The Pennsylvania State University

The Graduate School

ADVANCED SOFTWARE OBFUSCATION TECHNIQUES AND APPLICATIONS

A Dissertation in

Information Sciences and Technology

by

Pei Wang

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

August 2018

The dissertation of Pei Wang was reviewed and approved* by the following:

Dinghao Wu
Associate Professor of Information Sciences and Technology
Dissertation Advisor, Chair of Committee

Peng Liu
Professor of Information Sciences and Technology

Sencun Zhu
Associate Professor of Information Sciences and Technology
Associate Professor of Computer Science and Engineering

Danfeng Zhang
Assistant Professor of Computer Science and Engineering

Andrea Tapia
Associate Professor of Information Sciences and Technology
Director of Graduate Programs

*Signatures are on file in the Graduate School.

Abstract

Obfuscation is an important software protection technique that prevents automated or human analyses from revealing the internal design and implementation details of software. There has been a strong demand for advanced obfuscation techniques from software vendors to confront threats like intellectual property thefts and cybersecurity attacks. This dissertations approaches the software protection problem through obfuscation in three different aspects, i.e., techniques, applications, and experiences.

The dissertation first introduces translingual obfuscation, a novel software obfuscation technique that makes programs obscure by "misusing" certain features of programming languages derived from highly abstract computation theories. For programs written in imperative languages, which are popular but relatively easy to reverse engineer, translingual obfuscation translates part of a program to another language which has a much more complicated programming paradigm and execution model, thus increasing program complexity. The evaluation shows that this advanced obfuscation technique is suitable for protecting software in desktop and server computation environments. It provides effective and stealthy obfuscation effects with only modest performance cost, compared to one of the most popular commercial obfuscators on the market.

As for mobile software, its development, deployment, and execution are significantly different from those of traditional desktop software, while must less is known about the practice of software protection on this emerging platform. Therefore, the dissertation takes a first step to systematically studying the applications of software obfuscation techniques in mobile app development. With the help of an automated but coarse-grained method, we computed the likelihood of an app being obfuscated for over a million app samples crawled from Apple App Store.

We then inspected the top 6600 most likely obfuscated instances and managed to identify 601 obfuscated versions of 539 iOS apps. By analyzing this sample set with intensive manual effort, we made various observations that help reveal the status quo of mobile obfuscation in the real world. As such, the dissertation can provide insights into understanding and improving the situation of software protection on mobile platforms.

Finally, the dissertation reports field experience of applying obfuscation to multiple commercial mobile apps, each of which serves millions of users. In this case study, we leveraged the knowledge learned from the empirical study. The dissertation discusses the challenges of software obfuscation on the iOS platform and our efforts in overcoming these obstacles. This report can benefit many stakeholders in the mobile ecosystem, including developers, security service providers, and administrators of mobile software ecosystems such as Apple and Google.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Dr. Dinghao Wu for his invaluable mentorship. The knowledge and skills I learned from him not only supported my Ph.D. study but will constantly help me make new progress in my future research career.

I would also like to thank Dr. Wei Tao and Mr. Zhaofeng Chen for their help during my internship at Baidu X-Lab. A significant portion of the research presented in this dissertation would not happen without their input and collaboration.

I was lucky enough to work with a group of talented colleagues in Penn State. I would not be able to achieve many of the results without the thoughtful and inspiring discussions with them.

Finally, I would like to express my gratitude to my parents for supporting my research career in a foreign country, far from my homeland. Thanks to them, I can focus on my Ph.D. study for years without other concerns.

The dissertation is based on research supported in part by the National Science Foundation (NSF) under grants CNS-1223710, CCF-1320605, and CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-13-1-0175, N00014-16-1-2912, N00014-16-1-2265, and N00014-17-1-2894.[1]

[1]Findings and conclusions in this dissertation do not necessarily reflect the view of the funding agencies.

Chapter 1

Introduction

Software protection is critical in the software industry. According to a study [23] by the Software Alliance, software piracy leaded to 52.2 billion dollars of unlicensed software installation on desktop and server computers, in 2015. Since software piracy is usually linked to malicious incidents, a total loss of 400 billion dollars were caused, directly and indirectly, by software piracy in the same year. On mobile platforms, the situation is even more severe. Concerns on security breaches targeting mobile apps have kept rising in past years. It was reported that the piracy rates of popular mobile apps can approach to 60–95% [21]. Research by Gibler et al. found that a surprisingly large portion of mobile applications are "copies" of others [73].

This introductory chapter discusses the motivation of the research in the dissertation. In particular, we discuss the role of obfuscation techniques in software protection and explain why advanced obfuscation techniques and their appropriate deployment are critical to software developers. The chapter also summarizes the contributions of the dissertation.

1.1 Demand for Software Protection

Besides these traditional intellectual property theft problems, the industry is also facing new security threats as there are now many businesses heavily relying on mobile devices to operate across the globe. In most cases, mobile software is easier to reverse engineer than desktop software. Although, from the research point of view, there exist various challenges in automated reverse engineering that cannot be easily addressed, leading to beliefs that reverse engineering is not a realistic threat to common mobile software vendors. In reality, however, many of such challenges can be practically addressed or circumvented

Take iOS apps as examples. Since most iOS apps are built with the standard toolchain provided by Apple, the shapes of their binary code are utterly uniform. This is a highly desired situation for reverse engineering. By analyzing the common code patterns and developing corresponding analysis heuristics, modern binary analysis tools have grown reasonably proficient at decompiling iOS apps, making reverse engineering much less laborious than before. Figure 1.1 is an example that demonstrates the quality of the decompilation result for a popular open source iOS app. The decompilation is done by IDA Pro [15], the most widely used reverse engineering toolkit in industry. As can be seen, the generated pseudocode is almost identical to the original source code, except for the language implementation details which are implicit in the source code but recovered by the decompiler, e.g., the `self` pointer. To experienced reverse engineers, these differences are negligible.

In addition to the support of increasingly mature analysis tools, reverse engineering is made even more effective on iOS due to its development and production environment. The majority of iOS apps are written in Objective-C, a C-like, object-oriented, and fully reflexive programming language developed by Apple. In Objective-C, method names are called *selectors* and method invocations are implemented in a message forwarding scheme. When a method is called on an object, the language runtime will dynamically walk through the dispatch table of the class

```
@implementation TSAnimatedAdapter
...

- (BOOL)canPerformEditingAction:(SEL)action {
    return (action == @selector(copy:)
            || action == NSSelectorFromString(@"save:"));
}

...
@end
```

(a) Original Objective-C source code

```
// TSAnimatedAdapter - (bool)canPerformEditingAction:(SEL)
bool __cdecl -[TSAnimatedAdapter canPerformEditingAction:]
(struct TSAnimatedAdapter *self, SEL a2, SEL a3) {
  bool result;
  if ( "copy:" == a3 )
    result = 1;
  else
    result = NSSelectorFromString(CFSTR("save:")) == (_QWORD)a3;
  return result;
}
```

(b) Pseudocode obtained from decompiling the binary

Figure 1.1: Decompiling an open source iOS app [14] with IDA Pro

of the object to find a method implementation whose name matches with the selector. If no match is found, the runtime will repeat the procedure on the object's base class. This process is similar to the prototype system of JavaScript, except that the method name matching in Objective-C is conducted through class metadata while the similar process in JavaScript is conducted through the properties of each object. Naturally, the message forwarding scheme requires the Objective-C compiler to preserve all method names in program binaries. Method names are extremely useful information when analyzing large software binaries, for it allows human analysts to infer program semantics and quickly identify critical points worth in-depth inspection among a huge amount of code.

On the Android platform, there is a similar problem since Java is also a fully reflexive language. Having realized the potential risks, Google integrated a method and class name scrambler into the Android development toolchain [24]. In contrast, iOS developers do not get any support from Apple, leaving all code completely unprotected by default. Furthermore, Apple now advises iOS developers to submit apps in the form of LLVM intermediate representation, which is even less challenging to analyze than ARM machine code. Overall, reverse engineering iOS apps can be made very effective if developers do not take actions of prevention.

The lack of technical challenges in analyzing unprotected mobile apps grants adversaries strong reverse engineering capabilities and allows their malevolent attempts of exploiting mobile apps for illegal benefits to succeed with a high chance. App-specific vulnerabilities can certainly be devastating if their presences are learned by attackers. For example, a previous version of Uber's mobile app was found vulnerable and therefore can be exploited to get unlimited free rides [1]. On the other hand, besides those specialized threats, there also exist attacks that are generally applicable to many apps. We describe three common kinds of them, besides the typical intellectual property theft problem.

Man-in-the-middle attacks. By tricking users into connecting mobile devices to untrusted wireless networks or installing SSL certificates from unknown sources, attackers can intercept and counterfeit the communication between apps and servers [80]. After analyzing how apps process the data exchanged with servers, attackers can potentially control app behavior by forging certain server responses.

Repackaging. It has been reported that some cybercrime groups are able to reverse engineer popular social networking apps and weaponize them for stealing sensitive user information [48]. By developing information-stealing modules and repackaging them into genuine apps, attackers managed to create malicious mobile software with seemingly benign appearances and functionality. Contacts, chat logs, web browsing histories, and voice recordings are common targets of theft.

Fraud, spam, and malicious campaigns. Nowadays, many apps employ anomaly detection to identify suspicious client activities and prevent incidents like fraud, spam, and malicious campaigns. This is usually achieved through collecting necessary information about users and their devices and fitting the collected data into anomaly detection models. Since the data are harvested on device, attackers can reverse engineer the mobile apps and find out what kinds of data are being collected. In this way, they may be able to mimic normal user behavior by fabricating false data of the same kinds on rooted and jailbroken devices.

During the past few years, our industry collaborators have encountered many incidents of categories described above, among which the most concerning ones are the increasingly prevalent large-scale malicious campaigns. According to a report on fraudulent campaigns conducted in China [5], the business of "click farming" has formed a billion-dollar underground economy, in which hundreds of well organized collusive groups have participated. The technological means used to support these campaigns are also evolving quickly. Campaign runners can now programmatically control hundreds of mobile devices without the involvement of human labor, while

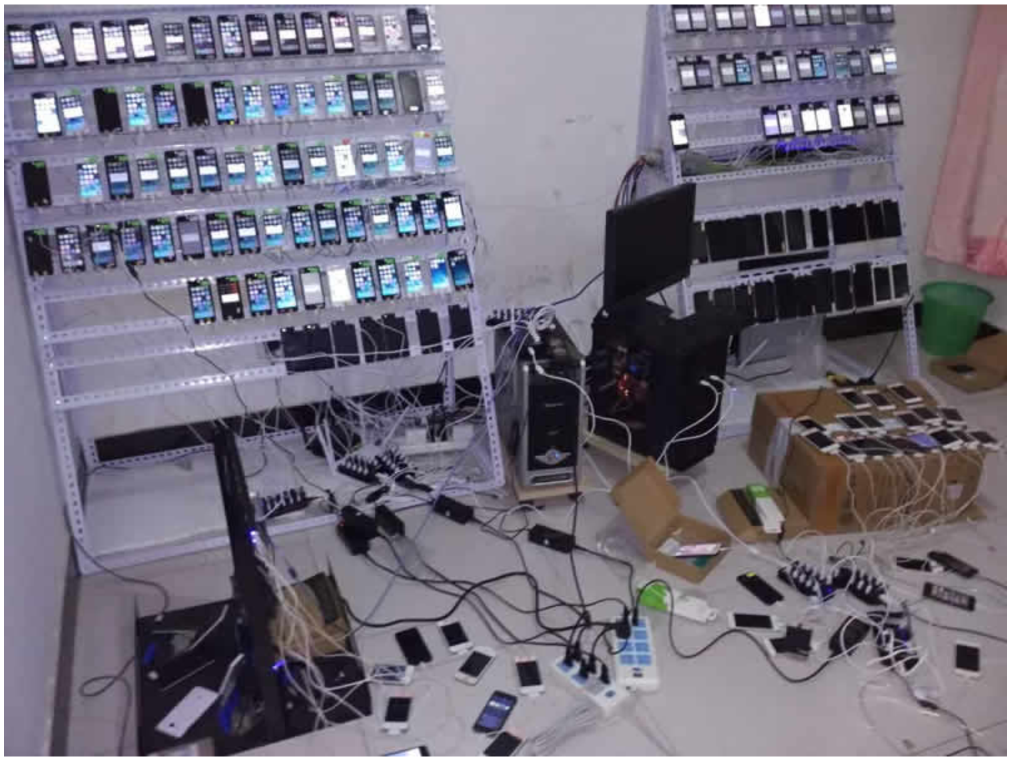


Figure 1.2: Programmatically controlling massive iOS devices as a service (http://shemeitong.com/index.php/anli/show/46.html).

each device can host over 50 instances of the same mobile app. Figure1.2 shows an example of such technology.

