57]. Therefore, the complexity still equals to $O(n)$. Note that n can be made arbitrarily large as the obfuscation task can be fully automated.

Finally, our complexity analysis results rely on the problem incurred by traditional anti-tampering protections. But different from the traditional work, we do not require the anti-tampering protections to approach theoretical secure, which can hardly be guaranteed. We only require that the protections cannot be thwarted automatically, which is more sound and realistic.

4.4.3 Overhead

Now, we discuss the overhead of our candidate solution for each client. Our baseline is a networked app which has already implemented a security checking mechanism which is not resilient to large-scale attacks. In this way, the overhead is mainly incurred by the traditional software protection techniques and newly added message authentication mechanism. While the first part of the overhead is dependent on specific protection techniques, the second part is more easy to evaluate. In particular, the size overhead of the SHA1 algorithm in binaries is about 10 KB, which is very small considering the size of an app. The execution overhead and network overhead are trivial in comparison with other popular transportation-layer security mechanisms (*e.g.,* SSL/TLS) adopted by such apps.

4.5 Related Work

To protect apps from unauthorized manipulation, Google offers ProGuard, which is a free obfuscator and optimizer for Android apps that can make the application harder to be analyzed. There are also premium versions of such tools (*e.g.,* DexGuard) which are more powerful. To our best knowledge, they provide no features against tampering replication.

Recent literature in protecting users from using tampered apps mainly focuses on detecting repacked apps in a large scale, such as [61, 155, 164, 171, 196, 197]. However, these investigations are not quite related to our problem. Several other investigations focus on detecting repacked apps with watermarking approaches [142, 195]. Zhou *et al.* [195] proposed the idea of manifest apps and the corresponding tool, AppInk. AppInk can take the source code of an app as input and automatically generate a new app with a transparently-embedded watermark and the associated manifest app. The manifest app can then be used for verification purpose by triggering certain app control flows to regenerate the watermark. Ren *et al.* [142] proposed another watermarking solution (*i.e.,* Droidmarking) for app plagiarism detection. Droidmarking is based on a primitive called self-decrypting code, and the watermark locations are not intentionally concealed. These watermarking approaches are generally effective for app repacking detection within the scope of code plagiarism, but they are not effective for other kinds repacking, such as third-party library replacement.

To protect apps from being repacked, Zhou *et al.* [194] proposed an approach that re-encodes an Android app with a transformed virtual instruction set, so that general reverse engineering tools cannot inspect the app. To run the protected app, the developers can use a specialized execute engine for these virtual instructions. The idea is similar to another work proposed by Shu *et al.* [159]. These approaches can increase difficulties for interpreting the program, and they are effective against popular reverse engineering tools. But their security relies on the secret of the instruction translation table, which can be discovered manually by decent adversaries.

As we have discussed, existing work in this area mainly focuses on increasing the difficulty of repacking, or the detection of repacked apps. Our work is different from them in that we focus on decreasing the reusability of repacked apps.

4.6 Conclusion

This chapter focuses on impeding the replication of software tampering, which is a unique perspective for anti-tampering research. Our proposed NVO approach can automatically generate and deliver functional non-equivalent software versions to different machines. In this way, it can disable the adaptability of a tampering approach to different victims. To demonstrate the applicability of NVO in practical scenarios, we propose a candidate solution for general networked apps. Specifically, the candidate solution introduces functional non-equivalence with a MAC mechanism. Our evaluation result shows that the achieved functionally non-equivalent diversities can be effective against tampering replication, and the complexity to tamper the software ecosystem is linearly increased with the number of software versions, which can be automatically generated with trivial cost.

Although the NVO idea is promising, this work can be extended in various ways. Our candidate solution highly depends on the characteristic of network communications and the MAC mechanism. More solutions are expected in the future to help us explore the technique more thoroughly. Besides, our candidate solution incorporates merely existing anti-tampering approaches. A systematic study on how to effectively combine them are needed. Finally, the candidate solution has not been examined publicly, and its security should be further improved with real-world applications.

□ **End of chapter.**

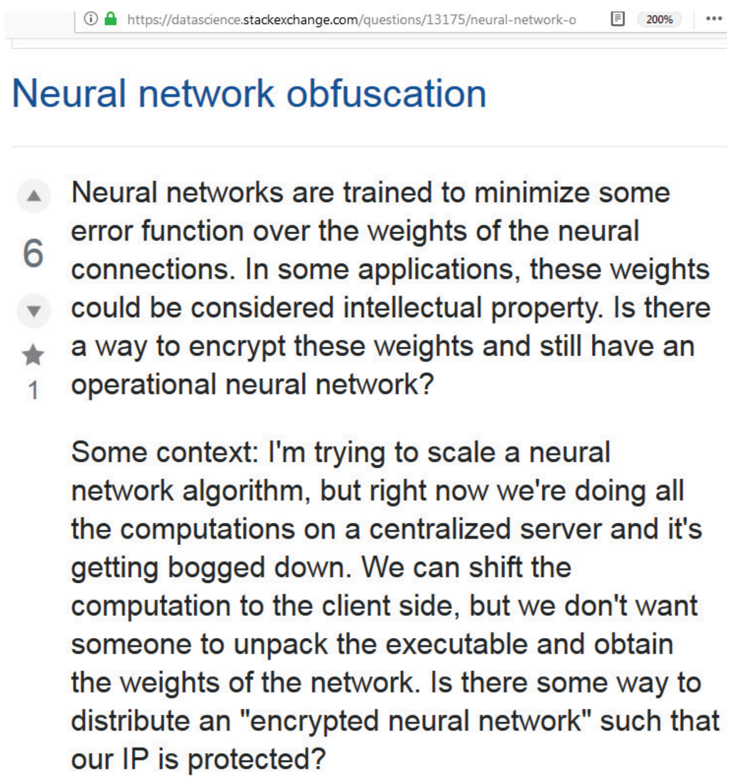
Chapter 5

DeepObfuscation

This chapter investigates the piracy threat to deep learning models. Designing and training a well-performing model is generally expensive. However, when releasing them, attackers may reverse engineer the models and pirate their design. Therefore, we propose to solve the problem with a novel obfuscation technique.

5.1 Rationale

We are experiencing a booming development of deep learning technologies. Nowadays, more and more systems employ deep learning models, such as self-driving cars [21] and face recognition systems [131]. Karpathy even proposes the concept of *Software 2.0* referring to the software written in neural network weights. If we treat deep learning as a new paradigm of programming, it actually suffers many software security and reliability issues. For example, DeepXplore [133] studies software testing approaches for deep learning models, because there are already several attacks (*e.g.,* [126, 130]) showing the effectiveness in fooling them. Another study [167] shows that attackers can steal machine learning models via prediction APIs. Therefore, the security of deep learning techniques becomes an urgent issue when being deployed in real-world systems.


https://datascience.stackexchange.com/questions/13175/neural-network-o
200%
6
1

Neural network obfuscation

Neural networks are trained to minimize some error function over the weights of the neural connections. In some applications, these weights could be considered intellectual property. Is there a way to encrypt these weights and still have an operational neural network?

Some context: I'm trying to scale a neural network algorithm, but right now we're doing all the computations on a centralized server and it's getting bogged down. We can shift the computation to the client side, but we don't want someone to unpack the executable and obtain the weights of the network. Is there some way to distribute an "encrypted neural network" such that our IP is protected?

Figure 5.1: A sample of real-world requirement for neural networks obfuscation solutions from an Internet forum.

In this chapter, we consider a particular security threat to deep learning, *model piracy*. Programming a superior deep learning model is expensive. It requires much domain expertise to design an effective deep learning network and a large set of labeled data to train the network, both of which are valuable resources. Because well-trained models are expensive, competitors or attackers may get interested in pirating them. An intuitive way is to copy the architecture of a network, which dominates the learning ability of a model. Besides, attackers may fine-tune a model for their own application scenarios. There are already many investigations focusing on building new deep learning applications based on existing ones, such as those with transfer learning [69, 129, 191] and incremental learning [141, 180] techniques. If a deep learning model runs on the client side, which is a trend (*e.g.,* smartphone [90, 103, 173]), attackers can easily reverse engineer the model and further pirate the design. Therefore, model piracy is a pressing security concern for deep learning application providers. Figure 5.1 demonstrates such a real-world protection requirement posted on an Internet forum[1]. To our best knowledge, there are no good solutions so far.

To secure a deep learning model against piracy attack, we propose to obfuscate the structures of well-trained deep learning models before releasing them to clients. Our idea is similar to classic code obfuscation techniques except that we tailor the idea for deep learning scenarios. Code obfuscation transforms code snippets into unintelligible versions while preserving their semantics [40]. Deep learning obfuscation, on the other hand, aims to scramble the structure of a well-designed deep learning network while preserving the inference accuracy. In this way, users can still employ an obfuscated model for inference, but attackers cannot learn useful structural information from the model. To our best knowledge, it is a first attempt to study the deep learning obfuscation problem.

Our study focuses on a prevalent type of deep learning networks, convolutional neural networks (CNN). Many companies start to engage CNN in their systems for image recognition tasks, *e.g.,* in mobile apps or auto-driving systems. To achieve a good recognition accuracy, state-of-the-art CNNs generally contain well-designed inception blocks for feature extraction and a fully-connected layer for classification. For example, GoogLeNet [165] employes four parallel convolutional sequences in one inception block with different settings to learn different features; ResNet [84] employs a special convolutional branch to learning residual information. For modern CNNs, their key difference generally lies in the design of the feature extractor, while the classifiers of different CNNs are very similar. Therefore, hiding the real structure of a feature extraction network is a major concern for obfuscation.

We propose to obfuscate the feature extractor of a CNN model

[1]https://datascience.stackexchange.com/questions/13175/neural-network-obfuscation

by simulating it with a shallow and sequential convolutional block. Consequently, the simulation network leaks little structural information about the original feature extractor. Meanwhile, the obfuscated model should also be resilient to fine-tuning attacks because the simulation network bears poor learning abilities due to a shallow structure. To simulate a feature extraction network precisely, we incorporate a novel *recursive simulation* method and a *joint training* method to training the simulation network. The recursive simulation method simulates a feature extractor in a recursive mode. In the first round, we simulate each inception block of a feature extractor with a simulation network. In the second round, we simulate the entire simulated feature extractor achieved in the first round. During each iteration of simulation, we employ the joint-training method to train a simulation network, *i.e.,* we employ both the intermediate output of the original network and the labels of the training data as the ground truth. Finally, we can obtain an obfuscated model with no loss of accuracy.

To verify the feasibility of our idea, we have conducted real-world experiments with popular CNNs, including GoogLeNet [165], ResNet-18 [84] and DenseNet-121 [91]. We choose these public CNNs for evaluation only because they are well-known to readers. In practice, we believe that our experimental results with these models should also be applicable to other private CNNs. Our final experimental results show that although these networks are very deep with tens or even hundreds of layers, we can simulate them with a shallow network of five or seven layers. We present that the obfuscated models suffer no loss of accuracy. On the other hand, they are even more efficient than the original models in both model size and inference time. We further show that the obfuscated models demonstrate promising resilience to fine-tuning attacks. Attackers would suffer obvious accuracy declination if they fine-tune the obfuscated models to create new applications.

To summarize, we make several contributions as follows.

- We formulate the deep learning obfuscation problem with respect to model piracy. In particular, we observe the potential *structure piracy* and *parameter piracy* threats to deep learning models and propose five metrics to evaluate a deep learning obfuscation solution, namely *cost*, *information leakage*, *fine-tuning ability*, *resilience* to deobfuscation attacks, and *scalability*.

- We propose a novel solution to obfuscate CNN models with *recursive simulation* and *joint training*. Our approach can simulate the feature extractor of a CNN model with a shallow convolutional block, which conceals the structural information of the original network and also deters attackers from fine-tuning an obfuscated model.

- We have verified the feasibility of our approach with several real-world experiments. Our resulting obfuscated models suffer no loss of accuracy, and they are even more efficient than the original models in both model size and inference time.

We organize the rest of the chapter as follows. Section 5.2 briefly reviews the background of CNN. Section 5.3 defines our attack model for model piracy attack. Section 5.4 discusses the deep learning obfuscation problem. Section 5.5 introduces our structural obfuscation approach, and the evaluation is provided in Section 5.6. Section 5.8 compares our work with related work, and Section 5.9 finally concludes this chapter.

5.2 Preliminary

This section briefly reviews the techniques of CNN, which is a preliminary for the deep learning obfuscation approach we propose in this work.

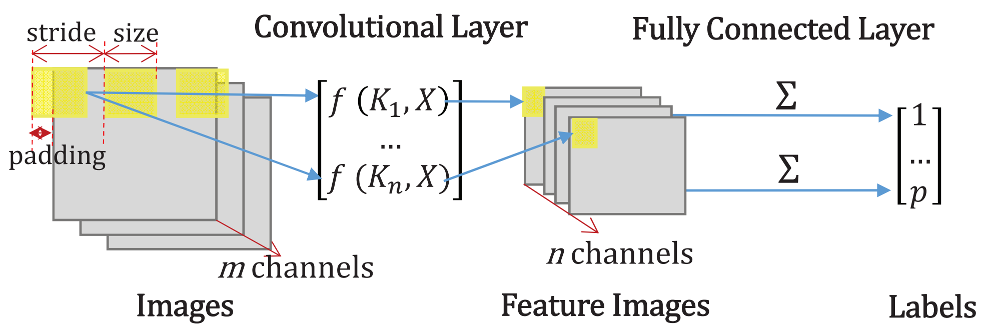


Figure 5.2: Toy example of convolutional neural networks.

5.2.1 CNN Basis

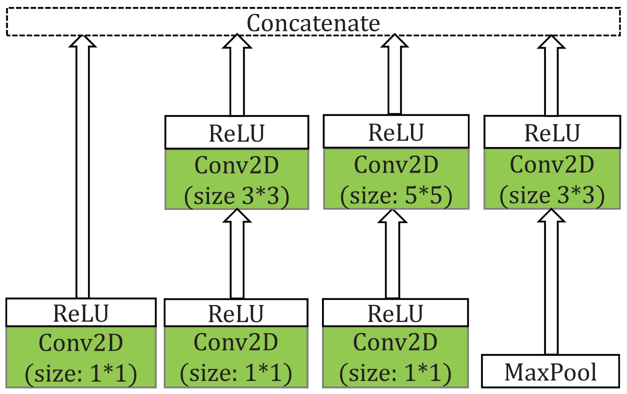
CNN is a special type of deep neural networks that contains convolutional layers and fully-connected layers. The convolutional layers serve as a feature extractor of the network, which inputs images and outputs features. The fully-connected layers serve as a classifier, which classifies images based on the extracted features.

Figure 5.2 demonstrates a simple CNN with one convolutional layer and one fully-connected layer. We discuss the detailed function of either layer in what follows. We use uppercase letters to denote matrices and lowercase letters to denote the elements of a matrix. For example, K is a matrix, and $k_{i,j}$ is an element of the matrix.

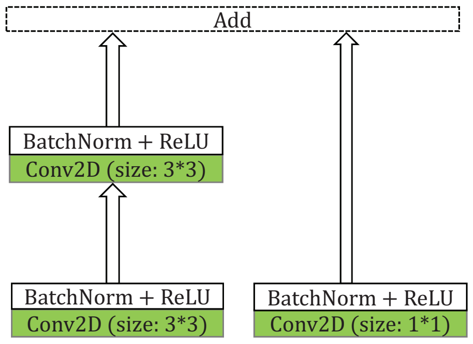
The convolutional layer reads raw images $X \in \mathbb{R}^{h \times w \times m}$ with size $h \times w$ and m channels (*e.g.*, red, green, and blue for colorful images), and outputs n feature images (*i.e.*, n channels). Each output channel corresponding to a convolutional kernel K_i and a bias $\beta_i \quad \forall i \in \{1, \ldots, n\}$ is calculated as follows.

$$\mathcal{F}(K_i, X) = \sum_{j=1}^{m} f(K_i, X_j) + \beta_i, \tag{5.1}$$

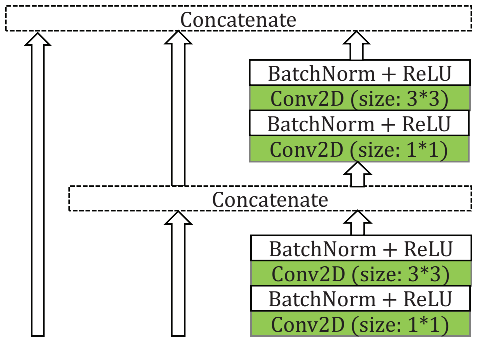
where K_i is a matrix whose elements are the weights of correspond-



(a) Inception block of GoogLeNet.



(b) Sub-block of ResNet for inception.



(c) Partial dense block of DenseNet.

Figure 5.3: Examples of inception blocks.

ing pixels on an image X, and

$$f(K_i, X_j) = \sum_{p=1}^{h} \sum_{q=1}^{w} k_{p,q} \cdot x_{p,q}. \tag{5.2}$$

The fully-connected layer connects each pixel of each feature image to all the class labels. A label with the highest value is voted as the final decision. The formula for computing the value of each label is demonstrated in below.

$$G(X_1, ...X_n) = \sum_{i=1}^{n} \sum_{j=1}^{h \times w} x_{i,j} w_{i,j} + \beta_{i,j}. \tag{5.3}$$

5.2.2 Modern CNNs

While conventional CNNs (*e.g.,* LeNet [107]) only contain different layers organized in sequential orders, the architectures of modern CNNs are more complex. They generally include well-designed inception blocks to facilitate the learning ability. An inception block is a sub-network with convolutional layers and other nonlinear layers, such as batch normalization [92] and ReLU (Rectified Linear Units). These layers are similar to building blocks, and a programmer can organize them in many ways.

Figure 5.3 demonstrates the inception blocks of several popular CNNs, including GoogLeNet, ResNet, and DenseNet. The inception block of GoogLeNet (Figure 5.3(a)) contains four parallel sequences, each of which has a unique convolutional function to learn particular features. All the convolutional sequences output feature tensors of the same size, and the inception block finally concatenates them as its output. ResNet (Figure 5.3(b)) contains two parallel sequences in an inception block. The sequence with a smaller convolutional kernel is designed to propagate the residual information. The design is essential when a neural network goes deeper. The inception block finally adds up the tensors outputted by the two convolutional sequences. DenseNet (Figure 5.3(c)) further improves the mechanism for propagating residual information. In each dense block, the output of every two convolutional layers is propagated to the following layers of the block.

A well-designed feature extractor is a key for a CNN to improve its performance when handling particular tasks, such as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [50]. Note that all our discussed modern CNNs have only one fully-connected layer, and they mainly differ in the feature extraction network. Therefore, hiding the structure of the feature extraction network is the most important concern when obfuscating deep learning models.

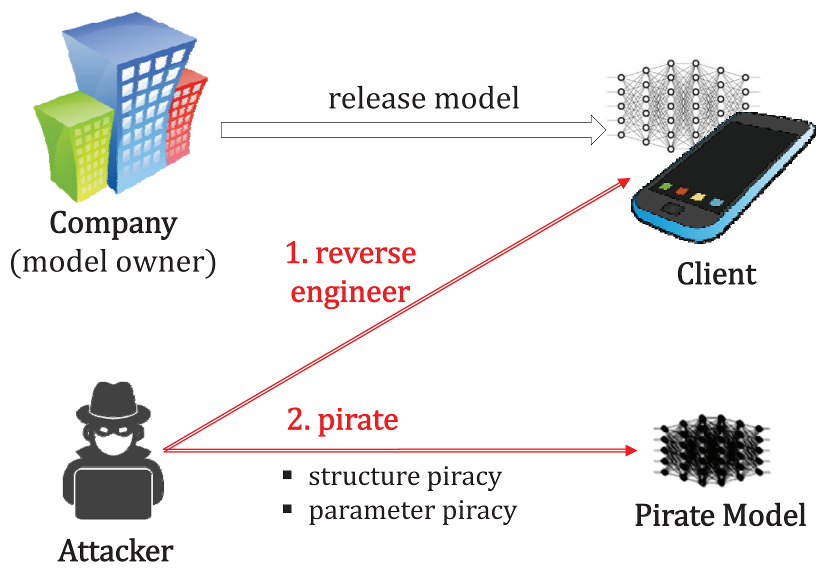
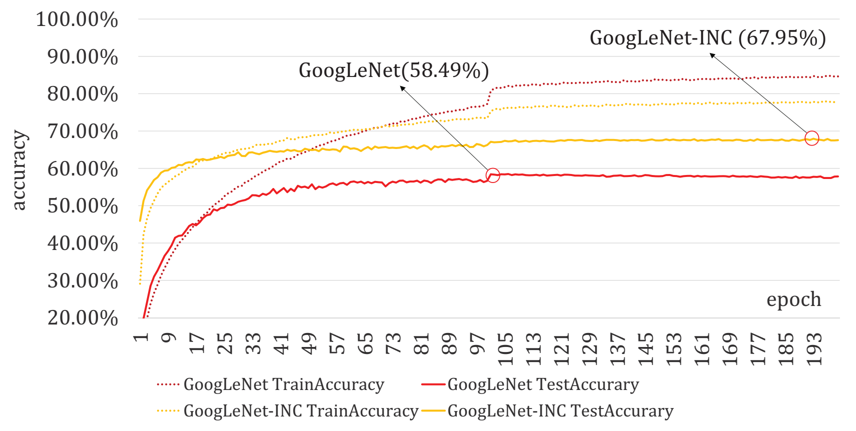


Figure 5.4: Our assumed attack scenario.

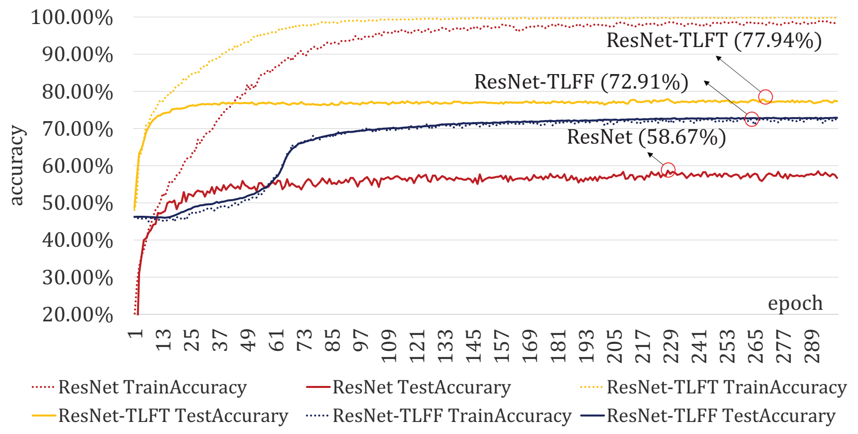
5.3 Attack Model

This work considers the man-at-the-end (MATE) attack [39]. Supposing a well-trained deep learning model has been installed on a client host (*e.g.,* PC server or smartphone) for inference, the MATE attack model assumes that attackers can have full access to the host. If attackers have adequate domain knowledge and determination, they may reverse engineer the application, and then they may pirate the design of the model.

Figure 5.4 demonstrates our attack scenario. We assume that a company (*i.e.,* model owner) has released an application to its users, and the application contains a deep learning model. We further assume that the model contains a novel design of network structure, and it is trained with a large corpus of valuable data. The assumption is very general for real-world deep learning applications because such data are either the private assets of a company or require many manual efforts to get labeled. In the MATE attacker model, attackers (*e.g.,* competitors) may reverse engineer the application and pirate the deep learning model. Such infringement of copyrights would



(a) GoogLeNet (a tailored version with 1024 features) incremental learning with the CIFAR-100 dataset. INC: incremental learning.



(b) ResNet transfer learning with the STL10 dataset. TLFF: transfer learning with frozen feature, TLFT: transfer learning with fine-tuning.

Figure 5.5: Effects of parameter piracy via fine-tuning.

cause loss to the model owner.

In this work, we consider two types of piracy behaviors: *structure piracy* and *parameter piracy*. In structure piracy, attackers reverse engineer an application and extract the model structure, and then they can employ the structure to design their own networks. For CNN models, the distinctive design of structures lies in their feature extractors. Therefore, an effective obfuscation solution should hide such structural information in the obfuscated model. *Parameter piracy* means attackers can employ the well-tuned model states (or parameters) to create new models, *e.g.,* with *incremental learning* [180] and *transfer learning* [76] techniques.

Incremental learning extends the number of classes that a deep learning network can support [180]. Attackers may leverage the technique to empower a model. Figure 5.5(a) demonstrates the effectiveness of a simple incremental learning approach. In this experiment, the original model is trained with the CIFAR-10 dataset, and it supports 10 classes. Our incremental learning program empowers the model to support a new dataset (*i.e.,* CIFAR-100) with 100 classes. We change the last fully-connected layer of the CNN to support 100 classes, and then train the model with the target dataset. Finally, the model can achieve an accuracy of 67.95% after 200 epochs. In comparison, if we train the same CNN from scratch, the best accuracy is only 58.49% with the same training settings.

Transfer learning adapts the domain differences between a target dataset and what a model supports [76]. It is an effective machine learning approach when there are insufficient training data in a target dataset. Attackers may employ the technique to pirate a new model and adapt it to their own scenarios. Figure 5.5(b) demonstrates the effectiveness of a transfer learning experiment. In this experiment, the original model is trained with the CIFAR-10 dataset. The target dataset is STL10[2], which contains only 500 labeled images for each class. If we train the model directly with the target dataset [84]), we only achieve an accuracy of 58.67%. However, if we adopt a straightforward transfer learning technique (*i.e.,* ResNet-TLFT[3] in Figure 5.5(b)), the accuracy can improve to 77.94%.

In the two examples above, we both fine-tune the original feature extractors to achieve a good accuracy. Note that fine-tuning can usually help attackers to obtain a better model than without fine-tuning. For example, in our transfer learning example in Figure 5.5(b), if we freeze the feature extraction layers and only tune the fully-connected

[2]https://cs.stanford.edu/ acoates/stl10/

[3]http://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

layer (*i.e.,* ResNet-TLFF in Figure 5.5(b)), the model accuracy can only achieve 72.91%.