Since the third quarter of 2016, our collaborators in the mobile software industry have captured a large-volume of suspicious activities being conducted around the resources and services offered to mobile app users. Through information cross-validation, we detected that there are millions of suspicious iOS devices, many of which are virtually faked, constantly trying to log into the account system of the apps, committing massive promotion operations like clicking links to a certain product, posting comments to a certain page, and exhaustively collecting bonuses provided to daily active users. Many of these activities have violated end user terms and affected the quality of the services.

To detect the malicious campaigns and nullify their impacts, app developers need to precisely identify those bot-like users through extensive data analysis. Since data collection must strictly respect user privacy, only certain types of data can be collected for this purpose, which attackers can easily guess out. For the sake of data genuineness, we have to ensure that malicious groups cannot tamper with the on-device data collection process through reverse engineering the corresponding program logic, which requires effective software protection techniques to be deployed.

1.2 Advanced Software Obfuscation Techniques

In general, obfuscation takes effect by hindering the program analysis capabilities of adversaries, which are the basics of many malicious activities targeting commodity software. There are two types of obfuscation techniques, the first of which originates from the research of cryptography and seeks to build mathematically hard-to-analyze programs. Different notions have been proposed to formally define effective obfuscation. In general, however, theoretically secure obfuscation algorithms are either impossible to craft or too expensive to be used for protecting software in production, depending which notion is considered. The second type of software obfuscation is based on heuristics rather than theoretical theorems. Typically, heuristic-base obfuscation are semantics-preserving program transformations that aim to make a program more difficult to understand and reverse engineer. In contrast to theoretical obfuscation, heuristic obfuscation has no guaranteed resilience and is potentially vulnerable to unknown attack methods. On the other hand, it is much more practical than obfuscation based on cryptography constructs and has been employed in real-world software development. The idea of using obfuscating transformations to prevent reverse engineering can be traced back to Collberg et al. [51, 52, 108]. Since then many obfuscation methods have been proposed [94, 106, 116, 125, 45, 145]. On the other hand, malware authors also heavily rely on obfuscation to compress or encrypt executable binaries so that their products can avoid malicious content detection [129, 127].

Currently the state-of-the-art obfuscation technique is to incorporate with *process-level virtualization*. For example, obfuscators such as VMProtect [28] and Code Virtualizer [7] replace the original binary code with new bytecode, and a custom interpreter is attached to interpret and execute the bytecode. The result is that the original binary code does not exist anymore, leaving only the bytecode and interpreter, making it difficult to directly reverse engineer [77]. However, recent work has shown that the decode-and-dispatch execution pattern of virtualization-based obfuscation can be a severe vulnerability leading to effective deobfuscation [54, 126], implying that we are in need of obfuscation techniques based on new schemes.

To help the software production community withstand the threats from malicious reverse engineering, this dissertation delivers research results that can advance the status quo of software protection by obfuscation. The research is two fold. Firstly, we propose a novel and practical obfuscation method called *translingual obfuscation*, which possesses strong security strength and good stealth, with only modest cost. The key idea is that instead of inventing brand new obfuscation techniques, we can exploit some existing programming languages for their unique design and implementation features to achieve obfuscation effects. In general, programming language features are rarely proposed or developed for obfuscation purposes; however, some of them indeed make reverse engineering much more challenging at the binary level and thus can be "misused" for software protection. In particular, some programming languages are designed with unique paradigms and have very complicated execution models. To make use of these language features, we can translate a program written in a certain language to another language which is more “confusing”, in the sense that it consists of features leading to obfuscation effects.

In this dissertation, we obfuscate C programs by translating them into Prolog, presenting a feasible example of the translingual obfuscation scheme. C is a traditional imperative programming language while Prolog is a typical logic programming language. The Prolog language has some prominent features that provide strong obfuscation effects. Programs written in Prolog are executed in a *search-and-backtrack* computation model which is dramatically different from the execution model of C and much more complicated. Therefore, translating C code to Prolog leads to obfuscated data layouts and control flows. Especially, *the complexity of Prolog’s execution model manifests mostly in the binary form of the programs,* making Prolog very suitable for software protection.

Translating one language to another is usually very difficult, especially when the target and source languages have different programming paradigms. However, we made an important observation that for obfuscation purposes, language translation could be conducted in a special manner. Instead of developing a “clean” translation from C to Prolog, we propose an “obfuscating” translation scheme which retains part of the C memory model, in some sense making two execution models mixed together. We believe this improves the obfuscating effect in a way that no obfuscation methods have achieved before, to the best of our knowledge. Consequently in translingual obfuscation, the obfuscation does not only come from the obfuscating features of the target language, but also from the translation itself. With this new translation scheme we manage to kill two birds with one stone, i.e., solving the technical problems in implementing translingual obfuscation and strengthening the obfuscation simultaneously.

We have implemented translingual obfuscation in a tool called Babel. Babel can selectively transform a C function into semantically equivalent Prolog code and compile code of both languages together into the executable form. Our experiment results show that translingual obfuscation is obscure and stealthy. The execution overhead of Babel is modest compared to a commercial obfuscator. We also show that translingual obfuscation is resilient to one of the most popular reverse engineering techniques.

1.3 Obfuscation in Mobile Software Development

Apart from developing new obfuscation techniques, the dissertation also aims to advance the application and deployment of software protection particularly on the emerging mobile platform. The prosperity of smartphone markets has raised new concerns about software security on mobile platforms, leading to a growing demand for effective software obfuscation techniques. Although software obfuscation has been intensively studied for the traditional desktop computing environment, the status of mobile application obfuscation is yet to be reviled. As far as we have learned, little emphasis is put on investigating how benign software authors take obfuscation as part of their development process in the real world, which is critical for software obfuscation techniques to be practical. Therefore, both the academia and the industry are interested in a comprehensive study on the latest status of mobile app obfuscation so that the strength of mobile software protection techniques can be understood and improved. This dissertation aims to fulfill this demand.

The mobile ecosystem is currently dominated by two major platforms, i.e., iOS and Android. The two systems are very distinguishable with respect to the development, deployment, and execution of mobile applications. iOS applications are written in C and C-like programming languages and compiled to native machine code before installed and executed on devices. In contrast, Android applications are mostly written in Java and compiled to byte code. Before Android 5.0, Android applications are executed in a managed environment called the Dalvik virtual machine. Starting from 5.0, application bytecode will be further compiled into native code right before being installed onto the devices. Either way, Android developers cannot directly obfuscate their applications on the native code level. In general, obfuscating Android application is more specific to the Java programming language, which is a unique research topic [65, 153].

In this dissertation, we focus on studying the iOS mobile system. By the time of this research being conducted, there are more than one million iOS applications published. By analyzing the latest versions of this large set of applications, we are able to get a comprehensive understanding of how software obfuscate is practiced in the iOS ecosystem, where millions of developers and hundreds of millions of users are involved.

As a step forward to investigating the applications of obfuscation techniques in real-world software development, we collaborate with mobile app developers in the industry and develop obfuscators to protect multiple commercial iOS apps with millions of users, leveraging the knowledge obtained from the empirical study. The dissertation reports our experience of obfuscating multiple commercial iOS apps with millions of active users. To date, there exist various supposedly effective obfuscation techniques that may fulfill the demand of the mobile software industry. However, the techniques themselves do not automatically lead to effective and practical software protection, especially for mobile apps. Oftentimes, the hardware and software environments of mobile platforms impose harsh restrictions on the types and configurations of obfuscations that can are applied to mobile apps. Additionally, obfuscation must not affect the regular development, distribution, and maintenance of mobile apps, which usually requires further customization to be made for the adopted obfuscation techniques.

1.4 Contributions

In summary, we make the following contributions in this dissertation:

- On developing novel software obfuscation techniques,
  - We proposed a new obfuscation method named translingual obfuscation. Translingual obfuscation is novel for utilizing exotic language features instead of ad-hoc program transformations to protect programs against reverse engineering. Our new method has a number of advantages over existing obfuscation techniques, which will be discussed in depth in later chapters.
  - We implemented translingual obfuscation in a tool called Babel which translates C to Prolog at the scale of subroutines, i.e., from C functions to Prolog predicates, to obfuscate the original programs. Language translation is always a challenging problem, especially when the target language has a heterogeneous execution model.
  - We evaluated Babel with respect to all four evaluation criteria proposed by Collberg et al. [52]: potency, resilience, cost, and stealth, on a set of real-world C programs with quite a bit of complexity and diversity. Our experiments demonstrate that Babel provides strong protection against reverse engineering with only modest cost.
- On investigating the status of software obfuscation application in real-world mobile software development,
  - We are the first to conduct a comprehensive empirical study targeting mobile software obfuscation. Our research focuses on iOS, an influential mobile platform that did not receive enough attention from the academia in contrast to Android.

  - We developed a scalable detection algorithm to estimate the likelihood of an iOS app being obfuscated and applied it to a large quantity of apps crawled from App Store. After manually analyzing the 6600 most likely obfuscated instances, we identified 539 truly obfuscated iOS apps with a total of 601 different versions. As far as we know, this is the first scientifically collected sample set of obfuscated iOS mobile apps. We plan to share these samples with the community in the future.
  - To overcome the limitations of existing automated software analysis on obfuscated binaries, we invested over 600 man-hours in manually examining the obfuscated iOS apps, extracting detailed information about how these apps are protected by different obfuscation algorithms. The human effort assured the accuracy of our analysis and therefore the credibility of our findings.
  - We made various observations about the characteristics of obfuscated apps, the obfuscation patterns applied, and their resilience to reverse engineering. Our findings can shed light on future research on mobile software protection.

- For applying obfuscation to iOS apps with large user bases, we help mobile developers form a deeper understanding of software obfuscation and avoid common pitfalls that may appear when obfuscating iOS apps, we discuss our learned lessons on the following topics,
  - Why iOS apps are in urgent need of the protection of software obfuscation, from an industrial point of view,
  - What restrictions are imposed by the iOS platform on obfuscation techniques,
  - How the centralized app distribution process can impact practice of obfuscation, and

  - How to balance obfuscation and app maintenance.

The report will benefit both mobile app developers, distributors, and researchers aiming to develop advanced obfuscation techniques applicable to mobile software.

Chapter 2

Related Work

This chapter reviews historical work related to the topics discussed in this dissertation. We introduce the related work in four aspects, i.e., software obfuscation techniques, the corresponding deobfuscation techniques, programming language translation methods, and studies on mobile software development.

2.1 Software Obfuscation

2.1.1 Cryptography Obfuscation

There are different notions of secure obfuscation in the theoretical sense. The most comprehensive definition known to date is called virtual black-box (VBB) obfuscation, proposed and studied by Hada [78] and Barak et al. [35]. Semi-formally, an *effective* VBB obfuscator is a program transformation algorithm $\mathcal{O}$, where given any program P, $\mathcal{O}(P)$ computes the same function $f \in \mathcal{F}$ as P does; meanwhile, for any non-trivial function property $\phi : \mathcal{F} \rightarrow \mathcal{S}$ and any program analyzer $\mathcal{A}_\phi$ that tries to efficiently compute ϕ, if $\phi(f)$ is intractably hard given only black box access to P as an oracle, the result of $\mathcal{A}_\phi(\mathcal{O}(P))$ is no better than randomly guessing $\phi(f)$.

Unfortunately, it has been proven by Barak et al. [35] that VBB obfuscation is theoretically impossible for general programs. In response to this finding, Barak et al. also defined a weaker notion of obfuscation called indistinguishability obfuscation. Garg et al. [70] developed an algorithm based on multi-linear maps to realize indistinguishability obfuscation for all circuits, but evaluation results [30] show that the technique is yet unpractical for actual deployment.

There exists a line of work that specially considers obfuscating memory access patterns for programs that are vulnerable to memory-based side channel attacks. The so-called Oblivious RAM (ORAM) is an obfuscation technique that hides the external memory access patterns of programs, proposed by Goldrech and Ostrovksy [76]. A large volume of follow-up work has tried to make ORAM implementations practical, including Tiny ORAM [64] and GhostRider [95].

2.1.2 Heuristic Obfuscation

Most obfuscation techniques that are practically used in the real world are based heuristics. Although they lack theoretical guarantees about their resilience to all possible program analysis methods, it has been shown empirically that they can raise the bar of reverse engineering.

Heuristic-based obfuscation can be on either source level or binary level. Early work on obfuscation is more source-code oriented. For source code obfuscation, Wang et al. [133] proposed to convert functions into single switch statements to hide logical links among basic blocks. The technique is called control flow flattening and frequently seen in practical software obfuscation tools. A conceptually similar technique called class hierarchy flattening writes the class inheritance relations of programs written in object-oriented programming languages to hinder both human and automated analysis [66]. Sharif et al. [125] encrypted equality conditions that depend on input data with some one-way hash functions. The evaluation shows that it is virtually impossible to reason about the inputs that satisfy the equality condition with symbolic execution. Moser et al. [106] demonstrate that opaque predicates can effectively hide control transfer destination and data locations from advanced malware detection techniques.

Obfuscation-oriented program transformations can also be performed at the binary level. The most basic form of binary obfuscation is to eliminate certain common code patterns, typically introduced by compilers, such that reverse engineers cannot easily reconstruct source-level structures by experience. For example, there has been an x86 binary obfuscation technique that replaces `call` instructions with a combination of `push` and `ret` instructions to hide calling contexts from analyzers [87].

More complicated binary obfuscations usually takes the whole binary into consideration rather than considering local code snippets. Popov et al. [116] obfuscate programs by replacing control transfers with exceptions, implementing real control transfers in exception handling code, and inserting redundant junk transfers after the exceptions. Mimimorphism [145] transforms a malicious binary into a mimicry benign program, with statistical and semantic characteristics highly similar to the mimicry target. As a result, obfuscated malware can successfully evade statistical anomaly detection. Chen et al. [45] propose a control-flow obfuscation method making use of Itanium processors' architectural support for information flow tracking. In detail, they utilize the deferred exception tokens in Itanium processor registers to implement opaque predicates. Domas [59] developed a compiler which generates a binary employing only the `mov` family instructions, based on the fact that x86 `mov` is Turing complete. There are other binary obfuscation methods which heavily relies on compression, encryption, and virtualization [77, 101, 120]. Among these obfuscation techniques, binary packers using compression and encryption can be vulnerable to dynamic analysis because the original code has to be restored at some point of program execution. As for virtualization-based obfuscation, most current approaches are implemented in the decode-dispatch scheme [128]. Recent effort [119, 126] has identified the characteristics of the decode-dispatch pattern in the virtualization-obfuscated binaries so that they can be effectively reverse engineered.

2.2 Software Deobfuscation

As for countering heuristic-based obfuscation, most recent work focuses on attacking virtualization-based obfuscation. Sharif et al. [126] has developed an outside-in approach which first reverse engineers the virtual machine and then decodes the bytecode to recover the protected program. This approach heavily relies on some assumptions about the structure and working process of he virtual machine. If the virtualizer does meet these assumptions, the deobfuscator is likely to fail. Another deobfuscation method presented by Coogan et al. [54] chooses the inside-out method which utilizes equational reasoning to simplify the execution traces of protected programs. In this way, the deobfuscator extracts instructions which are truly relevant to program logic. A very recent method proposed by Yadegari et al. [152] improved the inside-out approach with more generic control flow simplification algorithms that can deobfuscate programs protected by nested virtualization. Without access to these tools, we cannot directly test BABEL's resilience to them. However, since BABEL completely reforms C programs' data layout and reconstructs the control flows with a much different programming paradigm, we are very confident with BABEL's security strength against these approaches.

Binary diffing is another widely used reverse engineering technique that takes program obfuscation into account. Binary differs identify the syntactical or semantic similarity between two different binaries, and can be used to detect programming plagiarism and launch similarity-based attacks [40]. BinDiff [4] and CoP [97], the two differs we use for evaluating BABEL's resilience, are currently the state of the art. Other examples of binary differs include DarunGrim2 [10], Bdiff [3], BinHunt [69], and iBinHunt [104]. Although these tools can defeat certain types of program obfuscation, none of them are designed to handle the complexity of translingual obfuscation.

Recent research efforts have also shown interest in countering obfuscation in terms of revealing software author identity instead of understanding the logic and properties of the code. Caliskan-Islam et al. [43] used machine learning techniques to establish code stylometry for anonymized source code, via which the identities of code authors can be reliably guessed out. Latest progress [42] showed that code de-anonymization is possible even for obfuscated binary code.

2.3 Programming Language Translation

Translingual obfuscation is a technique that relies on the feasibility of translating one programming language to another, so it is essential to discuss historical work on language translation. Programming language translations have been proven useful for for software portability, software re-engineering, and security hardening purposes. The source-to-source translation from C/C++ to Java is one of the most extensively explored topics in this field, leading to tools such as C2J [86], C++2Java [6], and Cappuccino [41], etc. Trudel et al. [132] developed a converter that translates C to Eiffel, another object-oriented programming language. A tool called Emscripten can translate LLVM intermediate representation to JavaScript [154]. Since C/C++/Objective-C source code can be compiled into LLVM intermediate representation, Emscripten can also be used as a source-to-source translator without much additional effort. The C-to-Prolog translation introduced in this dissertation is partial since we need to keep the original C execution environment; however, our translation is for software obfuscation and being partial is not a limitation. Instead, we later show that for our purpose, many technical issues commonly seen in programming language translation can be either addressed or circumvented.

2.4 Empirical Studies on Mobile Apps and Software Obfuscation

To the best of our knowledge, most historical work on mobile app analysis targets the Android platform. The Android Malware Genome project is among the earliest research efforts that perform large-scale analysis on mobile app repositories [156]. By working on over 1200 samples, the authors managed to develop a systematic characterization for existing Android malware. According to this research, mobile malware authors by then had already started to apply obfuscation to bypass anti-virus analysis. Besides malware that harms users, mobile app repackaging that harms the interest of developers has also drawn attention. Various tools and systems have been developed to detect and analyze cloned mobile applications with both accuracy and scalability [73, 46, 134, 155]. Researchers have also worked on examining third-party libraries used by mobile developers. Tools like LibRadar [99] and LibD [91] were developed to detect third-party libraries in Android apps and classify them. Research by Chen at al. [47] detects libraries potentially harmful to user security and privacy for both Android and iOS.

Despite the progress in mobile app analysis, most studies of this kind either ignored or spent very limited effort in handling the presence and influence of software obfuscation. One of the few studies that systematically investigated the impact of obfuscation on mobile development is from Linares-Vásquez et al., who researched how obfuscation can affect the detection of Android code cloning [93]. Similar to our work, Linares-Vásquez et al. spent extensive manual work in identifying obfuscated code, but their analysis only covered 120 apps and did not consider obfuscation methods other than identifier scrambling. CodeMatch is a similar project that focuses on obfuscation-resilient Android library detection [75]. Xue et al. [149] proposed adaptive unpacking of Android apps to recover dex code, which can potentially enable obfuscation-resilient clone or library detection.

There is also research effort empirically analyzing code obufscation on platforms other than mobile. Xu et al. conducted a study [148] on obfuscated JavaScript code considered malicious by the online malware detection service VirusTotal [27]. Same as our study, their research started with manual sample collection due to the lack of a previously established data set. The sampling mostly relied on manual analysis since there was no publicly known automatic tools that detects obfuscation in JavaScript code with reliable accuracy. The study investigated the use of four pre-selected categories of obfuscation algorithms and measured their resilience to different malware detectors. Still, the focus of this study is exclusively on code that adopts obfuscation for malicious purposes. Our research, on the other hand, considers obfuscation to be part of common software engineering practices.

Chapter 3

Advanced Obfuscation for Desktop Software

This chapter introduces *Translingual Obfuscation*, an advanced software obfuscation technique designed for protecting desktop and server software.

3.1 Overview

The basic idea of translingual obfuscation is that some programming languages are more difficult to reverse engineer than others. Intuitively, C is relatively easy to reverse engineer because binary code compiled from C programs shares the same imperative execution model with the source code. For some programming languages like Prolog, however, there is a much deeper gap between the source code and the resulting binaries, since these languages have fundamentally different abstractions from the imperative execution model of the underlying hardware. Starting from this insight, we analyze and evaluate the features of a foreign programming language from the perspective of software protection. We also develop the translation technique that transforms the original language to the obfuscat-

The work of this chapter is published in *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, 2016 [136].

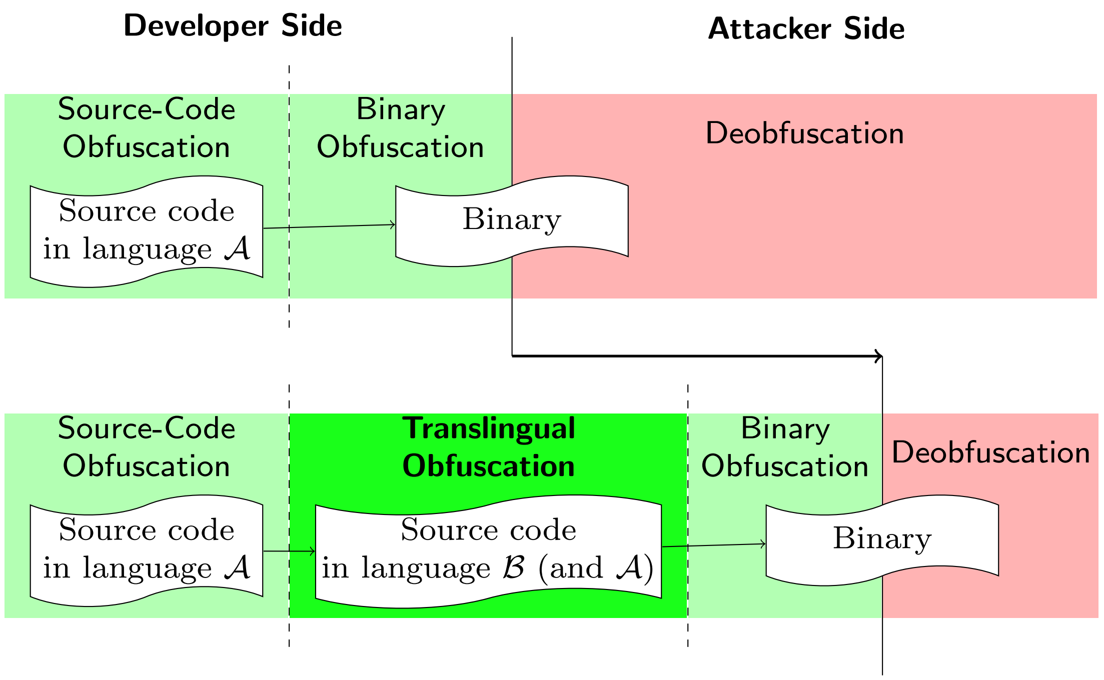


Figure 3.1: Translingual obfuscation is a new protection layer complementary to existing obfuscation methods, pushing the frontier forward in the battle with reverse engineering.

ing language. Only with these efforts devoted, translingual obfuscation can be a practical software protection scheme.

We view translingual obfuscation as a new layer of software protection in the obfuscation-deobfuscation arms race, as shown in Figure 3.1. Different from previous obfuscation methods which either work at the binary level or perform same-language source-to-source transformations, translingual obfuscation translates one language to another. Therefore, translingual obfuscation can be applied after source-code obfuscation and before binary obfuscation without affecting the applicability of existing obfuscation methods.