5.4 Deep Learning Obfuscation

5.4.1 Definition

Now we discuss the concept of *deep learning obfuscation*. It aims to transform the inference logic of a well-trained deep learning model to an obfuscated version, which can prevent attackers from learning its structural design or reusing its well-tuned parameters. The obfuscated model should retain an equivalent inference function as the original model contains. Meanwhile, it should incur very limited overhead and demonstrate adequate resilience to attackers.

We borrow the term "obfuscation" from classic code obfuscation problems because their purposes are similar. According to Collberg *et al.* [40], classic code obfuscation scrambles code spinets into unintelligible versions while preserving the semantics. It can increase the difficulties for attackers to interpret the real code logic or to further tamper the code.

However, deep learning obfuscation is a bit different from classic software obfuscation. In particular, the parameters of deep learning models are automatically generated via a training process. To tamper a model, attackers do not need to understand the model, and he can simply fine-tune it with new data. In comparison, classic software is written by programmers. Attackers cannot modify it without understanding the details.

5.4.2 Performance Metrics

A competent deep learning obfuscation approach should demonstrate good performance in the following aspects.

- *Cost*: Cost measures the extra model size and inference time incurred by obfuscation. A practical obfuscation approach should not incur too much overhead. Otherwise, it would be useless for real-world applications.

- *Information leakage*: It shows how much structural information is leaked by the obfuscated model. The metric corresponds to structure piracy issues. A potent obfuscation approach should leak as little information as possible.

- *Fine-tuning ability*: It measures the how much learning ability declination that an obfuscated model suffers. We can measure the metric by comparing the performance of an original model and an obfuscated model when performing the same fine-tuning task. It reflects the resilience of an obfuscated model to the parameter piracy attacks. A network with poor fine-tuning ability can deter attackers from fine-tuning the model parameters with new training data.

- *Resilience*: The metric reflects the resistance of an obfuscated model to deobfuscation attacks. In our scenario, deobfuscation means recovering the original structure of the network or empowering an obfuscated model with better fine-tuning ability.

- *Scalability*: It indicates whether an obfuscation approach can be applied to different deep learning models. In this work, we claim the effectiveness of our obfuscation approach for convolutional neural network only.

Note that resilience evaluates the security of an obfuscation approach, while information leakage and fine-tuning ability evaluate its effectiveness (or potency). Next, we introduce our structural obfuscation approach for protecting CNN models and then evaluate the performance of our obfuscation approach with respect to these metrics.

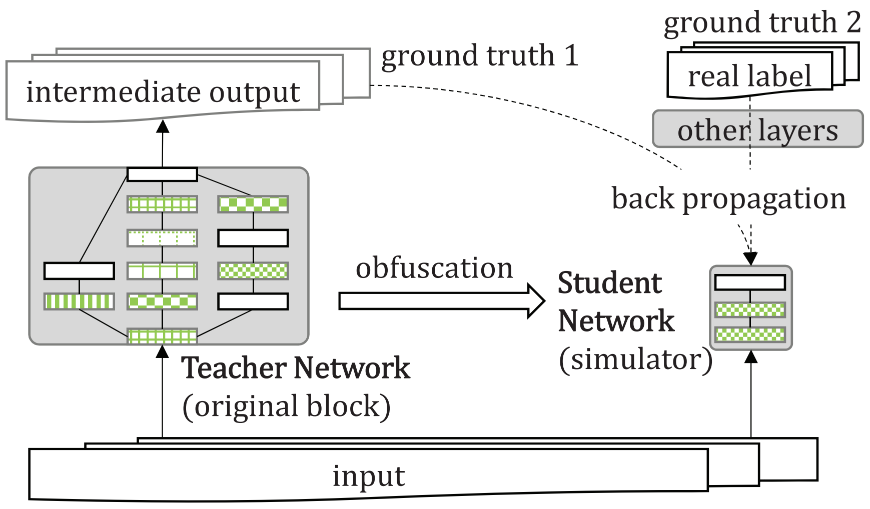


Figure 5.6: Joint-training approach to simulate the feature extractor.

5.5 Structural Obfuscation Approach

For state-of-the-art CNN models, the essential valuable design lies in the structure of their feature extractors. Our structural obfuscation approach, therefore, aims to hide the structures of the feature extraction networks via simulation. Below, we first introduce our basic idea of model simulation and then demonstrate how to employ the idea to obfuscate CNN models.

5.5.1 Basic Idea

In high level, our obfuscation idea simulates the feature extractor of a CNN model with a shallow and sequential network. In this way, the simulation network does not reveal the internal structure of the original feature extractor. Also, by employing a shallow network as the simulator, we intend to lower the fine-tuning ability of the obfuscated model with respect to parameter piracy attacks.

To obtain a competent simulation network, we propose a novel *joint training* idea, which employs two different ground truths to train the simulation network jointly. Our first ground truth is the output of the target feature extractor, which is the output of a hidden layer. A CNN model generally outputs more features in a hidden layer than the number of classes in the output layer. In this way, employing the intermediate output to train a simulation network can provide more fine-grained information than pure labels, and it will facilitate the learning process. The idea is also known as hint-based training [145]. In hint-based training, there is a teacher network and a student network, and the student network learns the intermediate knowledge generated by the hidden layers of the teacher network. However, employing a pure hint-based training approach is insufficient for our scenario because the original hint-based training approach is proposed for model compression scenarios. It compresses a teacher network into a student network which is deep and thin. Our approach, on the other hand, aims to simulate a teacher network with a shallow network. Because a shallow network tends to have worse learning ability, it may not learn the teacher network as well as a deep network. If there are simulation errors, we hope to mitigate the influence of the error with the information of the real label. Therefore, we also employ the label of raw input as our second ground truth for training. Figure 5.6 demonstrates our joint-training idea.

Let $\mathcal{T}$ denote the original feature extractor of a teacher network, $\mathcal{S}$ denotes the simulator of a student network, $\mathcal{S}'$ denotes a mixed network that connects the simulator with the rest layers of the original network, and L_x denotes the real labels of the input data x. We employ the following objective function to train the simulator.

$$\arg\min_{w} \|\mathcal{T}(x) - \mathcal{S}_w(x) + \alpha(L_x - \mathcal{S}'_w(x))\|, \tag{5.4}$$

where $\|\cdot\|$ represents the $L1$-norm, and w represents the parameters of the simulator.

Because the simulation network is shallow, we may not train it as well as the original model. In this situation, the real labels can serve

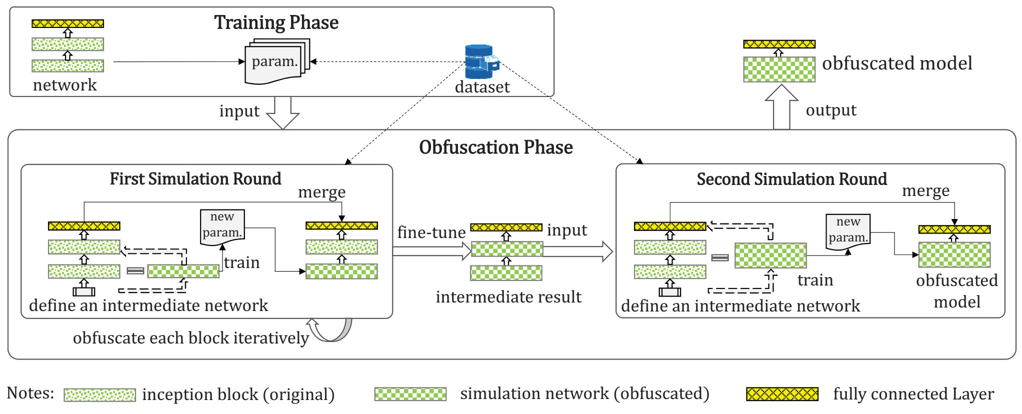


Figure 5.7: Framework to obfuscate CNN models.

as an error corrector or a regulator. By choosing an appropriate value for α, we can balance the resulting impacts to the learning process.

In our later evaluation section, we adopt a tricky approach to search optimal solutions for the joint optimization problem. We assume the two types of losses are comparable with an appropriate α, then we can employ the two ground truths iteratively to train the simulator.

5.5.2 Obfuscation Framework

Because real-world CNNs are very complex, we may not be able to simulate a feature extractor directly. To achieve an acceptable simulation result, we propose to simulate a feature extractor recursively. To elaborate, we first simulate each inception block of an extractor with a small simulator. Then we fine-tune the resulting model to achieve a good accuracy. In the next round, we simulate the new feature extractor which contains several small simulation networks.

The first round of simulation obtains two benefits. On one hand, we can lower the complexity of the target feature extractor to simulate. On the other hand, it examines whether a target model can be simulated with a shallow network. If we cannot simulate a feature extractor with a deep network of several small simulators, we are unlikely to be able to simulate it with one shallower network. After this round, we usually get an intermediate model with an equivalent or slightly higher accuracy.

Our obfuscation framework is demonstrated in Figure 5.7. We insert an obfuscation phase after the training phase and before the model is released to clients. The obfuscation phase contains two rounds. In the first round, we obfuscate each inception block of the model iteratively. The obfuscation starts from the bottom layer and goes up to the top layer. In this order, the residual errors of the lower layers can be mitigated when obfuscating the upper layers. There are several steps in each iteration.

1. *Define a simulation network:* We first create a small simulation network with a shallow and sequential structure. Then we plug the network into the original computation graph of the model and make its input and output the same as those of the original inception block. We defer our discussion about designing the structure of a simulation network to Section 5.5.3.

2. *Train the network*: We tune the parameters of the simulation network with the joint-training idea. The training data is the same as those for training the original model. To preserve the weights of other layers, we freeze all the parameters before and after the simulator.

3. *Merge the model:* We delete the original inception block from the original network and merge the corresponding parameters. This step outputs an intermediate obfuscated model. The obfuscation procedure goes to the next iteration until all the inception blocks have been obfuscated.

When all the inception blocks have been simulated, we fine-tune the model to achieve better accuracy. Now we can obtain a model M' (intermediate result) with only sequential connections, which is the input to the next simulation round. In the next round, we also employ the joint-training approach to simulate the entire feature extractor of M' at a time.

Note that the hint-based training approach requires that the teacher network and the student network should have the same dimension of output. So our intermediate result M' still leaks the information about the interfaces (*e.g.,* feature numbers) between neighboring inception blocks. Our second round of simulation can further hide such information. In this way, the finally obfuscated model leaks little information about the internal structure of a feature extractor.

5.5.3 Design of Simulation Networks

We should design the simulation networks carefully because they determine the performance of an obfuscated model. If the structure of a simulation network is too simple, it may not be able to simulate well. If it is too complicated, the obfuscated model would incur too much overhead.

When designing a simulation network, our main principle is to maintain the corresponding input and output relationships of the original inception block. For example, if a feature of output corresponds to a 5×5 image of input in the original network, we should guarantee that the simulation network also computes the feature based on the 5×5 image. To fulfill the principle, we should choose an appropriate combination of convolutional layers and kernel sizes. Below we elaborate more on the principle.

Warm Up

We first discuss an ideal scenario which assumes no nonlinear operators in an inception block. In this case, we can simulate the inception block precisely with only one convolutional layer. We draw this conjecture based on Rule 5.1 and Rule 5.2.

Rule 5.1. *For any two-layered sequential convolutional block with no nonlinear operators, there exists an equivalent convolutional layer to simulate it.*

Proof. To show that we can simulate any two-layered convolutional layers with only one layer, we show that the feature images outputted by a two-layered convolutional layers are linear to the input.

Supposing the first convolutional layer inputs images X of c_{in} channels and outputs feature images Y of c_{l1} channels, we can compute a pixel value y of Y as

$$y = \sum_{i=1}^{c_{in}} f(K, X_i) + \beta, \tag{5.5}$$

where K is a kernel matrix, X_i is a corresponding image of the ith channel, and $f(K, X_i) = k_{1,1}x_{1,1} + k_{1,2}x_{1,2} + ... + k_{h_{l1},w_{l1}}x_{h_{l1},w_{l1}}$, where $h_{l1} \times w_{l1}$ is the kernel size of the first convolutional layer.

Because K and β are constants in a pre-trained model and only X_i contains variables, we can simplify Equation 5.5 as

$$y = \alpha_1 x_{1,1,1} + ... + \alpha_{c_{l1},h_{l1},w_{l1}} x_{c_{l1},h_{l1},w_{l1}}, \tag{5.6}$$

where α_i is a constant, and y is linear to X.