The virtualization-based obfuscation is currently the state of the art in binary obfuscation. Some features of translingual obfuscation resemble the idea of virtualization-based obfuscation, but we want to emphasize a significant difference here. Currently, most implementations of virtualization-based obfuscation

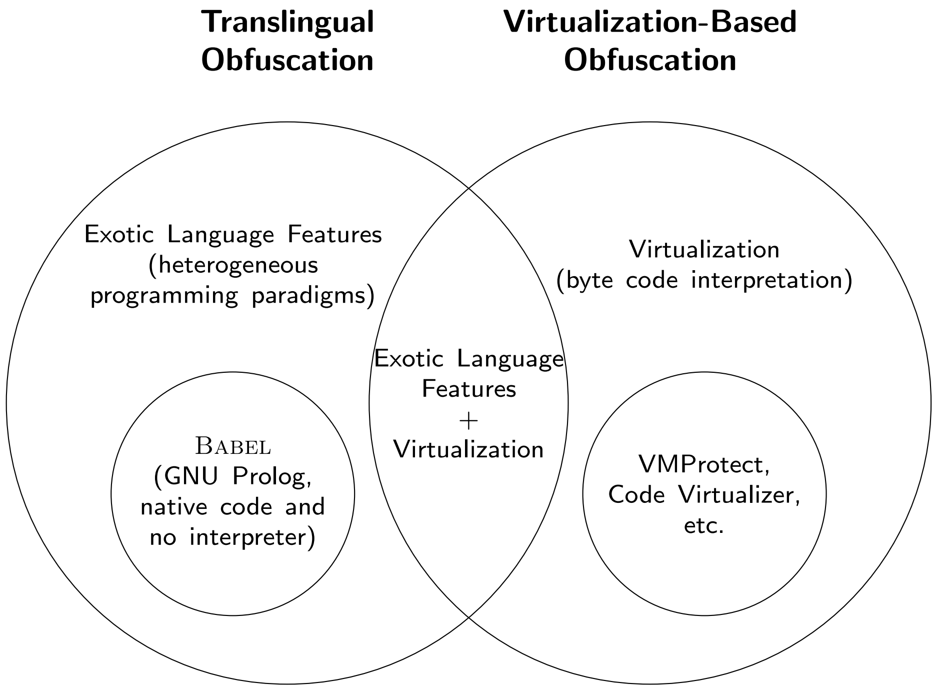

Translingual Obfuscation
Virtualization-Based Obfuscation
Exotic Language Features (heterogeneous programming paradigms)
Virtualization (byte code interpretation)
Exotic Language Features + Virtualization
Babel (GNU Prolog, native code and no interpreter)
VMProtect, Code Virtualizer, etc.

Figure 3.2: Comparing translingual obfuscation and virtualization-based obfuscation.

tend to encode original native machine code with a RISC-like virtual instruction set and interpret the encoded binary in a decode-dispatch pattern [128, 119, 72], which has been identified as a notable weakness of security and can be exploited by various attacks [126, 54, 152]. Translingual obfuscation, however, gets most of its security strength by *intentionally* relying on obfuscation-contributing language features that comes from a heterogeneous programming model. Essentially, translingual obfuscation does not have to re-encode the original binary as long as the foreign language employed supports compilation into native code. Figure 3.2 shows the relationship and key differences between the two methods. Our translingual obfuscation implementation Babel and virtualization-based obfuscation do not overlap.

Translingual obfuscation can provide benefits that cannot be delivered by any single obfuscation method developed before, to the best of our knowledge:

- Translingual obfuscation provides strong obfuscation strength and more obfuscation variety by introducing a different programming paradigm. If there exists *a universally effective and automated method*[1] to nullify the obfuscation effects, namely the additional program complexity, introduced by a programming language's execution model, that would mean it is possible to significantly simplify the design and implementation of that language, which is very unlikely for mature languages.

- Translingual obfuscation can be very stealthy, because programming with multiple languages is a completely legit practice. Compared with virtualization-based obfuscation which encodes native code into bytecode that has an exotic encoding format, translingual obfuscation introduces neither abnormal byte entropy nor deviant instruction distributions.

- Translingual obfuscation is not just a single obfuscation algorithm but a general framework. Although we particularly utilizes Prolog in this dissertation, there are other languages that can be misused for translingual obfuscation. For example, the New Jersey implementation of ML (SML/NJ) [31] does not even include a run-time stack. Instead, it allocates all frames and closures on a garbage-collected heap, potentially making program analysis much more difficult. Another example is Haskell, a pure functional language featuring lazy evaluation [90] which can be implemented with a unique execution model that greatly differs from the traditional imperative computation [100].

All these benefits make us believe that translingual obfuscation could be a new direction in software protection.

[1]As mentioned earlier, for every obfuscation method, there exists a program such that a reverse engineering method, maybe with manual effort, can effectively deobfuscate that particular obfuscated program.

3.2 Threat Model

For attackers who try to reverse engineer a program protected by obfuscation, we assume that they have full access to the binary form of the program. They can examine the static form of the binaries with whatever method available to them. They can also execute the victim binaries in a monitored environment with arbitrary input, thus can read any data that has lived in the memory.

Do note that although we assume attackers have unlimited access to program binaries, they should not posses any knowledge about the source code in our threat model. Assuming attackers can only examine the obfuscated program at the binary level is important, because that would mean any implementation detail of the language used in translingual obfuscation contributes to the effectiveness of obfuscation. As for the particular case of employing Prolog in translingual obfuscation, since Prolog is a declarative programming language, there is a much deeper semantic gap between its source code and binaries, which is highlighted as one of the major sources of translingual obfuscation's protection effects.

Finally, we explicitly clarify that in this work, attackers are assumed to try to reveal the logical structure of the binaries so that they can reproduce the algorithms by themselves. In practice there are different levels of reverse engineering objectives. Sometimes understanding what a program achieves is sufficient for attackers to fulfill their goals, but in our case attackers need a more thorough understanding on the semantics of the victim binaries.

Our threat model may seem too coarsely defined. However, we believe it is quite realistic, since reverse engineering could be a very ad-hoc process in practice. Actually, lack of specifications makes it difficult for us to design and evaluate a new obfuscation technique in a fully comprehensive manner, because we cannot make further assumptions on the methods or tools that attackers may make use of.

3.3 Misusing Prolog for Obfuscation

In this section we briefly introduce the Prolog programming language and explain why we can misuse its language features for obfuscation.

3.3.1 Prolog Basics

The basic building blocks of Prolog are *terms*. Both a Prolog program itself and the data it manipulates are built from terms. There are three kinds of terms: constants, variables, and structures. A constant is either a number (integer or real) or an atom. An atom is a general-purpose name, which is similar to a constant string entity in other languages. A structure term is of the form $f(t_1, \cdots, t_n)$, where f is a symbol called a *functor* and $t_1, \cdots, t_n$ are subterms. The number of subterms a functor takes is called its *arity*. It is allowed to use a symbol with different arities, so the notation '`f/n`' is used when referring to a structure term f with n subterms.

Structure terms become *clauses* when assigned semantics. A clause can be a fact, a rule, or a query. A predefined clause is a fact if it has an empty body, otherwise it is a rule. For example, "`parent(jack,bill).`" is a fact, which could mean that "jack is a parent of bill." One the other hand, a rule can be like the following line of code:

```
grandparent(G,C):-parent(G,P),parent(P,C).
```

This rule can be written as the following formula in the first-order logic:

$$\forall G, C, P.\text{grandparent}(G, C) \leftarrow \text{parent}(G, P) \wedge \text{parent}(P, C)$$

Clauses with the same name and the same number of arguments define a relation, namely a *predicate*. With facts and rules defined, programmers can issue *queries*, which are formulas for the Prolog resolution system to solve. In accordance with our previous examples, a query could be `grandparent(G,bill)` which is basically asking "who are bill's grandparents?"

A Prolog program is a set of terms. The Prolog resolution engine maintains an internal database of terms throughout program execution, trying to resolve queries with facts and rules by logical inference. Essentially, computation in Prolog is reduced to a searching problem. This is different from the commonly seen Turing machine computation model but the theoretical foundation of logic programming guarantees that Prolog is Turing complete [130].

3.3.2 Obfuscation-Contributing Features

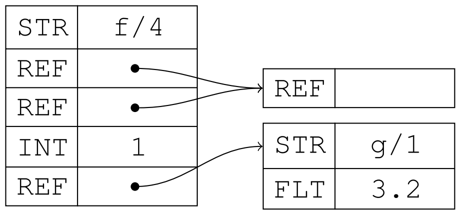
3.3.2.1 Unification

One of the core concepts in automated logic resolution, hence in logic programming, is unification. Essentially it is a pattern-matching technique. Two first-order terms t_1 and t_2 can be unified if there exists a substitution σ making them identical, i.e., $t_1^\sigma = t_2^\sigma$. For example, two terms $k(s(g), Y)$ and $k(X, t(k))$ are unified when X is substituted by $s(g)$ and Y is substituted by $t(k)$.

Unification is the basics of computation in Prolog and we explain this with a straight example. The following clause defines a simple "increment-by-one" procedure:

```
inc(Input,Output):-Output is Input+1.
```

Now for a query `inc(1,R)`, the Prolog resolution engine will first try to unify `inc(1,R)` with `inc(Input,Output)`, which means `Input` should be unified with `1` and `Output` should be unified with `R`. Once this unification succeeds, the original query is reduced to a subgoal `Output is Input+1`. Since `Input` is now unified with `1`, `Input+1` is evaluated as `2`. Finally `Output` gets unified with `2` (`is/2` is the evaluate-and-unify operator predicate), making `R` unified with `2` as well.



Meaning of the tags

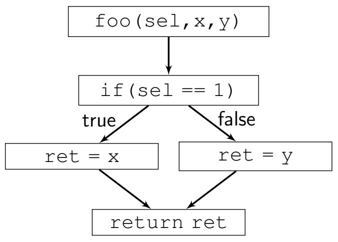
REF:	reference (variable)
STR:	structure
INT:	integer
FLT:	floating point

Figure 3.3: An example memory representation of term `f(X,Y,1,g(3.2))` in Prolog, where both `X` and `Y` are unified with another variable which itself is ununified.

To support unification, Prolog implement terms as vertices in directed acyclic graphs. Each term is represented by a `<tag,content>` tuple, where `tag` indicates whether the type of the term and `content` is either the value of a constant or the address of the term the variable is unified with. Figure 3.3 is an example showing how Prolog may represent a term in memory [33].

Unification makes data shapes in Prolog program memory dramatically different from C and much more obscure. The graph-like implementation of unification poses great challenges to binary data shape analyses which aim to recover high-level data structures from binary program images [81, 71, 121, 55]. Even if some of the graph structures can be identified, there is still a gap between this low-level representation and the logical organization of original data, which harshly tests attackers' reverse engineering abilities. Unification also complicates data access. To retrieve the true value of a variable, the Prolog engine has to iterate the entire unification list. It is well known that static analysis is weak against loops and indirect memory access. Also, the tags in the term tuples are encoded as bit fields, meaning that bit-level analysis algorithms are required to reveal the semantics of a binary compiled from Prolog code. However, achieving bit-level precision is another great technical challenge for both static and dynamic program analyses, mainly because of scalability issues [124, 84, 60, 151].

```
int foo(int sel,
        int x, int y)
{
  int ret;
  if(sel==1)
    ret=x;
  else
    ret=y;
  return ret;
}
```



```
pfoo(Sel,X,Y,R) :-
  (Sel =:= 1 ->
     R is X);
  (R is Y).
```

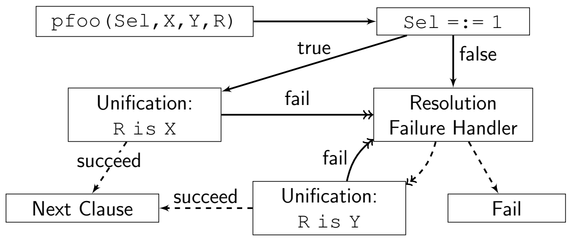


Figure 3.4: Different control flows of C and Prolog binaries implementing the same algorithm, due to different execution models. In the Prolog graph, dashed lines indicate indirect jumps and arrows with the same pattern indicate feasible paths through the resolution failure handler. Both control flow graphs are summarized from post-compilation binaries.

3.3.2.2 Backtracking

Different from Prolog unification which mainly obfuscates program data, the backtracking feature obfuscates the control flow. Backtracking is part of the resolution mechanism in Prolog. As explained earlier, finding a solution for a resolvable formula is essentially searching for a proper unifier, namely a substitution, so that the substituted formula can be expanded to consist of only facts and other formulas known to be true. Since there may be more than one solution for a unification problem instance, it is possible that the resolution process will unify two terms in the way that it makes resolving the formula later unfeasible. As a consequence, Prolog needs a mechanism to roll back from an incorrect proof path, which is called backtracking.

To make backtracking possible, Prolog saves the program state before taking one of the search branches. This saved state is called a “choice-point” by Prolog and is similar to the concept of “continuation” in functional programming. When searching along one path fails, the resolution engine will restore the latest choice-point and continue to search through one of the untried branches.

This *search-and-backtrack* execution model leads to a totally different control flow scheme in Prolog programs at the low level, compared to programs in the same logic written by C. Figure 3.4 is an example where a C function is transformed into a Prolog clause by our tool Babel (with manual edits to make the code more readable), along with the program execution flows before and after Babel transformation. The real control flow of the Prolog version of the function is much more complicated than presented, and we have greatly simplified the flow chart for readability. In the Prolog part of Figure 3.4, a choice point is created right after the execution flow enters the predicate `pfoo` which is a disjunction of two subclauses. The Prolog resolution routine will first try to satisfy the first subclause. If it fails, the engine will backtrack to the last choice-point and try the second subclause.

Due to the complicated backtracking model, a large portion of control flow transfers in Prolog are indirect. The implementation of backtracking also involves techniques such as long jump and stack unwinding. Clearly, Prolog has a much more obscure low-level execution model compared to C, and imperative programming in general, from the perspective of static analysis. Different from some other control flow obfuscation techniques that inject fake control flows which are never feasible at run time, Prolog's backtracking actually happens during program execution, making translingual obfuscation also resilient to dynamic analysis. Most importantly, after the C-to-Prolog translation the original C control flows are re-formed with a completely different programming paradigm, which is fundamentally different from existing control-flow based obfuscation techniques.

3.4 Technical Challenges

To make use of Prolog's execution model for obfuscating C programs, we need a translation technique to forge the Prolog counterpart of a C function. At this point, there are various challenges to resolve.

3.4.1 Control Flow

As an imperative programming language, C provides much flexibility of crafting program control flows almost with language key words such as continue, break, and return. Prolog programs, however, have to follow the general evaluation procedure of logical formulas, which inherently forbids some "fancy" control flows allowed by C.

3.4.2 Memory Model

In C programming, many low-level details are not opaque to programmers. As for memory manipulation, C programmers can access almost arbitrary memory locations via pointers. Prolog lacks the semantics to express direct memory access. Moreover, the C memory model is closely coupled with other sub-structures of the language, e.g., the type system. C types are not only logical abstractions but are also implications on low-level memory layouts of the data. For instance, logically adjacent elements in a C array and fields in a C struct are also physically adjacent in memory. Therefore, some logical operations on C data structures can be implemented as direct memory accesses which are semantically equivalent only with the C memory modeling. Below is an example.

```
struct { int a, b; } s[2];

/* Equivalent to s[0].a=s[0].b=s[1].a=0;
 * with many compilers and architectures */
```

```
memset((void*)s, 0, 3*sizeof(int));
```

Translating the code snippet above into pure Prolog could be difficult because the translator will have to infer the logic effects of the `memset` statement.

3.4.3 Type Casting

C type casting is of full flexibility in the sense that a C programmer can cast any type to any other type, no matter the conversion makes sense or not. This can be realized by violating the *load-store consistency*, namely storing a variable of some type into a memory location and later loading the content of the same chunk of memory into a variable of another type. The C union type is a high-level support for type castings that breaks the load-store consistency, but C programmers can choose to use pointers to directly achieve the same effect. Imitating this type casting system could be a notable challenge for other languages.

3.5 C-to-Prolog Translation

This section explains how we address the challenges mentioned in the previous section. Considering the many obstacles for developing a complete translation from C to Prolog, we do not seek to obtain a pure Prolog version of the original C program. The section explains how we address the challenges mentioned in the previous section and how we develop a partial C-to-Prolog translation method, which is suitable for translingual obfuscation.

3.5.1 Control Flow Regularization

There has been a large amount of research on refining C program control flows, especially on eliminating goto statements [85, 117, 142]. For now, we consider that goto elimination is a solved problem and assume the C programs to be protected do not contain goto statements. Given a C function without goto statements, there are two control flow patterns that cannot be directly adopted by Prolog programming, i.e., control flow cuts and loops. We call these patterns *irregular* control flows.

3.5.1.1 Control Flow Cuts

Control flow cuts refer to the termination of control flows in the middle of a C function, for example:

```
int foo (int m, int n) {
 if(m)
  return n; // Flow of if branch ends
 else
  n=n+1;
 n=n+2;
 return n;  // Flow of else branch ends
}
```

The C language grants programmers much freedom in building control flows, even without using goto statements. In Prolog, however, control flows have to be routed based on the short-circuit rules in evaluating logical expressions. With short-circuit effects, parallel statements can be connected by disjunction and sequential statements can be connected by conjunction. To show why the control flow pattern in the C code above cannot be implemented by only adopting short-circuit rules, consider a C function with body {`if(e) {a;} else {b;} c;`}. Naturally, it should be translated into a Prolog sentence `(((e->a);b),c)`, where `->` denotes implication, `;` denotes disjunction, and `,` denotes conjunction. However, this translation is not semantics-preserving when `a` is a return statement, because if the clause `a` is evaluated, at least one of `b` and `c` has to be evaluated to decide the truth of the whole logic formula.

We fix control flow cuts by replicating and/or reordering basic blocks syntactically subsequent to the cuts. For example, we rewrite the previously shown C function into the following structure:

```
int foo (int m, int n) {
 if(m)
  return n;
 else {
  n=n+1;
  { n=n+2; return n; }
 }
}
```

After the revision, the C code is naturally translated into a new Prolog clause `(e->a);(b,c)`, which is consistent with the original C semantics.

3.5.1.2 Loops

Most loops cannot be directly implemented in Prolog. The fundamental reason is that Prolog does not allow unifying a variable more than once. We can address this problem by transforming loops into recursive functions, but in-loop irregular control flows complicates the situation. The irregularity comes from the use of keywords “continue” and “return.” A continue statement cuts the control flow in the middle of a loop, bringing up a problem similar to the aforementioned asymmetric returns in functions. As such, irregular control flows resulting from continue statements can be regularized in the same way, i.e., replicating and/or reordering basic blocks syntactically subsequent to continue statements.

Like a continue statement, a return statement also cuts the control flow in a loop, but its impact reaches outside because it cuts the control flow of the function enclosing the loop. Hence, a recursive Prolog predicate transformed from a loop needs an extra argument to carry a flag indicating whether an in-loop return has occurred.

3.5.2 C Memory Model Simulation

As stated in Chapter 3.4, the C memory model is closely coupled with other parts of the language and it is hard to separate them. However, translingual obfuscation keeps the original C memory model, making preserving semantic equivalence much easier. In our design, the Prolog runtime is embedded in the C execution environment, so it is possible for Prolog code to directly operate memories within a program's address space.

The way we handle C memory simulation illustrates the advantage of developing language translations for obfuscation purposes. Unlike tools seeking complete translation from C to other languages, translingual obfuscation does not have to mimic C memory completely with target language features (e.g., converting C pointers to Java references [57]), meaning we can reduce translation complexity and circumvent various limitations. That being said, partially imitating the C memory model in Prolog is still a non-trivial task.

3.5.2.1 Supporting C Memory-Access Operators

The fist step to simulating the C memory model is to support pointer operations. We introduce the following new clauses into our target Prolog language:

```
rdPtrInt(+Ptr, +Size, -Content)
wrPtrInt(+Ptr, +Size, +Content)
rdPtrFloat(+Ptr, +Size, -Content)
wrPtrFloat(+Ptr, +Size, +Content)
```

These clauses are implemented in C. `rdPtrInt/3` and `rdPtrFloat/3` allow us to load the content of a memory cell (address and size indicated by `Ptr` and

`Size`, respectively) into a Prolog variable `Content`. Similarly, `wrPtrInt/3` and `wrPtrFloat/3` can write the content of a Prolog variable into a memory cell. These four clauses simulate the behaviors of the "pointer dereference" operator ($\star$) in C.

In addition to read-from-pointer and write-to-pointer operations, C also has the "address-of" operator which takes an lvalue, i.e., an expression that is allocated a storage location, as the operand and returns its associated storage location, namely address. There is no need to explicitly support this operator in Prolog because the address of any lvalue in C has a static representation which is known by the compiler.[2] We can obtain the results of "address-of" operations in the C environment and pass those values into the Prolog environment as arguments.

We also handle several C syntax sugers related to memory access: "subscript" (`[]`) and "field-of" (`.` and `->`). We convert these operators into equivalent combinations of pointer arithmetic and dereference so that we do not need to coin their counterparts in Prolog. This conversion requires assumptions on compiler implementation and target architecture to calculate type sizes and field displacements.

3.5.2.2 Maintaining Consistency

It is a natural scheme that a C-to-Prolog translation maps every C variable to a corresponding Prolog variable. Prolog does not allow variable update, but we can overcome this restriction by transforming C code into a form close to static single assignment (SSA), in which variables are only initialized at one program location and never updated. In the strict SSA form, variables can only be statically initialized once even if the scopes are disjoint. Prolog does not require this because the language checks re-unification at run time, meaning variables can be updated

[2]For example, a local variable is usually allocated on the stack and the compiler will have a static expression of its address. On x86, the expression is likely to be `$offset(%ebp)` or `$offset(%esp)`, where `$offset` is a constant. Compilers can also decide how to statically represent the addresses of global variables.

```
a=0;
p=&a;// p points to a
a=1; // a gets 1
b=*p;// b gets a(1)

c=0;
p=&c;
c=1; // c gets 1
*p=3;// c gets 3
d=c; // d gets c(3)
```

(a) Original

```
a1=0;
p1=&a1;// p1 points to a1
a2=1;  // a2 gets 1
b1=*p1;// b1 gets a1(0)

c1=0;
p2=&c1;// p2 points to c1
c2=1;  // c2 gets 1
*p2=3; // c1 gets 3
d1=c2; // d1 gets c2(1)
```

(b) Renamed

Figure 3.5: Memory operations affecting the correctness of C source code SSA renaming.

in exclusively executed parts of the program, e.g., the "then" and "else" branches of the same if statement. Therefore, we do not need to implement the ϕ function in our SSA transformation.

The SSA transformation can be implemented by renaming variables. The challenging part is that simply renaming variables in the original C code could break program semantics because of side effects caused by memory operations, i.e., variable contents can be accessed without referring to variable names. This is the consistency problem we have discussed earlier. Figure 3.5 shows an instance of the problem.

To address this issue, we keep the addresses of local variables and parameters if they are possibly accessed via pointers. Then we flush variable contents back to the memory before a read-from-pointer operation and reload variable contents from the memory after a write-to-pointer operation. Inter-procedural pointer dereferences are also taken into account. When callee functions accept pointers as arguments, we do variable flush before function calls and do variable reload after. The flush makes sure that changes made by Prolog code are committed to the underlying C memory before they are read again. Similarly, the reload assures that values

```
a=0;
p=&a;
a=1;

b=*p;

c=0;
p=&c;
c=1;
*p=3;

d=c;
```

(a) Original

```
pa=&a;
pc=&c;

a=0;
p=&a;
a=1;
*pa=a;// Flush
b=*p;

c=0;
p=&c;
c=1;
*p=3;
c=*pc;// Reload
d=c;
```

(b) With flush and reload

```
pa=&a1;// const pointer
pc=&c1;// const pointer

a1=0;
p1=&a1;
a2=1;  // a2 gets 1
*pa=a2;// a1 gets a2(1)
b1=*p1;// b1 gets a1(1)

c1=0;
p2=&c1;
c2=1;
*p2=3; // c1 gets 3
c3=*pc;// c3 gets c1(3)
d1=c3; // d1 gets c3(3)
```

(c) Renamed

Figure 3.6: Semantic-preserving SSA renaming on C source code with the presence of pointer operations.

unified with Prolog logical variables are always consistent with the content in C memory. We perform a sound points-to analysis to compute the set of variables that need to be reloaded or flushed at each program point. After inserting the flush and reload operations, the SSA variable renaming no longer breaks the original program semantics. Figure 3.6 illustrates our solution based on the example in Figure 3.5.

3.5.3 Supporting Other C Features

3.5.3.1 Struct, Union, and Array

In Chapter 3.4, we showed that C data types like struct and array can be manipulated via memory access. Since we have already built support for the C memory model in Prolog, the original challenge now becomes a shortcut to supporting C struct, union, and array. We simply transform the original C code and implement all operations on structs, unions, and arrays through pointers. After this transformation the primitive data types provided by Prolog are enough to represent any C data structure.

3.5.3.2 Type Casting

With our C memory simulation method, supporting type castings performed via pointers does not require additional effort, even if they may violate the load-store consistency. As for explicit castings, e.g., from integers to floating points, we utilize the built-in Prolog type casting clauses like `float/1`.

3.5.3.3 External and Indirect Function Call

Since the source code of library functions is usually unavailable, translating them into Prolog is not an option. In general, translations of translingual obfuscation can support external subroutine invocation with the help of foreign language interfaces. As for C+Prolog obfuscation, most Prolog implementations provide the interface for calling C functions from a Prolog context. The same interface can also be used to invoke functions via pointers.