Similarly, supposing the second convolutional layer outputs feature images Z of c_{l2} channels, we can compute a pixel value z of Z with the form of $z = \alpha_1 y_{1,1,1} + ... + \alpha_{c_{l2},h_{l2},w_{l2}} y_{c_{l1},h_{l1},w_{l1}}$. Unfolding each y, we can get

$$z = \lambda_1 x_{1,1,1} + ... + \lambda_q x_{c_{in},h,w}, \tag{5.7}$$

where λ_i is a constant, $h = h_{l1} + (h_{l2} - 1)s$, $w = w_{l1} + (w_{l2} - 1)s$, and s is the stride of the first convolutional layer. □

Rule 5.2. *For any two-paralleled convolutional layers merged with concatenation or add, there exists an equivalent convolutional layer to simulate it.*

Proof. Suppose the input images are X with c_{in} channels, the first convolutional layer outputs features Y of c_1 channels, and the second convolutional layer outputs features Z of c_2 channels. A pixel value y of Y can be computed as

$$y = \sum_{i=1}^{c_{in}} f(K_y, X_i) + \beta, \tag{5.8}$$

and a pixel value z of Z can be computed as

$$z = \sum_{i=1}^{c_{in}} f(K_z, X_i) + \beta, \tag{5.9}$$

where K_y is a kernel of the first convolutional layer, and K_z is a kernel of the second convolutional layer.

If the layers are merged with concatenation, we can directly substitute the block with an equivalent convolutional layer with a channel size $c_1 + c_2$ and a kernel size $Max(Size(K_y), Size(K_z))$. If the layers are merged with add, the two convolutional layers should have an equivalent number of channels, c_1 and c_2. □

Given any inception block without nonlinear operators, we can apply Rule 5.1 and Rule 5.2 iteratively to compress the module into one convolutional layer. The channel number of the resulting convolutional layer should be equivalent to the original inception block. Its kernel size should be selected as the max kernel size when simulating each convolutional sequence.

Choose an Appropriate Structure

When an inception block has nonlinear operators, we cannot simulate the module precisely. However, we can still apply Rule 5.1 and Rule 5.2 to compute an appropriate kernel size. Take the inception block of ResNet (Figure 5.3(b)) as an example, we can compute that each feature corresponds to a 5×5 image, so that we can simulate the block with one convolutional layer whose kernel size is 5, or with two convolutional layers whose kernel sizes are both 3. If an inception block has pooling operations (*e.g.,* the inception block of GoogLeNet (Figure 5.3(a))), we treat them as same as convolution, because they are also kernel-based operations.

In general, we can design a simulation network with two to four convolutional layers to simulate an inception block. By default, we apply both batch normalization and ReLU after each convolutional layer. The specific choice generally depends on the complexity of the inception block to simulate. For example, to simulate one inception block of GoogleNet, two convolutional layers are sufficient. However, if the target block has tens of convolutional layers, we may employ a simulation network with more convolutional layers. We will discuss more details about how we design the simulation network for several practical deep learning models in the evaluation section (Section 5.6).

5.6 Evaluation

In this section, we examine the performance of our obfuscation approach with real-world experiments. We focus on three aspects: 1) whether an obfuscated model can achieve negligible accuracy declination in comparison with the original model; 2) how much overhead will be incurred by our obfuscation approach; 3) and whether the fine-tuning abilities of obfuscated models become worse.

5.6.1 Experimental Setting

To show that our obfuscation approach has no bias on particular CNN structures, we choose three prevalent CNNs to obfuscate, *i.e.,* GoogLeNet [165], ResNet [84] and DenseNet [91]. We employ public CNNs because they are well-known to the community, and they can achieve good accuracy for image recognition tasks. Although our purpose is to obfuscate private CNNs, the results achieved on public CNNs should also be applicable to private ones. Figure 5.8, 5.9, and 5.10 demonstrate the architecture of the networks we adopted in our experiments, and we train them with the CIFAR-10 dataset. To fit the CNN structures for the dataset, we have slightly changed their original settings.

The CIFAR-10 dataset[4] contains 10 classes of 32×32 colorful images (*e.g.,* airplane, automobile, bird). Each class contains 5000 images for training and 1000 images for testing. The dataset is widely employed to benchmark the performance of CNNs, such as ResNet [84] and DenseNet [91]. We train the CNN models using the suggested settings of learning rates until the testing accuracy cannot get further improved. In our experiment, it takes one thousand epochs each to train a model. Our GoogLeNet model finally reaches an accuracy of 90.83%, ResNet reaches 90.94%, and DenseNet reaches 90.14%. The detailed training results are demonstrated in Figure 5.11(a), Figure 5.12(a), and Figure 5.13(a) respectively.

To conduct obfuscation experiments, we write all of our deep learning programs with Pytorch 3.1 and experimental scripts with Python 3.6. Our experimental host is an Ubuntu (version 16.04) server with a Xeon CPU (E5-2650) and 128G memory, and the experimental GPU is Nvidia Titan Xp Pascal with 12G memory.

5.6.2 Steps of Obfuscation

Simulating GoogLeNet

Figure 5.8 demonstrates our obfuscation steps for GoogLeNet. The original GoogLeNet has nine inception blocks, each of which is demonstrated as Figure 5.3(a). These nine inception blocks are separated into three groups by max pooling operations: $\{2A, 2B\}$, $\{3A, ..., 3E\}$, and $\{4A, 4B\}$.

[4]https://www.cs.toronto.edu/ kriz/cifar.html

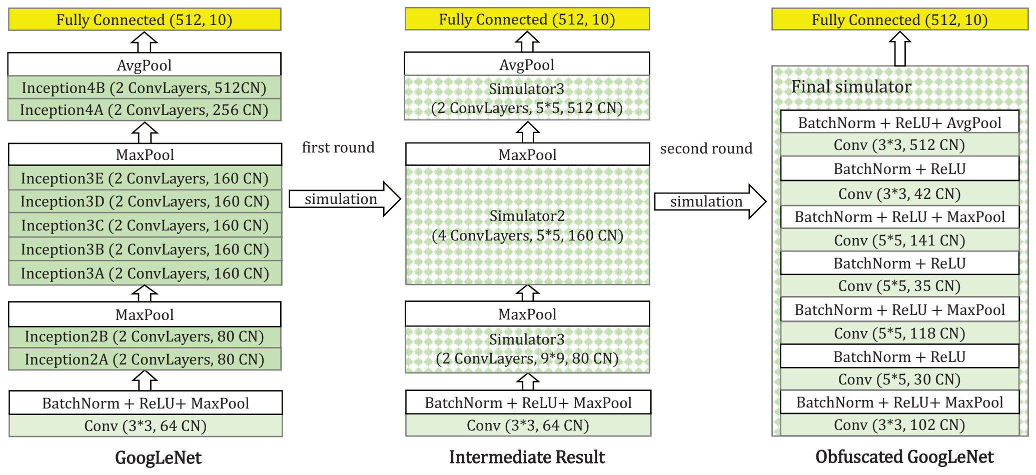

Fully Connected (512, 10)
AvgPool
Inception4B (2 ConvLayers, 512CN)
Inception4A (2 ConvLayers, 256 CN)
MaxPool
Inception3E (2 ConvLayers, 160 CN)
Inception3D (2 ConvLayers, 160 CN)
Inception3C (2 ConvLayers, 160 CN)
Inception3B (2 ConvLayers, 160 CN)
Inception3A (2 ConvLayers, 160 CN)
MaxPool
Inception2B (2 ConvLayers, 80 CN)
Inception2A (2 ConvLayers, 80 CN)
BatchNorm + ReLU+ MaxPool
Conv (3*3, 64 CN)
GoogLeNet
first round
simulation
Fully Connected (512, 10)
AvgPool
Simulator3
(2 ConvLayers, 5*5, 512 CN)
MaxPool
Simulator2
(4 ConvLayers, 5*5, 160 CN)
MaxPool
Simulator3
(2 ConvLayers, 9*9, 80 CN)
BatchNorm + ReLU+ MaxPool
Conv (3*3, 64 CN)
Intermediate Result
second round
simulation
Fully Connected (512, 10)
Final simulator
BatchNorm + ReLU+ AvgPool
Conv (3*3, 512 CN)
BatchNorm + ReLU
Conv (3*3, 42 CN)
BatchNorm + ReLU + MaxPool
Conv (5*5, 141 CN)
BatchNorm + ReLU
Conv (5*5, 35 CN)
BatchNorm + ReLU + MaxPool
Conv (5*5, 118 CN)
BatchNorm + ReLU
Conv (5*5, 30 CN)
BatchNorm + ReLU+ MaxPool
Conv (3*3, 102 CN)
Obfuscated GoogLeNet

Figure 5.8: Procedure to obfuscate GoogLeNet.

We finally obfuscate GoogLeNet with a five-layered sequential block. To this end, we adopt the recursive simulation idea. In the first simulation round, we simulate each group of inception blocks with a sequential block. For the groups of $\{2A, 2B\}$ and $\{4A, 4B\}$, we employ a two-layered convolutional block each. For the group of $\{3A, ..., 3E\}$, we employ a four-layered convolutional block, and for $\{4A, 4B\}$ with a three-layered convolutional block. Then we fine-tune the resulting model to obtain an intermediate result. Note that when simulating each group of inception blocks, we freeze all the parameters of other layers. When fine-tuning the model, we tune all the parameters of the intermediate model. Then in the second simulation round, we employ the fine-tuned intermediate model to train a six-layered convolutional block, which is the feature extractor of our final obfuscated GoogLeNet.

Simulating ResNet

Figure 5.9 demonstrates our process to obfuscate ResNet. We employ ResNet-18, which contains four inception blocks, and each block has two sub-blocks. In the first round, we simulate each inception block of the network at one time with a two-layered

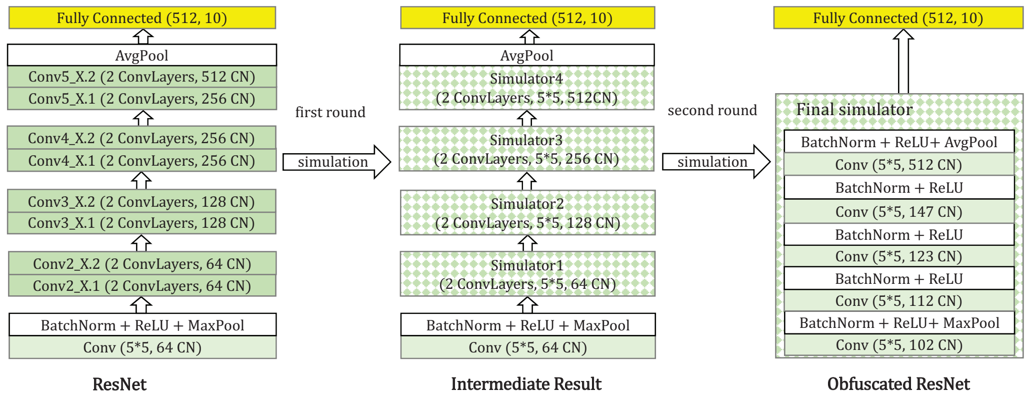

Fully Connected (512, 10)
AvgPool
Conv5_X.2 (2 ConvLayers, 512 CN)
Conv5_X.1 (2 ConvLayers, 256 CN)
Conv4_X.2 (2 ConvLayers, 256 CN)
Conv4_X.1 (2 ConvLayers, 256 CN)
Conv3_X.2 (2 ConvLayers, 128 CN)
Conv3_X.1 (2 ConvLayers, 128 CN)
Conv2_X.2 (2 ConvLayers, 64 CN)
Conv2_X.1 (2 ConvLayers, 64 CN)
BatchNorm + ReLU + MaxPool
Conv (5*5, 64 CN)
ResNet
first round
simulation
Fully Connected (512, 10)
AvgPool
Simulator4 (2 ConvLayers, 5*5, 512CN)
Simulator3 (2 ConvLayers, 5*5, 256 CN)
Simulator2 (2 ConvLayers, 5*5, 128 CN)
Simulator1 (2 ConvLayers, 5*5, 64 CN)
BatchNorm + ReLU + MaxPool
Conv (5*5, 64 CN)
Intermediate Result
second round
simulation
Fully Connected (512, 10)
Final simulator
BatchNorm + ReLU+ AvgPool
Conv (5*5, 512 CN)
BatchNorm + ReLU
Conv (5*5, 147 CN)
BatchNorm + ReLU
Conv (5*5, 123 CN)
BatchNorm + ReLU
Conv (5*5, 112 CN)
BatchNorm + ReLU+ MaxPool
Conv (5*5, 102 CN)
Obfuscated ResNet

Figure 5.9: Procedure to obfuscate ResNet.