3.5.4 Obfuscating Translation

Our translation scheme fully exploits the obfuscation-contributing features introduced in Chapter 3.3.2, generally because:

- The conversion from C data structures to Prolog data structures happens by default, and every C assignment is translated to Prolog unification.
- Intra-procedural control-flow transfers originally coded in C are now implemented by Prolog's backtracking mechanism. This significantly complicates the low-level logic of the resulting binaries.

Especially, we would like to highlight the method we use to support the C memory model in Prolog. At the high level, the original C memory layout is kept after the translation. However, the behavior of the C-part memory becomes much different from the original program. To maintain the consistency between the C-side memory and Prolog-side memory, we introduce the flush-reload method which disturbs the sequence of memory access. In this way, the memory footprint of the obfuscated program is no longer what it was during program execution.

We believe our translation method is one of the factors that make translingual obfuscation resilient to both semantics-based and syntax-based binary diffing, as will be shown in Chapter 3.7.2.

3.6 Implementation

Babel is our translingual obfuscation prototype. The workflow of Babel has three steps: C code preprocessing, C-to-Prolog translation, and C+Prolog compilation. The preprocessing step reforms the original C source code so that the processed program becomes suitable for line-by-line translation to Prolog. The second step translates C functions to Prolog predicates. In the last step, Babel combines C and Prolog code together with a carefully designed interface.

We choose GNU Prolog [58] as the Prolog implementation to employ in Babel. Like many other Prolog systems, GNU Prolog compiles Prolog source into the "standard" Warren Abstract Machine (WAM) [141] instructions. What is desirable to us is that GNU Prolog can further compile WAM code into native code. This feature makes Babel more distinguishable from virtualization-based obfuscation tools which compile the original program to bytecode and execute it with a custom virtual machine.

3.6.1 Preprocessing and Translating C to Prolog

Before actually translating C to Prolog, we need to preprocess the C code first. The preprocessing includes the following steps, which is done with the help of the CIL library [110].

1. Simplify C code into the three-address form without switch statements and ternary conditional expressions.
2. Convert loops to tail-recursive functions.
3. Eliminate control flow cuts.
4. Transform operations on global, struct, union, and array variables into pointer operations.
5. Perform variable flush and reload whenever necessary.
6. Eliminate all memory operators except pointer dereferences.
7. Rename variables so that the C code is in a form close to SSA.

After preprocessing, we can translate C to Prolog line by line. The translation rules are listed in Figure 3.7. Note that by the time we start translating C to Prolog, the preprocessed C code does not contain any switch and loop statements, because they are transformed into either nested if statements or recursive functions. As discussed in Chapter 3.5.1, we do not consider goto statements.

We take translating arithmetic and logical expressions as a trivial task, but that leads to a limitation in our translation. Due to the fact that Prolog does not subdivide integer types, integer arithmetics in Prolog are not equivalent to their C counterparts. For example, given two C variables `x` and `y` of type `int` (4 bytes long) and their addition `x+y`, the equivalent expression in Prolog should be `(X+Y)/\0xffffffff`, assuming that `X` and `Y` are the corresponding logical variables of `x` and `y`. Therefore, if a C program intentionally relies on integer

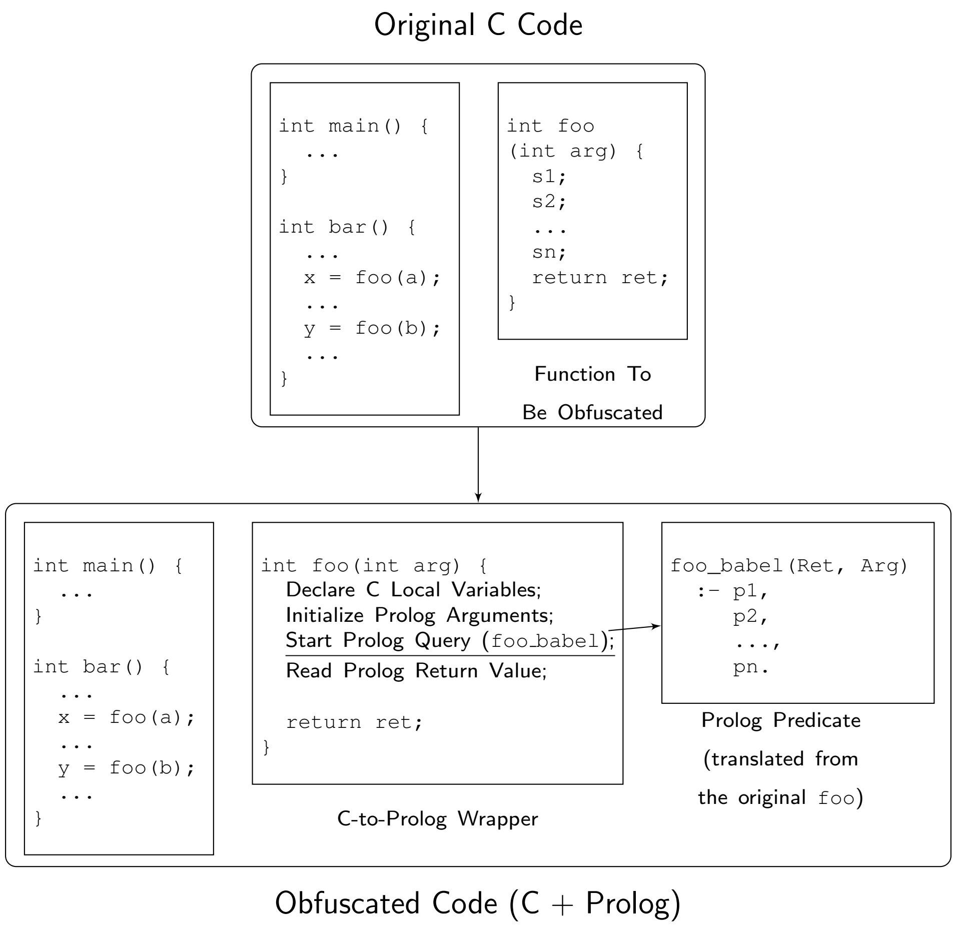
$(\texttt{foo=}e\texttt{;})^{\mathcal{T}}$	$\rightarrow (\texttt{Pfoo is } e^{\mathcal{T}})$
$(\texttt{p2=p1+intVal;})^{\mathcal{T}}$	$\rightarrow (\sigma(\texttt{p2})\texttt{ is }\sigma(\texttt{p1})\texttt{+SizeOf}(\ast\texttt{p1})\ast\sigma(\texttt{intVal}))$
$(\texttt{foo=}\ast\texttt{p;})^{\mathcal{T}}$	$\rightarrow \texttt{rdPtr}(\sigma(\texttt{foo}),\texttt{SizeOf}(\ast\texttt{p}),\sigma(\texttt{p}))$
$(\ast\texttt{p=foo;})^{\mathcal{T}}$	$\rightarrow \texttt{wrPtr}(\sigma(\texttt{p}),\texttt{SizeOf}(\ast\texttt{p}),\sigma(\texttt{foo}))$
$(\{\})^{\mathcal{T}}$	$\rightarrow (\texttt{true})$
$(\{s_1 \cdots s_n\})^{\mathcal{T}}$	$\rightarrow (s_1^{\mathcal{T}},\ \cdots,\ s_n^{\mathcal{T}})$
$(\texttt{if }(e)\ \{b_{then}\}\texttt{ else }\{b_{else}\})^{\mathcal{T}}$	$\rightarrow (e^{\mathcal{T}},\{b_{then}\}^{\mathcal{T}};\{b_{else}\}^{\mathcal{T}})$
$(\texttt{ret=fun(a1, }\cdots\texttt{,an);})^{\mathcal{T}}$	$\rightarrow \texttt{pred}(\sigma(\texttt{a1}),\cdots,\sigma(\texttt{an}),\sigma(\texttt{ret}))$
$(\texttt{ret=funptr(a1,}\cdots\texttt{,an);})^{\mathcal{T}}$	$\rightarrow \texttt{predInd}(\sigma(\texttt{funptr}),\sigma(\texttt{a1}),\cdots,\sigma(\texttt{an}),\sigma(\texttt{ret}))$
$(\texttt{return }e\texttt{;})^{\mathcal{T}}$	$\rightarrow (\mathcal{R}\texttt{ is }e^{\mathcal{T}})$
$(\texttt{fun}(T_1\texttt{ a1,}\cdots\texttt{,}T_n\texttt{ an})\{b_{body}\})^{\mathcal{T}}$	$\rightarrow \texttt{pred}(\sigma(\texttt{a1}),\cdots,\sigma(\texttt{an}))\texttt{:-}b_{body}^{\mathcal{T}}\texttt{.}$

Figure 3.7: Definition of $\mathcal{T}$, BABEL's C-to-Prolog translation. e, s, b, and T denote C expressions, statements, blocks, and types, respectively. σ is the bijective mapping from C identifiers to corresponding Prolog identifiers. $\mathcal{R}$ denotes the Prolog identifier used to hold the returned value in the translated predicate. `pred` can be either a real Prolog predicate or a wrapper of a foreign C function, depending on whether the target function is translated or not. `predInd` is a wrapper for a special foreign C function which further calls into `funptr` with given arguments.

overflows or underflows, there is a chance that our translation will fail. However, fully emulating C semantics incurs significant performance penalty.

Previous work on translating C to other languages faces the same issue, and many of them chose to ignore it [41, 102, 132]. The C-to-JavaScript converter Emscripten provides the option to fully emulate the C semantics [154]. It also has a set of optional heuristics to infer program points where full emulation is necessary, but that method is not guaranteed to work correctly. We do not particularly take this issue into account when implementing BABEL. However, we expect BABEL's translation to have a low failure chance thanks to the employment of write-to-pointer operations in Prolog and the variable flush-reload method. Since the write-to-pointer operation specifies data sizes, the truncation automatically takes place whenever an integer variable is flushed and reloaded. In GNU Prolog on 64-bit platforms, all integers are represented by 61-bit two's complement (3 bits are occupied by a WAM tag), which is large enough to hold most practical integer and pointer[3] values.

[3]Most 64-bit CPUs only implement a 48-bit virtual address space.


Original C Code
Function To Be Obfuscated
C-to-Prolog Wrapper
Prolog Predicate (translated from the original foo)
Obfuscated Code (C + Prolog)

```
int main() {
  ...
}

int bar() {
  ...
  x = foo(a);
  ...
  y = foo(b);
  ...
}
```

```
int foo
(int arg) {
  s1;
  s2;
  ...
  sn;
  return ret;
}
```

```
int foo(int arg) {
  Declare C Local Variables;
  Initialize Prolog Arguments;
  Start Prolog Query (foo_babel);
  Read Prolog Return Value;

  return ret;
}
```

```
foo_babel(Ret, Arg)
  :- p1,
     p2,
     ...,
     pn.
```

Figure 3.8: The context for executing obfuscated code in Babel.

3.6.2 Combining C and Prolog

Babel combines the C and Prolog runtime environments together, and the program starts from executing C code. When the execution encounters an obfuscated function (which is now a wrapper for initiating queries to the corresponding Prolog predicate), it setups a context prior to evaluating the Prolog predicate. In the setup process the wrapper allocates local variables whose addresses are referred to in the preprocessed C function. The wrapper then passes the addresses along with function arguments to the Prolog predicate through the C-to-Prolog interface provided by GNU Prolog. Figure 3.8 illustrates how the two languages are combined.

3.6.3 Customizing Prolog Engine

Although GNU Prolog has some nice features that make it a mostly adequate candidate for implementing Babel, it still does not fully satisfy our requirements, thus requiring some customization. A notable issue about GNU Prolog is that its interface for calling Prolog from C is not reentrant. This is critical because by design, users of Babel can freely choose the functions they want to obfuscate. To support this, it is in general not possible to avoid stack traces that interleave C and Prolog subroutines. We found that the non-reentrant issue results from the use of a global WAM state across the whole GNU Prolog engine. We fixed it by maintaining a stack to save the WAM state before a new C-to-Prolog interface invocation and restore the state after the call is finished.

Another issue is that GNU Prolog does not implement garbage collection; therefore memory consumption can easily explode. This problem is not as severe as it looks because we do not have to maintain a heap for Prolog runtime throughout the lifetime of the program. Because we know that the life cycles of all Prolog variables are bounded by the scope of predicates, we can safely empty the Prolog heap when there are no pending Prolog subroutines during the execution. Since GNU Prolog implements the heap as a large global array and indicates heap usage with a heap-top pointer, we can empty the heap by simply resetting the heap-top pointer to the starting point of the heap array, which is very efficient.