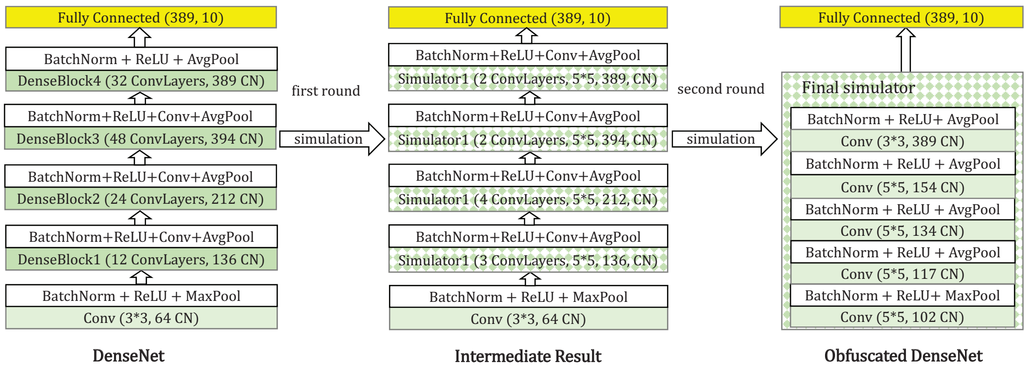
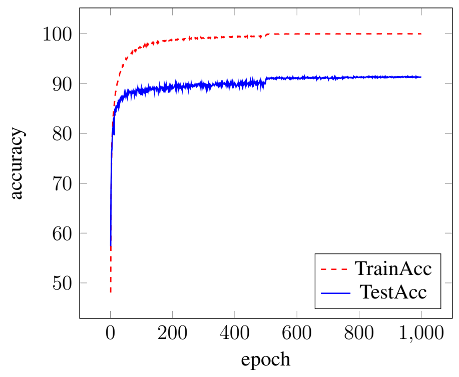

Fully Connected (389, 10)
BatchNorm + ReLU + AvgPool
DenseBlock4 (32 ConvLayers, 389 CN)
BatchNorm+ReLU+Conv+AvgPool
DenseBlock3 (48 ConvLayers, 394 CN)
BatchNorm+ReLU+Conv+AvgPool
DenseBlock2 (24 ConvLayers, 212 CN)
BatchNorm+ReLU+Conv+AvgPool
DenseBlock1 (12 ConvLayers, 136 CN)
BatchNorm + ReLU + MaxPool
Conv (3*3, 64 CN)
DenseNet
first round
simulation
Fully Connected (389, 10)
BatchNorm+ReLU+Conv+AvgPool
Simulator1 (2 ConvLayers, 5*5, 389, CN)
BatchNorm+ReLU+Conv+AvgPool
Simulator1 (2 ConvLayers, 5*5, 394, CN)
BatchNorm+ReLU+Conv+AvgPool
Simulator1 (4 ConvLayers, 5*5, 212, CN)
BatchNorm+ReLU+Conv+AvgPool
Simulator1 (3 ConvLayers, 5*5, 136, CN)
BatchNorm + ReLU + MaxPool
Conv (3*3, 64 CN)
Intermediate Result
second round
simulation
Fully Connected (389, 10)
Final simulator
BatchNorm + ReLU+ AvgPool
Conv (3*3, 389 CN)
BatchNorm + ReLU + AvgPool
Conv (5*5, 154 CN)
BatchNorm + ReLU + AvgPool
Conv (5*5, 134 CN)
BatchNorm + ReLU + AvgPool
Conv (5*5, 117 CN)
BatchNorm + ReLU+ MaxPool
Conv (5*5, 102 CN)
Obfuscated DenseNet

Figure 5.10: Procedure to obfuscate DenseNet.

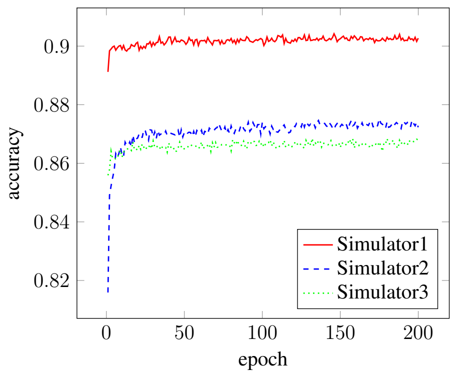
convolutional block. In the second round, we simulate the whole feature extractor of the intermediate model with a five-layered convolutional block.

Simulating DenseNet

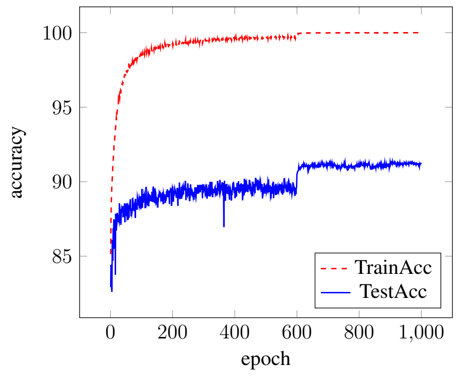
Figure 5.10 shows how we obfuscate DenseNet-121. It has four dense blocks, and each block has 12, 24, 48, or 32 convolutional layers respectively. Their connection mode is demonstrated as Figure 5.3(c). In the first round, we simulate each dense block at one time. The four simulation networks have 3,4,2,2 convolutional layers respectively. In the second round, we simulate the whole feature extractor at one time with a five-layered convolutional block.



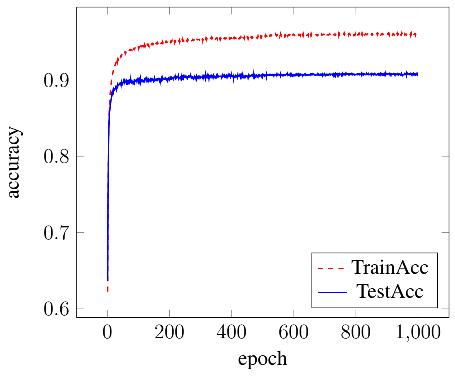
(a) GoogLeNet training process.



(b) The 1st round of GoogLeNet simulation.



(c) Fine-tune the intermediate GoogLeNet.



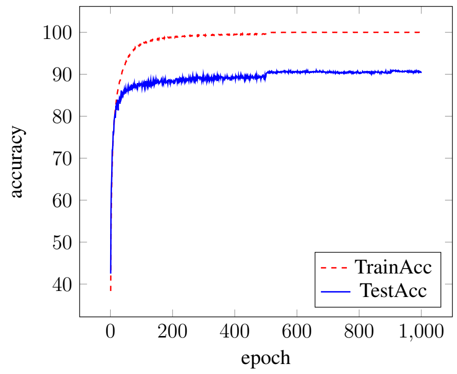
(d) The 2nd round of GoogLeNet simulation.

Figure 5.11: Experimental results for obfuscating GoogLeNet.

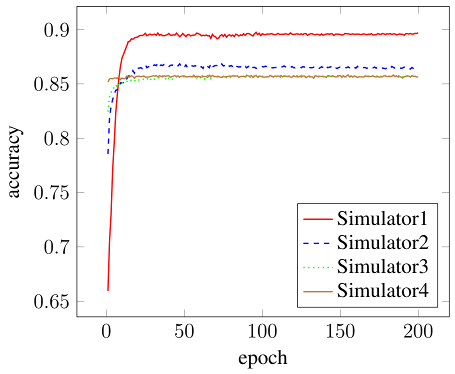
5.6.3 Performance of Obfuscated Models

Table 5.1 demonstrates our experimental results when obfuscating the three models. For each obfuscation experiment, we report the performance of the original model, the intermediate model achieved after the first round of simulation, and the finally obfuscated model. For each of the models, we measure its performance with testing accuracy, model size, and average inference time. From the results, we can observe that all our finally obfuscated models suffer no accuracy declination, and some are even more efficient than the original models.

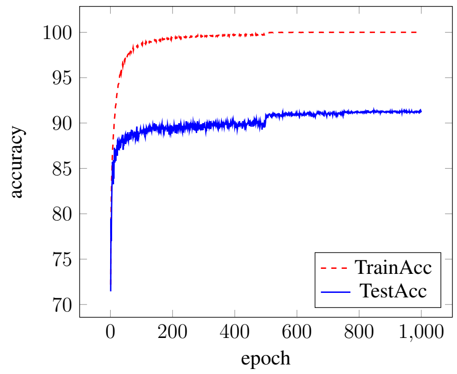
For ResNet, our finally obfuscated model saves 74% size of the



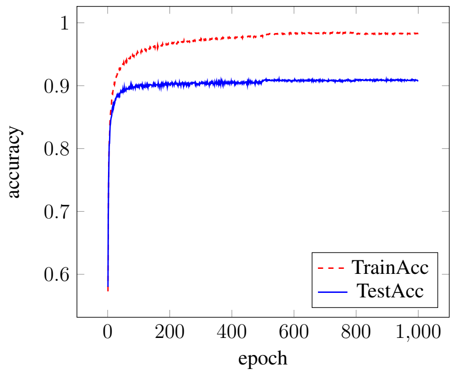
(a) ResNet training process.



(b) The 1st round of ResNet simulation.



(c) Fine-tune the intermediate ResNet.

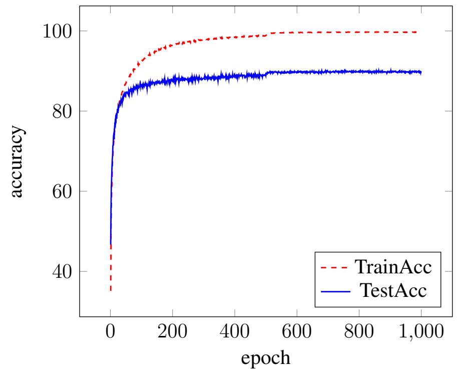


(d) The 2nd round of ResNet simulation.

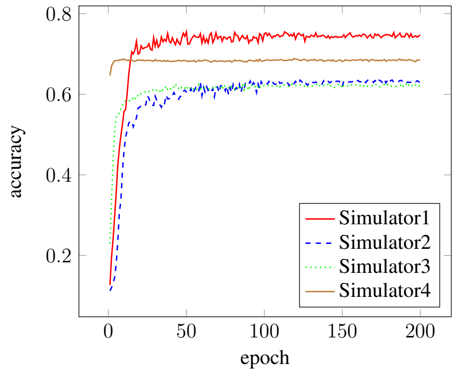
Figure 5.12: Experimental results for obfuscating ResNet.

original model, and it is about two times faster than the original model. This is not very surprising, because ResNet is not very efficient in model size. Note that the inception block of ResNet adds up the results of two parallel convolutional sequences as its output. Therefore, it needs more parameters than a sequential network to compute the same number of features.

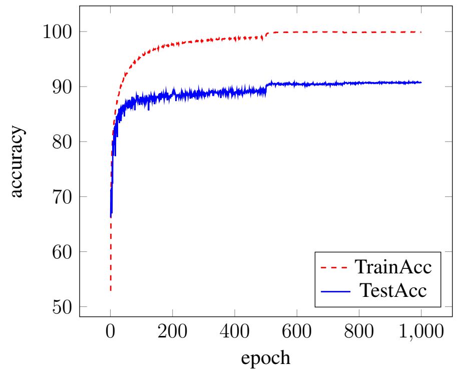
The results show that we can also obfuscate GoogLeNet and DenseNet with no size overhead, and the obfuscated models can save 63% and 84% average inference time correspondingly. This is not easy because the two networks are already very efficient in model size. Their inception blocks concatenate the resulting feature images of parallel convolutional sequences. In this way, they need



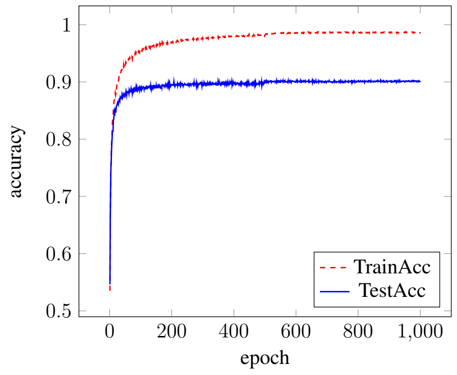
(a) DenseNet training process.



(b) The 1st round of DenseNet simulation.



(c) Fine-tune the intermediate DenseNet.

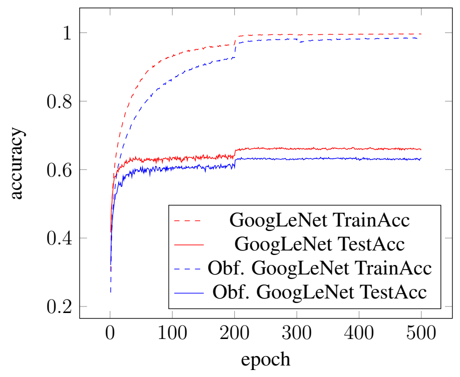


(d) The 2nd round of DenseNet simulation.

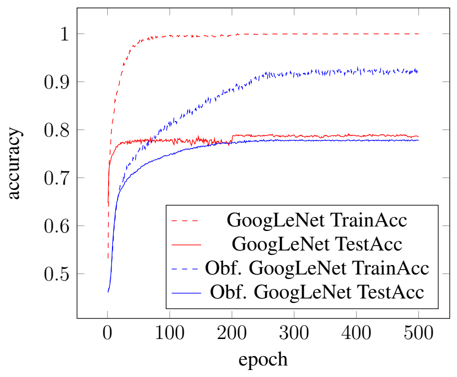
Figure 5.13: Experimental results for obfuscating DenseNet.

fewer parameters than a sequential network to compute the same number of features. Although our intermediate results achieved in the first round incur extra size overhead, such overhead can be fully mitigated after the second round.

Figure 5.11, 5.12, and 5.13 demonstrates more detailed records about each round of our obfuscation experiments. From the figures, we can observe that after the first simulation round, the resulting model generally suffers some accuracy declination. However, we can fine-tune the intermediate model (with the data labels as the ground truth) to achieve better accuracy. This is not because the architecture of the intermediate model is more powerful, but because it employs a good initial state for fine-tuning. Note that

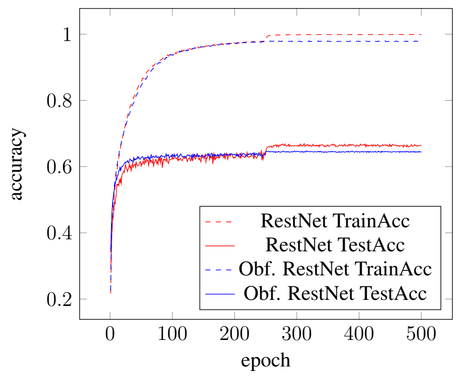


(a) Incremental learning with GoogLeNet.

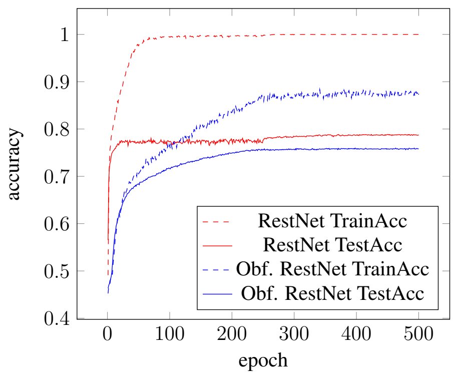


(b) Transfer learning with GoogLeNet.

Figure 5.14: Evaluation results for fine-tuning GoogLeNet.



(a) Incremental learning with ResNet.

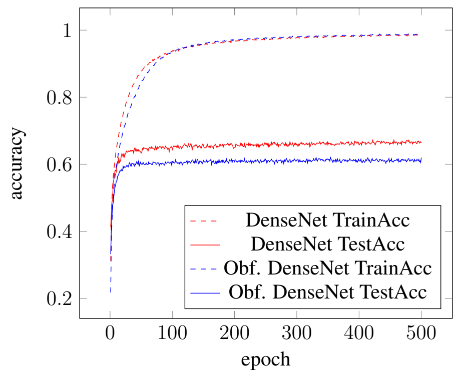


(b) Transfer learning with ResNet.

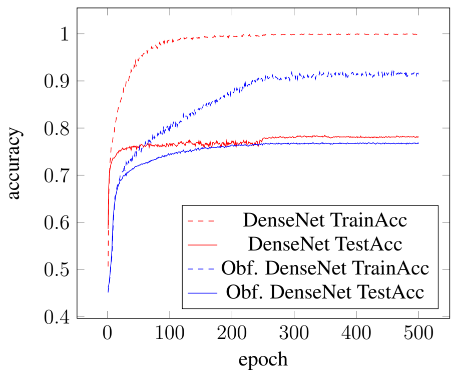
Figure 5.15: Evaluation results for fine-tuning ResNet.

we can hardly improve the accuracy of the original models before. The interesting phenomenon provides some benefits for our second round of simulation. The recursive simulation, therefore, can help us to approach a more competent obfuscation result with better accuracy. In some cases, if we could not obtain an intermediate model with slightly better or at least equivalent accuracy, it implies the simulation network may not be powerful enough, and we should empower the simulation network, such as by adjusting the number of convolutional layers or channels.

To demonstrate that our results have no bias on the dataset, we



(a) Incremental learning with DenseNet.



(b) Transfer learning with DenseNet.

Figure 5.16: Evaluation results for fine-tuning DenseNet.

Table 5.1: Performance of obfuscated models. The overhead is computed as $cost_2/cost_1 - 1$.

Model	**Sim. Round**	**Performance**			**Overhead**	
		acc.	size (MB)	time (μs)	size	time
GoogLe-Net	-	90.83%	2.51	17.85	-	-
	1st	90.99%	7.82	7.69	212%	-59%
	2nd	90.92%	2.49	7.01	-1%	-63%
ResNet	-	90.94%	43.36	10.50	-	-
	1st	91.39%	26.80	6.76	-38%	-36%
	2nd	91.04%	11.38	5.17	-74%	-51%
Dense-Net	-	90.14%	4.24	35.53	-	-
	1st	90.87%	8.86	8.86	109%	-75%
	2nd	90.31%	4.21	5.52	-1%	-84%

repeat the same obfuscation experiments of ResNet and DenseNet with another dataset, which contains five classes randomly selected from ImageNet [50]. Because the image size of ImageNet is 224 $\times$224, we slightly tune the network structures to fit the format. Table 5.2 presents our experimental results. In general, the results are consistent with those using the CIFAR-10. The obfuscated models are as accurate as the original models, and they incur no overhead. Note that the results of some size overhead are different from those on CIFAR-10 because the format of images in ImageNet requires a different configuration of kernel sizes during simulation.

Table 5.2: Evaluation results of our obfuscated models using ImageNet as the dataset.

Model	**Sim. Round**	**Performance**			**Overhead**	
		acc.	size (MB)	time (μs)	size	time
ResNet	-	92.4%	43.37	89	-	-
	1st	92.4%	26.81	60	-38%	-33%
	2nd	92.4%	36.72	59	-15%	-34%
Dense -Net	-	91.6%	4.27	154	-	-
	1st	93.2%	8.89	72	108%	-53%
	2nd	92.8%	2.94	56	-31%	-64%

Table 5.3: Evaluation results of fine-tuning abilities. The declination is computed as $1 - accuracy_2/accuracy_1$.

Network	**CIFAR-100**		**STL10**	
	accuracy	decline	accuracy	decline
GoogLeNet	66.5%	-	79.15%	-
Obf. GoogLeNet	63.59%	4.4%	77.95%	1.5%
ResNet	66.92%	-	78.86%	-
Obf. ResNet	64.77%	3.2%	75.97%	3.7%
DenseNet	67.16%	-	78.45%	-
Obf. DenseNet	62.91%	6.3%	76.90%	2.0%

We ignore the details here since our configuration strategy simply follows the principle discussed in Section 5.5.3.

5.6.4 Fine-tuning Ability

To evaluate the fine-tuning ability of each obfuscated model, we conduct an incremental learning experiment with the CIFAR-100 dataset, and a transfer learning experiment with the STL10 dataset. The CIFAR-100 dataset has 100 classes of images, and each class contains 500 images for training and 100 images for testing. The STL10 dataset has 10 classes of images, which is the same as CIFAR-10. However, the original image format of STL10 is 96×96, and we have to resize the images to 32×32 to fit our model. Besides, there are also only 500 images for training in each class. Because the training dataset of CIFAR-100 and STL10 are both 10 times smaller than CIFAR-10 for each class, we may not achieve an accuracy as good as CIFAR-10. Employing transfer learning and incremental learning techniques should be helpful to achieve better accuracy.

Table 5.3 demonstrates our experimental results. As a comparison, we also report the corresponding performance of the original model. Overall, the obfuscated models suffer obvious accuracy declination when performing both the incremental learning and transfer learning tasks, ranging from 1.5% to 6.3%. Our detailed experimental records are demonstrated in Figure 5.14, 5.14, and 5.14. To compare the performance fair, we employ the same training strategy for each model and its obfuscated model. The experimental results verify our idea that an obfuscated model with shallow structure should have worse fine-tuning ability than its original model.

In particular, the declination degrees are related to specific fine-tuning tasks and the structures of simulation networks. In our experiment, the declinations of the incremental learning experiments are more obvious than those of the transfer learning experiments. We think one reason is that the incremental learning task is more difficult than transfer learning task because it requires distinguishing $10\times$ classes. The fine-tuning ability declination may further exaggerate when pursuing more difficult tasks. Therefore, the metric of fine-tuning ability is related to particular tasks, and such results only provide developers a relative measurement about the resilience of obfuscated models to fine-tuning attacks. In practice, one may try several different simulation networks and choose one with the best performance and the worst fine-tuning ability as the final solution.

5.6.5 Discussion

Besides the cost and fine-tuning ability which we have already evaluated, a competent obfuscation approach should also perform well concerning information leakage, resilience, and scalability. Below, we discuss the performance of our obfuscation approach in these aspects.

Information Leakage

Our obfuscation approach conceals the internal structures of the feature extraction network. We simulate the entire feature extractor of a CNN model with a simulation network. In this way, the obfuscated model exposes no information about the original feature extraction network expect the interfaces of input and output, including the raw image size and the number of extracted features. Besides, the fully-connected layer also leaks the number of classes supported. However, when attackers want to pirate a deep learning model, such information is not very critical. The most important information lies in the design of the feature extraction network, which has already been protected by our approach.

Resilience to Deobfuscation

We discuss the resilience of our approach in two aspects. Firstly, can attackers recover the structure of an original network? The answer is no. Attackers cannot figure out the original structure of a feature extractor, because such information is not retained within the obfuscated model.

Secondly, can attackers empower an obfuscated model with respect to learning? One possible way might be employing another powerful network to simulate the obfuscated model. However, there are two barriers for attackers to launch such attacks. One barrier is that attackers should know a powerful network, and another barrier is that they should have high-quality data to train the simulation network. Note that in our attacker model, we assume the original training data are unavailable to attackers. Without the training data, attackers can only tune the new kernel parameters with other artificial datasets (*e.g.,* some randomly generated images).

Scalability

Currently, we have only evaluated the feasibility of our approach for several CNN models. Therefore, we claim the effectiveness of our solution for CNN models only. However, we may extend the idea to other types of neural networks, such as recurrent neural networks. This is a direction of our future work.

For CNN models, the performance of our obfuscation approach could be related to the complexity of a target model, such as the depth of the network to simulate and the knowledge it contains. In this chapter, we have verified that our approach can effectively obfuscate a well-trained DenseNet-121 model, which is already very deep with more than a hundred layers. Therefore, our simulation-based approach is very promising to obfuscate different convolutional neural networks. Besides, several recent processes (*e.g.,* [80, 81]) achieved in model compression area also adopt simulation-based ideas, and their results also coincide the scalability of our approach.

However, our approach may not be able to obfuscate a model efficiently if it is already very efficient. For example, we can hardly obfuscate a compressed model with no cost, because such models memorize knowledge efficiently with only a small number of parameters. In that case, we may apply obfuscation techniques on an uncompressed model first and then compress the obfuscated model.

5.7 Comparison with Model Compression

We obfuscate deep learning models via a simulation-based approach. The similar approaches are also employed in model compression work (*e.g.,* [51, 80, 81, 85, 108]). However, since obfuscation has a different purpose from compression, their priorities in designing the simulation network are different. Note that obfuscation intends to conceal the structure of the neural network and degrade its learning ability by making the simulation network much shallower. But these two goals are not considered by compression. As a result, the security effectiveness of model compression against piracy attacks is very limited. Below, we discuss more details about existing model compression techniques.

Some investigations compress a well-trained model without changing its structure, such as network pruning [81, 108], and quantization [80]. Network pruning compresses well-trained CNN models by prune the less important connections and weights. A quantization-based approach employs fewer bits for each parameter of the model. They all show that the compressed models after fine-tuning suffer no declination of accuracy.

There also other compression approaches which intend to compress the structure of the network. They try to distill the knowledge learned by the large and cumbersome network, and install the knowledge into another small model [85]. FitNets [145] is a representative work in this area. To compress a model, FitNets simulates a deep learning model with another network that is deeper and thinner. Our work, on the other hand, attempts to simulate a model with a shallow network. This is because the purpose of our work is different from model compression. As a result, we may have different priorities when designing the simulation networks. Specifically, we purposely hide the structure of a model and degrade its fine-tuning ability. Therefore, employing a deeper simulation network is not our prior choice, because a deep network is likely to have good learning abilities. On the other hand, model compression aims to compress a model with high efficiency, while security is not a concern. FitNets proposes a hint-based training approach to train the simulation networks. Our joint-training approach also incorporates the idea, while it is an enhancement of joint-based training.

In brief, model compression is a technique orthogonal to ours.

Although our approach and existing model compression investigations share some common technical background concerning network simulation, they are different in both purposes and detailed designs. We may further employ model compression techniques to compress an obfuscated model in our future work.

5.8 Related Work

In this work, we study model piracy threats and the deep learning obfuscation problem. To our best knowledge, it is a pilot study in this area. There are other investigations which also focus on the security issues of deep learning models (*e.g.,* [86, 126, 167], but their attack models and purposes are very different from ours. For example, Hitaj *et al.* [86] and Tramer *et al.* [167] studied the information leakage issue of deep learning models. Nguyen *et al.* [126] studied how to generate images to fool deep learning models.

5.9 Conclusion

In this chapter, we have discussed the piracy threats to deep learning models, and we propose to obfuscate such models before releasing them to clients. To our best knowledge, it is a first attempt to investigate the deep learning obfuscation problem. In particular, we have discussed the concept of deep learning obfuscation and the performance metrics for evaluating a competent obfuscation solution. Moreover, we propose a structural obfuscation approach for obfuscating CNN models. Our obfuscation approach simulates the feature extractor of a CNN model with a shallow and sequential convolutional block. We train the simulation network incorporating a novel joint-training method and a recursive training method. We have verified the effectiveness of our approach with three prevalent CNNs, and the results show that we can obfuscate their structures with no declination of accuracy. Furthermore, the obfuscated models can be even more efficient than their original models. As a security benefit, our obfuscated model leaks no critical information about internal structure of the original CNN network. In the meanwhile, by choosing a shallow simulation network with a poor fine-tuning ability, the obfuscated model can be resilient to parameter piracy attacks.