3.7 Evaluation

Collberg et al. [52] proposed to evaluate an obfuscation technique with respect to four dimensions: *potency*, *resilience*, *cost*, and *stealth*. Potency measures how obscure and complex the program has become after being obfuscated. Resilience indicates how well programs obfuscated by Babel can withstand reverse engineering effort, especially automated deobfuscation. Cost measures the execution overhead imposed by obfuscation. Stealth measures the difficulty in detecting the existence of obfuscation, given the obfuscated binaries. We evaluate BABEL and observe to what extent it meets these four criteria.

To show that our tool can effectively protect real-world software of different categories, we apply BABEL to six open source C programs that have been widely deployed for years or even decades. Among the six programs, four are CPU-bound applications and the other two are IO-bound servers. The CPU-bound applications include algebraic transformation (bzip2), integer computation (mcf), state machine (regexp), and floating-point computation (svm_light). The two IO-bound servers cover two of the most popular network protocols, i.e., FTP (oftpd) and HTTP (mongoose). We believe that our selection is a representative evaluation set covering a wide range of real-world software. Table 3.1 presents the details of these programs.[4]

We define the term *obfuscation level* as the percentage of functions obfuscated in a C program. For example, an obfuscated bzip2 instance at the 20% obfuscation level is a bzip2 binary compiled from source code consisting of 80% of the original functions in C and Prolog predicates translated from the other 20% C functions by BABEL. We achieve all obfuscation levels by randomly selecting candidates from all functions that can be obfuscated by BABEL, but note that this random selection scheme is just for avoiding subjective picking in our research. In practice, BABEL users should decide which functions are critical and in need of protection. This is the same as popular commercial virtualization-based obfuscation tools [7, 28].

In the evaluation, we compare BABEL with one of the most popular commercial obfuscators, Code Virtualizer (CV) [7], which is virtualization based and has

[4]We notice that some previous work [45] on obfuscation employed the SPEC benchmarks or GNU Coreutils, which are also widely used in other research, for evaluation. Unfortunately, these two software suites use very complicated build infrastructures. Since BABEL needs to compile C and Prolog together, a specialized build procedure is required. Currently our prototypical implementation of BABEL cannot automatically hook an existing build system, so we are not able to include SPEC or GNU Coreutils into our evaluation.

Table 3.1: Programs used for Babel evaluation.

Program	Description	LoC	# of Func.
bzip2	Data compressor	8,117	108
mcf	Vehicle scheduler	2,685	25
regexp	Regular expression engine	1,391	22
svm_light	Support vector machine	7,101	103
oftpd	Anonymous FTP server	5,211	96
mongoose	Light-weight HTTP server	5,711	203

Table 3.2: Program complexity before and after Babel obfuscation at 30% obfuscation level

Program	# of Call Graph Edges			# of CFG Edges			# of Basic Blocks			Cyclomatic Number			Knot Count		
	Original	Babel	Ratio	Original	Babel	Ratio	Original	Babel	Ratio	Original	Babel	Ratio	Original	Babel	Ratio
bzip2	353	5964	16.9	5382	19771	3.7	3528	17078	4.8	1856	2695	1.5	3120	12396	4.0
mcf	78	5449	69.9	854	14233	16.7	583	13086	22.4	273	1149	4.2	153	8792	57.5
regexp	72	5276	73.3	855	13290	15.5	591	11802	20.0	266	1490	5.6	1135	9530	8.4
svm	511	6739	13.2	5375	20752	3.9	3545	18533	5.2	1832	2221	1.2	2972	11521	3.9
oftpd	455	5810	12.8	2035	15501	7.6	1667	14422	8.7	370	1081	2.9	1277	9911	7.8
mongoose	1027	6762	6.6	2788	17115	6.1	2086	16079	7.7	704	1038	1.5	493	9491	19.3
Geom.Mean.	279.2	5972.7	21.4	2220.4	16555.5	7.5	1570.2	14987.8	9.5	633.0	1502.4	2.4	1002.3	10199.1	10.2

been on the market since 2006. The comparison covers all the four dimensions of evaluation, but some of the evaluation methodologies we designed for Babel may not be suitable for evaluating Code Virtualizer. For those evaluations that we consider not suitable for CV, we will explain the reasons and readers should be cautious in interpreting the data.

3.7.1 Potency

We use two groups of static metrics to show how much complexity Babel has injected into the obfuscated programs. The first group consists of basic statistics about the call graph and control-flow graph (CFG), including the number of edges in both graphs and the number of basic blocks. These metrics have been used to evaluate obfuscation techniques in related work [45].

In addition to basic statistics, we also calculate two metrics used to quantify program complexity, proposed by historical software engineering research. The measures are the cyclomatic number [103] and the knot count [143]. Both metrics reflect Gilb's statement that logic complexity is a measure of the degree of decision

Table 3.3: Program complexity before and after Code Virtualizer (CV) obfuscation at 30% obfuscation level

Program	# of Call Graph Edges			# of CFG Edges			# of Basic Blocks			Cyclomatic Number			Knot Count		
	Original	CV	Ratio	Original	CV	Ratio	Original	CV	Ratio	Original	CV	Ratio	Original	CV	Ratio
bzip2	353	261	0.7	5382	3868	0.7	3528	2826	0.8	1856	1044	0.6	3120	713	0.2
mcf	78	34	0.4	854	461	0.5	583	329	0.6	273	134	0.5	153	68	0.4
regexp	72	66	0.9	855	525	0.6	591	377	0.6	266	150	0.6	1135	603	0.5
svm	511	357	0.7	5375	3267	0.6	3545	2358	0.7	1832	911	0.5	2972	302	0.1
oftpd	455	390	0.9	2035	1727	0.8	1667	1435	0.9	370	294	0.8	1277	542	0.4
mongoose	1027	585	0.6	2788	2063	0.7	2086	1638	0.8	704	427	0.6	493	442	0.9
Geom.Mean.	279.2	190.4	0.7	2220.4	1489.0	0.6	1570.2	1117.0	0.7	633.0	365.9	0.6	1002.3	358.3	0.3

making within a system [74]. They also have been considered for evaluating obfuscation effects [52]. The cyclomatic number is defined as $e - n + 2$ where e and n are the numbers of edges and vertices in the CFG. Intuitively, the cyclomatic number represents the amount of decision points in a program [53]. The knot count is the amount of edge crossings in the CFG when all nodes are placed linearly and all edges are drawn on the same side.

Table 3.2 shows the comparison between binaries with and without Babel obfuscation on the complexity measures we have chosen, at the obfuscation level of 30%. Readers can refer to Appendix for potency evaluation data at other obfuscation levels, i.e., from 10% to 50%. To be conservative, by the time of measurement we have stripped the code belonging to GNU Prolog runtime itself, so the extra complexity (if there is any) should be purely credited to Babel's obfuscation. We use IDA Pro [15], an advanced commercial reverse engineering tool, to disassemble the binary and generate call graphs and control-flow graphs. As can be seen, the obfuscated binaries have a significant advantage on all metrics. The geometric mean of all six programs shows that Babel can vastly increase program complexity. Note that different from static complexity produced by obfuscation methods using opaque predicates, the additional control-flow branches injected by Babel are "real" in the sense that all branches can be feasible at run time.

Our potency evaluation is not an ideal methodology for measuring the same aspect of Code Virtualizer. The reason is that Code Virtualizer transforms binary instructions into bytecode and the reverse engineering tool we use, i.e., IDA Pro, is incapable of handling this situation. Table 3.3 shows the complexity of binaries with and without Code Virtualizer obfuscation, obtained in the same way as Table 3.2. The data suggests that after Code Virtualizer is applied, the complexity of the protected binaries has a notable decrease compared to the original ones. In reality, this is a consequence of the aforementioned issue. Because IDA Pro is unable to analyze the re-encoded functions, the potency evaluation inevitably misses a significant portion of the complexity.

3.7.2 Resilience

In general, the resilience of an obfuscation technique is hard to assess because reverse engineering can be a very ad-hoc process. On the other hand, very few practical deobfuscation tools are publicly available to the community. Because of these difficulties, some previous work on software obfuscation failed to evaluate resilience properly. In our evaluation, we choose binary diffing to assess BABEL's resilience after intensive investigation, although it does not mean binary diffing is the only deobfuscation technique.

Binary diffing is a commonly used reverse engineering technique which calculates the similarity between two binaries. We consider binary diffing a deobfuscation technique because it reveals the connection from an obfuscated program to its original version. Given a program binary and its obfuscated version, if a binary diffing tool reports high similarity score for the comparison (ignoring potential false positives), then in some sense the differ has successfully undone the obfuscation effect, Most historical work on deobfuscation known to us uses similarity-based metrics to evaluate the effectiveness of their techniques [126, 54, 152]. Deciding the similarity of untrusted programs, especially binaries, has been such an important topic in computer security that DARPA has initiated the four year, $43 million Cyber Genome Program to support related research [9].

Binary similarity can be calculated based on either syntax or semantics. The syntax mostly refers to the control flows of the binary and syntax-based binary diffing usually takes a graph-theoretic approach which compares the call graphs of two binaries and further the control flow graphs of pairs of functions between two binaries, looking for any graph or subgraph isomorphism. The intuition is, if two binaries have similar call graphs, the functions located at corresponding nodes in the call graph isomorphism are likely to be similar ones; if two functions have similar control flows, they are likely to implement the same computation logic.

On the other hand, semantics-based binary diffing focuses more on the observable behavior of the binaries. There are various ways to describe program behavior, e.g., the post- or pre-condition of a given chunk of code and certain effects the code commits such as system calls. If two binaries have matched behavior, a semantics-based binary diffing tool will consider them similar.

In general, syntax-based similarity is less strict than semantics-based similarity. Relatively, syntax-based differs tend to report more false positives while semantics-based differs tend to get more false negatives. To avoid bias as much as possible in the evaluation, we pick binary differs of both kinds to test BABEL's resilience to reverse engineering. We employ CoP [97] and BinDiff [4], of which CoP is a semantics-based binary differ and BinDiff is syntax based [63, 131]. To measure BABEL's resilience to a differ, we randomly pick 50% functions from each program in Table 3.1, obfuscate them with BABEL, and then launch the differs to calculate the similarity between the original and obfuscated functions.

3.7.2.1 Resilience to Semantics-Based Binary Diffing

CoP, a "*semantics-based obfuscation-resilient*" binary similarity detector [97], is currently one of the state-of-the-art semantics-based binary diffing tools. The detection algorithm of CoP is founded on the concept of "longest common subsequence of semantically equivalent basic blocks." By constructing symbolic formulas to describe the input-output relations of basic blocks, CoP checks the semantic equivalence of two basic blocks with a theorem prover. It is reported that this new binary diffing technique can defeat many traditional obfuscation methods. CoP is built upon several cutting-edge techniques in the field of reverse engineering, including the binary analysis toolkit BAP [39] and the constraint solver STP [68]. CoP defines the similarity score as the number of matched basic blocks divided by the count of all basic blocks in the original function.

Figure 3.9 is the box plot showing the distribution of similarity scores. For all programs, the third quartile of the scores is below 20%. Considering that the original paper of CoP reports over 70% similarity in most of their tests on transformed or obfuscated programs, the scores calculated from Babel-obfuscated functions are not convincing evidence of similarity. One may notice that there are a few outliers in Figure 3.9, i.e., the similarity scores for some functions can reach 100%. These functions are all "simple" ones, namely they have only one basic block and very few lines of C code. With the presence of false positives, it is very likely that the binary differ can report 100% similarity for these functions, according to CoP's similarity score definition.

3.7.2.2 Resilience to Syntax-Based Binary Diffing

BinDiff is a proprietary syntax-based binary diffing tool which is the de facto industrial standard with wide availability. It has motivated the creation of several academia-developed binary differs such as BinHunt [69] and its successor iBinHunt [104].