□ **End of chapter.**

Chapter 6

Related Work

This chapter summarizes the novelty of the thesis and compares with related work.

This thesis aims to obfuscate software with layered security. To facilitate developers in adopting the idea. We proposed a taxonomy hierarchy to categorize software obfuscation techniques. It is the first taxonomy that considers the complicated nature of current software. In this way, it is different from other existing obfuscation surveys (*e.g.,* [7, 56, 118, 146, 151]) which generally consider obfuscation techniques in code-component layer.

Next, we enriched the taxonomy with three novel obfuscation techniques: symbolic opaque predicates, N-version obfuscation, and deep learning obfuscation.

Symbolic opaque predicates is an original work that systematically discussed how to compose opaque predicates leveraging the challenges faced by symbolic execution techniques. It is different from other work (*e.g.,* [176]) which tries to employ particular problems to secure opaque predicates. Our approach is more general and is not limited to any particular challenging problems. Moreover, we proposed a novel logic bomb-based approach to benchmark symbolic execution tools accurately and efficiently. Some existing papers may also include systematizations of such challenges (*e.g.,* [9, 46, 94, 138]. However, our investigated challenge categories are more complete, and our benchmarking approach is more general than them.

N-version obfuscation employs program diversities to impede large-scale software tampering attacks. We addressed the threats of both static software repacking and dynamic injection. In comparison, existing program diversification approaches [89] are mainly resilient to dynamic injection.

Our deep learning obfuscation work is a pilot study for obfuscating deep learning models. It is the first solution for protecting deep learning models against structure piracy and parameter piracy attacks. Our approach shares some similar basis with model compression (e.g., [85, 145]) in knowledge distillation, but their purposes are very different.

□ **End of chapter.**

Chapter 7

Conclusion and Future Work

This chapter concludes the thesis and discusses our future work.

7.1 Summary of Thesis

In this thesis, we have presented the idea of layered obfuscation, which extends the classic layered security concept to software obfuscation area. We have justified the validity of the idea, and we believe it should be a promising way towards reliable obfuscation in the future.

To promote the idea, we have achieved contributions on two folds. Firstly, we have developed a novel taxonomy for present obfuscation techniques. Our taxonomy follows the idea of layered obfuscation and can assist developers in choosing and integrating various obfuscation techniques. Secondly, we have enriched the taxonomy of obfuscation with three novel techniques. These techniques provide developers more options in designing layered obfuscation solutions. Our first obfuscation technique is bi-opaque predicates, which enables current control-flow obfuscation techniques with resilience to attackers with symbolic executions. Our second technique is n-version obfuscation, which can impede large-scale software tampering attacks by introducing diversities among software instances. Our third technique is deepobfuscation, which focus on protecting private deep learning models with a simulation-based approach.

To summarize, this thesis lays a foundation for layered obfuscation, and we believe it is important to the development of software obfuscation area. On one hand, it presents that the critical path towards achieving reliable obfuscation is layered security. On the other hand, it relaxes the evaluation requirements for obfuscation whereas an obfuscation technique should not be secure-against-all but only mitigates some specific risks. In this way, it may inspire more obfuscation techniques in the future.

7.2 Future Work

Our future work will mainly focus on practicing layered obfuscation with real-world software, such as mobile apps, and deep learning software. We plan to develop a whole methodology for practicing the idea, from software risk analysis and to obfuscation solutions. In this process, we believe we will find some information that cannot be obfuscated well with present obfuscation techniques and some techniques that will become vulnerable as attackers are evolving. We will generalize such attacker models and investigate new obfuscation techniques to handle such problems.

In the meanwhile, we plan to develop practical obfuscation tools for conducting large-scale real-world experiments. Currently, the novel obfuscation techniques we discussed in this thesis only have prototypes. We will improve those prototypes with the capability and compatibility to handle large projects. Such tasks require a complete design of obfuscation tools and involve many issues in software engineering fields.

□ **End of chapter.**

Appendix A

Publications Related to the Thesis

1. "Benchmarking the Capability of Symbolic Execution Tools," **Hui Xu**, Zirui Zhao, Yangfan Zhou, and Michael R. Lyu, IEEE Transactions on Dependable and Secure Computing, 2018.

2. "Manufacturing Resilient Bi-Opaque Predicates against Symbolic Execution," **Hui Xu**, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael R. Lyu, in *Proc. of the 48th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

3. "Concolic Execution on Small-Size Binary Codes: Challenges and Empirical Study," **Hui Xu**, Yangfan Zhou, Yu Kang, and Michael R. Lyu, in *Proc. of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

4. "Assessing the Security Properties of Software Obfuscation," **Hui Xu**, and Michael R. Lyu, in *IEEE Security & Privacy Magazine*, Oct, 2016.

5. "N-Version Obfuscation," **Hui Xu**, Yangfan Zhou, and Michael R. Lyu, in *Proc. of the 2nd Cyber-Phsical System Security Workshop (in conjunction with AsiaCCS)*, 2016.

6. "DeepObfuscation: Obfuscating Deep Neural Networks via

Knowledge Distillation," **Hui Xu**, Yuxin Su, Zirui Zhao, Yangfan Zhou, Michael R. Lyu, and Irwin King, *under review*.

7. "Layered Obfuscation: A Taxonomy of Software Obfuscation Techniques for Layered Security," **Hui Xu**, Jiang Ming, Yangfan Zhou, and Michael R. Lyu, *under review*.

□ **End of chapter.**

Bibliography

[1] D. Apon, Y. Huang, J. Katz, and A. J. Malozemoff. Implementing cryptographic program obfuscation. *IACR Cryptology ePrint Archive*, 2014.

[2] A. Appel. Deobfuscation is in np. *Princeton University, Aug*, 2002.

[3] G. Arboit. A method for watermarking java programs via opaque predicates. In *The 5th International Conference on Electronic Commerce Research*, 2002.

[4] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 2014.

[5] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proc. of the 36th International Conference on Software Engineering (ICSE)*. ACM, 2014.

[6] V. Balachandran and S. Emmanuel. Software code obfuscation by hiding control flow information in stack. In *Proc. of the IEEE International Workshop on Information Forensics and Security*, 2011.

[7] A. Balakrishnan and C. Schulze. Code obfuscation literature survey. *CS701 Construction of Compilers*, 2005.

[8] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Survey (CSUR)*, 2018.

[9] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*. ACM, 2016.

[10] S. Banescu, M. Ochoa, and A. Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In *Proc. of the 1st IEEE/ACM International Workshop on Software Protection*, 2015.

[11] B. Barak. Hopes, fears, and software obfuscation. *Communications of the ACM*, 2016.

[12] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology*. Springer, 2001.

[13] J. K. Barr, B. A. Bradley, B. T. Hannigan, A. M. Alattar, and R. Durst. Layered security in digital watermarking, 2012. US Patent 8,190,901.

[14] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC)*, 2005.

[15] D. A. Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC1. In *Proc. of the 18th Annual ACM Symposium on Theory of Computing (STOC)*, 1986.

[16] T. Bergan, D. Grossman, and L. Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. 2014.

[17] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, 2003.

[18] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev. Statistical deobfuscation of android applications. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[19] F. Biondi, S. Josse, A. Legay, and T. Sirvent. Effectiveness of synthesis in concolic deobfuscation. 2015.

[20] D. Bohannon and L. Holmes. Revoke-obfuscation: powershell obfuscation detection using science. BlackHat, 2017.

[21] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[22] J.-M. Borello and L. Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 2008.

[23] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability (STVR)*, 2006.

[24] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *Proc. of the International Conference on Computer Aided Verification*. Springer, 2011.

[25] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008.

[26] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex

systems programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[27] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 2013.

[28] J. Cappaert and B. Preneel. A general model for hiding control flow. In *Proc. of the 10th Annual ACM Workshop on Digital Rights Management*, 2010.

[29] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*, 2012.

[30] J.-T. Chan and W. Yang. Advanced obfuscation techniques for Java bytecode. *Journal of Systems and Software*, 2004.

[31] H. Chang and M. J. Atallah. Protecting software code by guards. In *Security and Privacy in Digital Rights Management*. Springer, 2002.

[32] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-c. Yew. Control flow obfuscation with information flow tracking. In *Proc. of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[33] L. Chen and A. Avizienis. N-version programming: a fault-tolerance approach to reliability of software operation. In *Proc. the 8th IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, 1978.

[34] Y. Chen, R. Venkatesan, et al. Oblivious hashing: A stealthy software integrity verification primitive. In *Information Hiding*. Springer, 2003.

[35] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. 2011.

[36] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot. White-box cryptography and an aes implementation. In *Proc. of the International Workshop on Selected Areas in Cryptography*. Springer, 2002.

[37] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot. A white-box DES implementation for DRM applications. In *ACM Workshop on Digital Rights Management*. Springer, 2002.

[38] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Information Security*. Springer, 2001.

[39] C. Collberg, J. Davidson, R. Giacobazzi, Y. X. Gu, A. Herzberg, and F.-Y. Wang. Toward digital asset protection. *IEEE Intelligent Systems*, 2011.

[40] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, The University of Auckland, 1997.

[41] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proc. of the IEEE International Conference on Computer Languages*, 1998.

[42] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1998.

[43] R. Corin and F. A. Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. *ESSoS*, 2011.

[44] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, 2015.

[45] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a TRaP: table randomization and protection against function-reuse attacks. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[46] L. Cseppento and Z. Micskei. Evaluating symbolic execution-based test tools. In *Proc. of the IEEE 8th International Conference on Software Testing, Verification and Validation*, 2015.

[47] M. Dalla Preda and F. Maggi. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*, 2017.

[48] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, 2013.

[49] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[50] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proc. of the 2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[51] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for

efficient evaluation. In *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[52] S. Dolan. mov is Turing-complete, 2013.

[53] D. Dolz and G. Parra. Using exception handling to build opaque predicates in intermediate code obfuscation techniques. *Journal of Computer Science and Technology (JCST)*, 2008.

[54] C. Domas. The movfuscator: Turning 'move' into a soul-crushing RE nightmare. REcon, 2015.

[55] S. Drape et al. *Obfuscation of abstract data types*. Citeseer, 2004.

[56] S. Drape et al. Intellectual property protection using obfuscation. *Proc. of SAS*, 2009.

[57] D. Dunaev and L. Lengyel. Complexity of a special deobfuscation problem. In *Proc. of the 19th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, 2012.

[58] E. Eilam. *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[59] L. Ertaul and S. Venkatesh. Jhide-a tool kit for code obfuscation. In *IASTED Conf. on Software Engineering and Applications*, 2004.

[60] L. Ertaul and S. Venkatesh. Novel obfuscation algorithms for software security. In *Proc. of the 2005 International Conference on Software Engineering Research and Practice*. Citeseer, 2005.

[61] P. Faruki, V. Laxmi, V. Ganmoor, M. S. Gaur, and A. Bharmal. Droidolytics: robust feature signature for repackaged

android apps on official and third party android markets. In *Proc. of the 2nd IEEE International Conference on Advanced Computing, Networking and Security*, 2013.

[62] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *Proc. of the 9th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, 2013.

[63] C. Foket, B. De Sutter, B. Coppens, and K. De Bosschere. A novel obfuscation: class hierarchy flattening. In *International Symposium on Foundations and Practice of Security*. Springer, 2012.

[64] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proc. of the 6th IEEE Workshop on Hot Topics in Operating Systems*, 1997.

[65] K. Fukushima, S. Kiyomoto, T. Tanaka, and K. Sakurai. Analysis of program obfuscation schemes with variable encoding technique. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 2008.

[66] K. Fukushima, T. Tabata, and K. Sakurai. Evaluation of obfuscation scheme focusing on calling relationships of fields and methods in methods. *Communication, Network, and Information Security*, 2003.

[67] M. N. Gagnon, S. Taylor, and A. K. Ghosh. Software protection through anti-debugging. 2007.

[68] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. of the International Conference on Computer Aided Verification*. Springer, 2007.

[69] Y. Ganin and V. Lempitsky. Unsupervised domain adaptation by backpropagation. In *International Conference on Machine Learning (ICML)*, 2015.

[70] S. Garg, C. Gentry, and S. Halevi. Candidate multilinear maps from ideal lattices. In *Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013.

[71] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proc. of the 54th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2013.

[72] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits (full version). In *Cryptology ePrint Archive*, 2013.

[73] J. Ge, S. Chaudhuri, and A. Tyagi. Control flow based obfuscation. In *Proc. of the 5th ACM Workshop on Digital Rights Management*, 2005.

[74] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM Sigplan Notices*, 2005.

[75] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 1991.

[76] B. Gong, Y. Shi, F. Sha, and K. Grauman. Geodesic flow kernel for unsupervised domain adaptation. In *Proc. of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[77] Y. Guillot and A. Gazet. Automatic binary deobfuscation. *Journal in Computer Virology*, 2010.

[78] S. Guo, M. Kusano, and C. Wang. Conc-iSE: Incremental symbolic execution of concurrent software. In *Proc. of*

the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), 2016.

[79] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *Proc. of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015.

[80] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. 2016.

[81] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[82] W. A. Harrison and K. I. Magel. A complexity measure based on nesting level. *ACM Sigplan Notices*, 1981.

[83] N. Hasabnis and R. Sekar. Extracting instruction semantics via symbolic execution of code generators. In *Proc. of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[84] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[85] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2014.

[86] B. Hitaj, G. Ateniese, and F. Pérez-Cruz. Deep models under the GAN: information leakage from collaborative deep learning. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[87] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.

[88] M. Horváth and L. Buttyán. The birth of cryptographic obfuscation-a survey. 2016.

[89] S. Hosseinzadeh, S. Rauti, S. Laurén, J.-M. Mäkelä, J. Holvitie, S. Hyrynsalmi, and V. Leppänen. Diversification and obfuscation techniques for software security: a systematic literature review. *Information and Software Technology*, 2018.

[90] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[91] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten. Densely connected convolutional networks. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[92] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning (ICML)*, 2015.

[93] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM: software protection for the masses. 2015.

[94] R. Kannavara, C. J. Havlicek, B. Chen, M. R. Tuttle, K. Cong, S. Ray, and F. Xie. Challenges and opportunities with concolic testing. In *Proc. of the National Aerospace and Electronics Conference*. IEEE, 2015.

[95] P. Khodamoradi, M. Fazlali, F. Mardukhi, and M. Nosrati. Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms. In

Proc. of the 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS). IEEE, 2015.

[96] J. Kilian. Founding crytpography on oblivious transfer. In *Proc. of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, 1988.

[97] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha. Testing intermediate representations for binary analysis. In *Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[98] A. Kovacheva. Efficient code obfuscation for android. In *International Conference on Advances in Information Technology*. Springer, 2013.

[99] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. *ACM Sigplan Notices*, 2012.

[100] N. Kuzurin, A. Shokurov, N. Varnovsky, and V. Zakharov. On the concept of software obfuscation in computer security. In *Information Security*. Springer, 2007.

[101] J. C. Lagarias. The 3x + 1 problem and its generalizations. *The American Mathematical Monthly*, 1985.

[102] W. Landi and B. G. Ryder. Pointer-induced aliasing: a problem classification. In *Proc. of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1991.

[103] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: a software accelerator for low-power deep learning inference on mobile devices. In *Proc. of the 15th ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2016.

[104] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy*, 2014.

[105] T. László and A. Kiss. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 2009.

[106] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society, 2004.

[107] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems (NIPS)*, 1990.

[108] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Advances in Neural Information Processing Systems (NIPS)*, 1990.

[109] K. Lewi, A. J. Malozemoff, D. Apon, B. Carmer, A. Foltzer, D. Wagner, D. W. Archer, D. Boneh, J. Katz, and M. Raykova. 5gen: A framework for prototyping applications using multilinear maps and matrix branching programs. In *Proc. of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[110] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle. Floating-point symbolic execution: A case study in N-version programming. In *Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[111] D. S. Liew. Symbolic execution of verification languages and floating-point code. 2018.

[112] Z. Lin, R. D. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2009.

[113] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. of the 10th ACM Conference on Computer and Communications Security (CCS)*, 2003.

[114] D. Low. Protecting java code via code obfuscation. *Crossroads*, 1998.

[115] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, 2005.

[116] M. R. Lyu and Y.-T. He. Improving the n-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability (TR)*, 1993.

[117] A. Majumdar and C. Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Proc. of the 29th Australasian Computer Science Conference*. Australian Computer Society, Inc., 2006.

[118] A. Majumdar, C. Thomborson, and S. Drape. A survey of control-flow obfuscations. In *Information Systems Security*. Springer, 2006.

[119] A. Marcelli, E. Sanchez, G. Squillerò, M. U. Jamal, A. Imtiaz, S. Machetti, F. Mangani, P. Monti, D. Pola, A. Salvato, et al. Defeating hardware trojan in microprocessor cores

through software obfuscation. In *the 19th Latin-American Test Symposium*. IEEE, 2018.

[120] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. In *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[121] T. J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering (TSE)*, 1976.

[122] J. Ming, D. Xu, L. Wang, and D. Wu. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[123] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proc. of the 23rd IEEE Annual Computer Security Applications Conference (ACSAC)*, 2007.

[124] G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 2006.

[125] N. Nethercote. *Dynamic binary analysis and instrumentation*. PhD thesis, PhD thesis, University of Cambridge, 2004.

[126] A. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[127] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, 2003.

[128] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proc. of the 16th IEEE Annual Computer Security Applications Conference (ACSAC)*, 2000.

[129] N. Papernot, M. Abadi, U. Erlingsson, I. Goodfellow, and K. Talwar. Semi-supervised knowledge transfer for deep learning from private training data. *Proc. of the 4th International Conference on Learning Representations (ICLR)*, 2016.

[130] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proc. of the 2017 ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2017.

[131] O. M. Parkhi, A. Vedaldi, A. Zisserman, et al. Deep face recognition. In *BMVC*, 2015.

[132] A. Pawlowski, M. Contag, and T. Holz. Probfuscation: an obfuscation approach using probabilistic control flows. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016.

[133] K. Pei, Y. Cao, J. Yang, and S. Jana. DeepXplore: automated whitebox testing of deep learning systems. *Proc. of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[134] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Proc. of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[135] M. Protsenko and T. Muller. Pandora applies non-deterministic obfuscation randomly to android. In *Proc. of the 8th International Conference on Malicious and Unwanted Software*. IEEE, 2013.

[136] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su. Identifying and understanding self-checksumming defenses in software. 2015.

[137] J. Qiu, B. Yadegari, B. Johannesmeyer, et al. A framework for understanding dynamic anti-analysis defenses. In *Proc. of the 4th ACM Program Protection and Reverse Engineering Workshop*, 2014.

[138] X. Qu and B. Robinson. A case study of concolic testing tools and their limitations. In *Proc. of the IEEE International Symposium on Empirical Software Engineering and Measurement*, 2011.

[139] M. Quan. Hotspot symbolic execution of floating-point programs. In *Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[140] N. Razavi, F. Ivančić, V. Kahlon, and A. Gupta. Concurrent test generation using concolic multi-trace analysis. In *Asian Symposium on Programming Languages and Systems (APLAS)*. Springer, 2012.

[141] S.-A. Rebuffi, A. Kolesnikov, G. Sperl, and C. H. Lampert. icarl: Incremental classifier and representation learning. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[142] C. Ren, K. Chen, and P. Liu. Droidmarking: Resilient software watermarking for impeding android application repackaging. In *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014.

[143] E. Rescorla. *SSL and TLS: designing and building secure systems*. Addison-Wesley Reading, 2001.

[144] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2012.

[145] A. Romero, S. E. Kahou, and Y. Bengio. Fitnets: Hints for thin deep nets. 2015.

[146] K. A. Roundy and B. P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR)*, 2013.

[147] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*. Springer, 1998.

[148] F. Saudel and J. Salwan. Triton: a dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de linformation et des communications, SSTIC, France, Rennes*, 2015.

[149] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proc. of the eighteenth International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2009.

[150] S. Schrittwieser and S. Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding*. Springer, 2011.

[151] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)*, 2016.

[152] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward

symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[153] B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 1996.

[154] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, 2005.

[155] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proc. of the 30th ACM Annual Computer Security Applications Conference (ACSAC)*, 2014.

[156] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[157] T. Shields. Anti-debugging: a developers view, 2010.

[158] Y. Shoshitaishvili and *et al.* SoK: (State of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[159] J. Shu, J. Li, Y. Zhang, and D. Gu. Android app protection via interpretation obfuscation. In *Proc. of the 12th IEEE International Conference on Dependable, Autonomic and Secure Computing*, 2014.

[160] C. Simonyi. Hungarian notation. *MSDN Library*, 1999.

[161] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *International Symposium on Formal Methods*. Springer, 2015.

[162] M. Sosonkin, G. Naumovich, and N. Memon. Obfuscation of design intent in object-oriented applications. In *Proc. of the Third ACM workshop on Digital Rights Management*, 2003.

[163] T. Su, Z. Fu, G. Pu, J. He, and Z. Su. Combining symbolic execution and model checking for data flow testing. In *Proc. of the 37th International Conference on Software Engineering (ICSE)*. IEEE Press, 2015.

[164] M. Sun, M. Li, and J. Lui. Droideagle: seamless detection of visually similar android apps. In *Proc. of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2015.

[165] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[166] R. Tiella and M. Ceccato. Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation. In *Proc. of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017.

[167] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction APIs. In *USENIX Security Symposium*, 2016.

[168] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *Proc. of the 12th IEEE Working Conference on Reverse Engineering*, 2005.

[169] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proc. of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2001.

[170] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, University of Virginia, 2000.

[171] H. Wang, Y. Guo, Z. Ma, and X. Chen. Wukong: a scalable and accurate two-phase approach to android app clone detection. In *Proc. of the ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[172] P. Wang, Q. Bao, L. Wang, S. Wang, Z. Chen, T. Wei, and D. Wu. Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *Proc. of the 40th International Conference on Software Engineering (ICSE)*, 2018.

[173] P. Wang and J. Cheng. Accelerating convolutional neural networks for mobile applications. In *Proc. of the 2016 ACM Multimedia Conference*, 2016.

[174] P. Wang, S. Wang, J. Ming, Y. Jiang, and D. Wu. Translingual obfuscation. In *IEEE European Symposium on Security and Privacy*, 2016.

[175] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy*, 2010.

[176] Z. Wang, J. Ming, C. Jia, and D. Gao. Linear obfuscation to combat symbolic execution. In *ESORICS*. Springer, 2011.

[177] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl. A large scale investigation of obfuscation use in google play. *arXiv preprint arXiv:1801.02742*, 2018.

[178] G. Wroblewski. *General method of program code obfuscation*. PhD thesis, Wroclaw University of Technology, 2002.

[179] G. Wurster, P. V. Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *IEEE Symposium on Security and Privacy*, 2005.

[180] T. Xiao, J. Zhang, K. Yang, Y. Peng, and Z. Zhang. Error-driven incremental learning in deep convolutional neural network for large-scale image classification. In *Proc. of the 22nd ACM International Conference on Multimedia*, 2014.

[181] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. of the 39th IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, 2009.

[182] Z. Xin, H. Chen, H. Han, B. Mao, and L. Xie. Misleading malware similarities analysis by automatic data structure obfuscation. In *Information Security*. Springer, 2010.

[183] D. Xu, J. Ming, and D. Wu. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *IEEE Symposium on Security and Privacy*, 2017.

[184] H. Xu, Y. Su, Z. Zhao, Y. Zhou, M. R. Lyu, and I. King. DeepObfuscation: Securing the structure of convolutional neural networks via knowledge distillation. *arXiv preprint arXiv:1806.10313*, 2018.

[185] H. Xu, Y. Zhou, C. Gao, Y. Kang, and M. R. Lyu. Spyaware: Investigating the privacy leakage signatures in app execution traces. In *Proc. of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

[186] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu. Concolic execution on small-size binaries: challenges and empirical study. In *Proc. of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[187] H. Xu, Y. Zhou, and M. R. Lyu. N-version obfuscation. In *Proc. of the 2nd ACM International Workshop on Cyber-Physical System Security (CPSS)*, 2016.

[188] B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[189] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *IEEE Symposium on Security and Privacy*, 2015.

[190] M. Yildiz, J. Abawajy, T. Ercan, and A. Bernoth. A layered security approach for cloud computing infrastructure. In *International Symposium on Pervasive Systems, Algorithms, and Networks*. IEEE, 2009.

[191] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems (NIPS)*, 2014.

[192] I. You and K. Yim. Malware obfuscation techniques: a brief survey. In *Proc. of International Conference on Broadband, Wireless Computing, Communication and Applications*, 2010.

[193] X. Zhang, F. He, and W. Zuo. *Theory and practice of program obfuscation*. INTECH Open Access Publisher, 2010.

[194] W. Zhou, Z. Wang, Y. Zhou, and X. Jiang. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *Proc. of the 4th ACM Conference on Data and Application Security and Privacy*, 2014.

[195] W. Zhou, X. Zhang, and X. Jiang. Appink: watermarking android apps for repackaging deterrence. In *Proc. of the 8th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2013.

[196] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. of the 2nd ACM Conference on Data and Application Security and Privacy*, 2012.

[197] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE Symposium on Security and Privacy*, 2012.

[198] W. Zhu and C. Thomborson. A provable scheme for homomorphic obfuscation in software security. In *The IASTED International Conference on Communication, Network and Information Security*, 2005.

[199] W. Zhu, C. Thomborson, and F.-Y. Wang. Applications of homomorphic functions to software obfuscation. In *Intelligence and Security Informatics*. Springer, 2006.

[200] W. F. Zhu. *Concepts and techniques in software watermarking and obfuscation*. PhD thesis, ResearchSpace, Auckland, 2007.

[201] J. Zimmerman. How to obfuscate programs directly. In *Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015.