Given two binaries, BinDiff will give a list of function pairs that are considered similar based on a set of different algorithms. In addition to the similarity level like CoP reports, BinDiff also reports its "confidence" on the results, based on which algorithm is used to get that score. It is not completely clear to us how each of BinDiff's algorithms works and how BinDiff ranks the confidence level. Therefore,

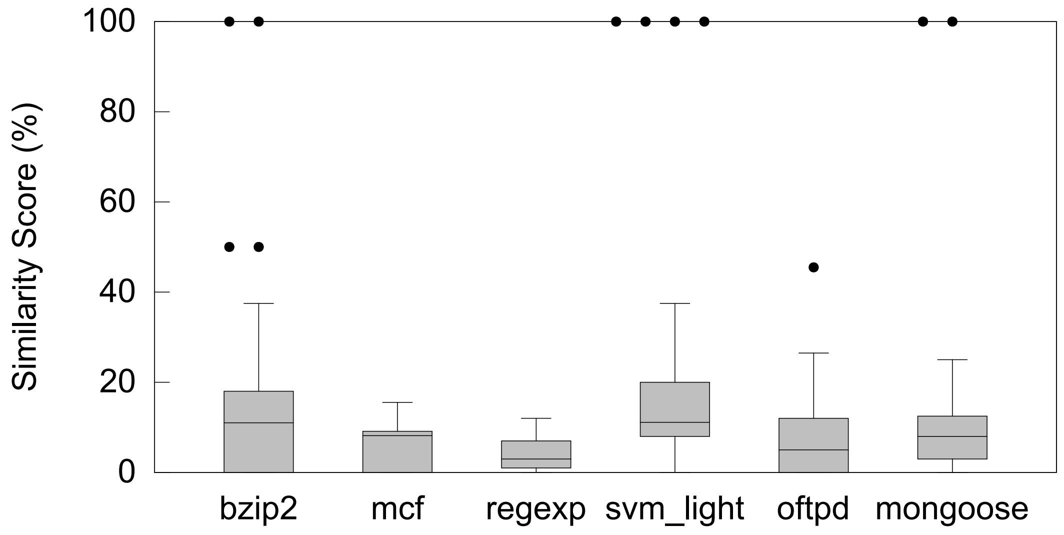


Figure 3.9: Distributions of similarity scores between the original and Babel-obfuscated functions in the evaluated programs.

we report how many obfuscated functions are correctly matched to their originals by BinDiff regardless of the similarity score and the confidence. This makes sure Babel does not take any unfair advantage over its opponent in the evaluation of performance. In other words, the results reported here indicate a lower bound of Babel's resilience to syntax-based binary diffing.

Table 3.4 shows how many obfuscated functions in each program are matched (although some of them get low similarity scores or confidence). Since BinDiff can match functions solely based on their coordinates in the call graphs, two functions can be matched even if they have totally different semantics. This explains why BinDiff can achieve a relatively high matching rate for mongoose, because mongoose has the largest number of functions and potentially has a more iconic call graph. Nevertheless, only 26.22% of the obfuscated functions are matched by BinDiff over all six programs. Note that matching does not yet imply success-

Table 3.4: Function matching result from BinDiff on BABEL-obfuscated programs

Program	# of Obfuscated	# of Matched	Match Rate
bzip2	54	6	11.11%
mcf	13	7	53.85%
regexp	11	3	27.27%
svm_light	52	10	19.23%
oftpd	48	4	8.33%
mongoose	102	39	38.24%
Overall	280	69	24.64%

Table 3.5: Function matching result from BinDiff on CV-obfuscated programs

Program	# of Obfuscated	# of Matched	Match Rate
bzip2	54	7	12.96%
mcf	13	4	30.77%
regexp	11	0	0.00%
svm_light	52	1	1.92%
oftpd	48	2	4.17%
mongoose	102	11	10.78%
Overall	280	25	8.93%

ful deobfuscation or recovery of program logic, especially for syntax-based binary differs. In that sense, we believe BABEL's performance is satisfying.

3.7.2.3 Comparing BABEL with Code Virtualizer

We present Code Virtualizer's resilience to CoP and BinDiff in Figure 3.10 and Table 3.5, respectively. The data are obtained in experiments of which the settings are consistent with the resilience evaluation on BABEL. Based on the data, it seems that Code Virtualizer is more resilient to CoP and BinDiff than BABEL. However, as aforementioned in the potency evaluation, reverse engineering binaries protected by virtualization-based obfuscators like Code Virtualizer requires specialized approaches. Since neither CoP nor BinDiff is made aware of the fact that the obfuscated parts of the binaries have been transformed from code to data, their poor performance is not a surprising result. After all, a major weakness of virtualization-based obfuscation is that although the original program may be well

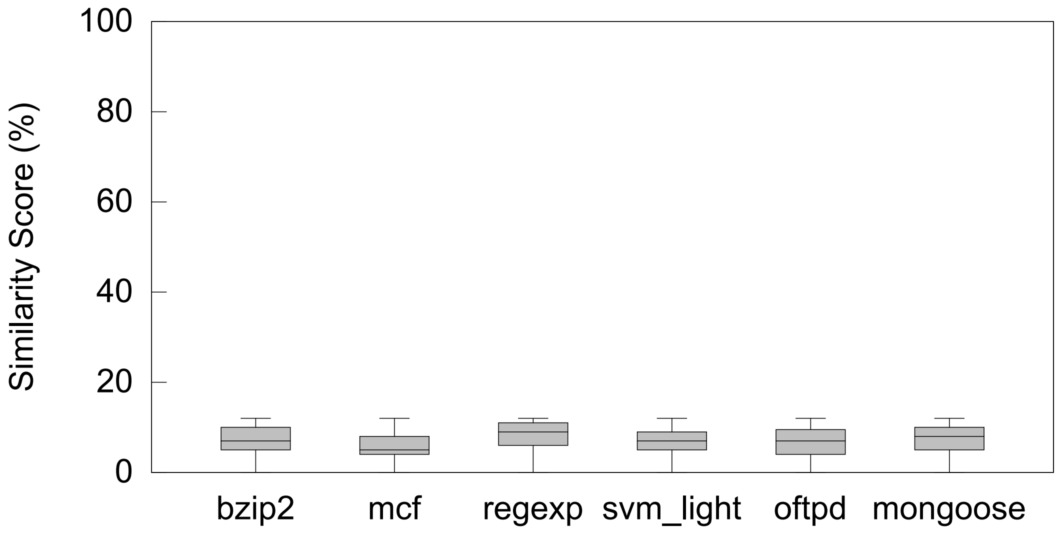


Figure 3.10: Distributions of similarity scores between the original and CV-obfuscated functions in the evaluated programs.

obfuscated, the virtual machine itself is still exposed to attacks. By reverse engineering the logic of the virtual machine and revealing the encoding format of the bytecode, attackers can effectively deobfuscate the protected binaries [126, 152].

3.7.3 Cost

We measure execution slowdown introduced by BABEL from the obfuscation level of 10% to 50%. We use the test cases shipped with the obfuscated software as the performance test input for CPU-bound applications. For FTP server oftpd, we transfer 10 files ranging from 1KB to 128MB. For HTTP server mongoose, we sequentially send 100 quests for a 2.5KB HTML page. We conduct the experiments on a desktop with Xeon E5-1607 3.00GHz CPU and 4GB memory running 64-bit Ubuntu 12.04 LTS, over a 1Gbps Ethernet link. We run each test 10 times and report the average slowdown. For servers, time spent on network communication is included by our measurement.

Table 3.6: Time overhead introduced by Babel and Code Virtualizer (CV)

Program	10% obfuscated			20% obfuscated			30% obfuscated			40% obfuscated			50% obfuscated		
	Coverage	Slowdown		Coverage	Slowdown		Coverage	Slowdown		Coverage	Slowdown		Coverage	Slowdown	
		Babel	CV		Babel	CV		Babel	CV		Babel	CV		Babel	CV
bzip2	0.00%	1.5	1.9	0.00%	1.5	1.9	27.78%	8.0	×	33.34%	100.9	×	33.34%	105.8	×
mcf	0.66%	1.0	12.0	4.30%	6.9	52.4	15.54%	65.8	×	69.33%	135.9	×	83.33%	169.3	×
regexp	18.33%	138.5	660.9	19.74%	173.8	834.8	19.74%	198.7	1122.2	28.91%	285.9	×	28.91%	288.8	×
svm_light	13.33%	1.0	58.0	20.00%	4.0	×	26.67%	4.1	×	26.67%	4.1	×	33.33%	11.8	×
oftpd	-	1.0	1.0	-	1.1	1.1	-	1.1	1.1	-	1.1	×	-	1.1	×
mongoose	-	1.0	1.4	-	1.2	8.8	-	1.6	9.3	-	1.7	×	-	2.1	×

Coverage denotes the percentage of CPU time taken by the obfuscated functions in the execution of the original programs (not available for IO-bound servers). The percentage is a lower bound because some functions may be inlined into others. × indicates that the corresponding test failed due to program crash or incorrect output.

Unlike potency and resilience, we can easily conduct a fair comparison between Babel and Code Virtualizer on execution overhead. In the comparison, we configure Code Virtualizer to minimize obfuscation strength and maximize execution speed. The implication of our comparison setting is that, if Babel can achieve comparable or better performance than a mature commercial product, the runtime overhead introduced by Babel should be acceptable for practical use. To show that the functions we obfuscate are non-trivial, we use `gprof` to get their performance coverage, i.e., the percentage of CPU time taken by the obfuscated functions in the execution of the original programs. Note that the percentage we obtain is just a lower bound because some functions may be inlined and `gprof` will contribute their execution time to other functions. We are only able to get the coverage data for the four CPU-bound applications, because the CPU time spent by the two original server programs is too short for meaningful profiling. Profiling server programs usually requires dedicated profilers and we are unaware of the existence of such tools for the two server programs we picked.

Table 3.6 gives the experiment results, showing that Babel outperforms Code Virtualizer in most of the cases we tested. In particular, Babel's obfuscation is more reliable in the sense that the obfuscated programs exit normally and give correct output on test input, while Code Virtualizer fails to provide reliable obfuscation on most of the tested programs when obfuscation level reaches 40%. Both Babel and Code Virtualizer impose considerably high performance overhead after

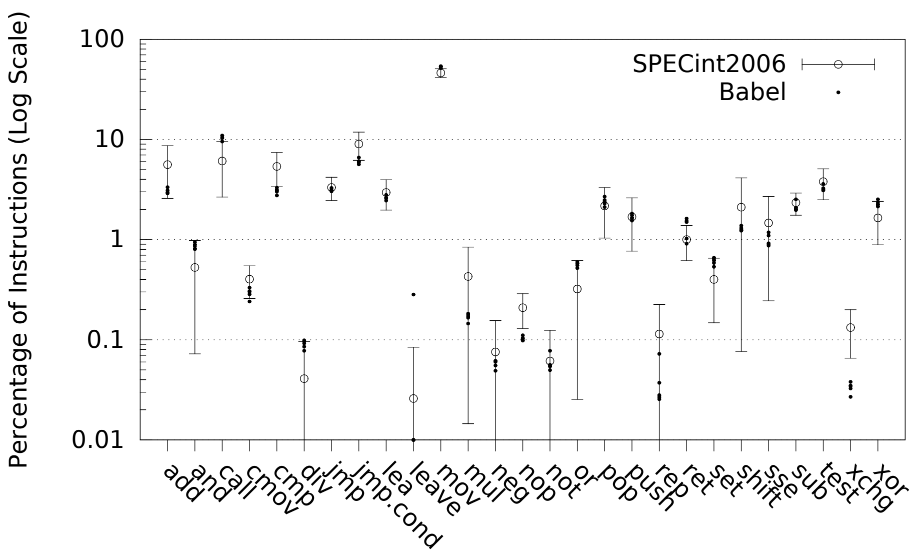


Figure 3.11: Instruction distributions of SPECint2006 programs (mean and standard deviation) and Babel-obfuscated integer programs.

the obfuscation level reaches 30%, for many of the CPU-bound applications. This is expected, because in general such heavy-weight obfuscation methods should be avoided when protecting program hot spots [22]. In the evaluation, the coverage of many applications exceeds 30% after the obfuscation level reaches 50%, which is rarely the case if Babel and Code Virtualizer are to be deployed in practice. Regardless, the key point of this evaluation is to demonstrate that Babel's performance cost is lower than a industry-quality obfuscator which shares certain similarity with Babel.

3.7.4 Stealth

By evaluating stealth we investigate whether Babel introduces abnormal statistical characteristics to the obfuscated code. In stealth evaluation, we pick the 30% obfuscation level.

Some previous work measures obfuscation stealth by the byte entropy of program binaries [145], for byte entropy has been used to detect packed and encrypted binaries [98]. Since Babel does not re-encode original binary code, possessing nor-

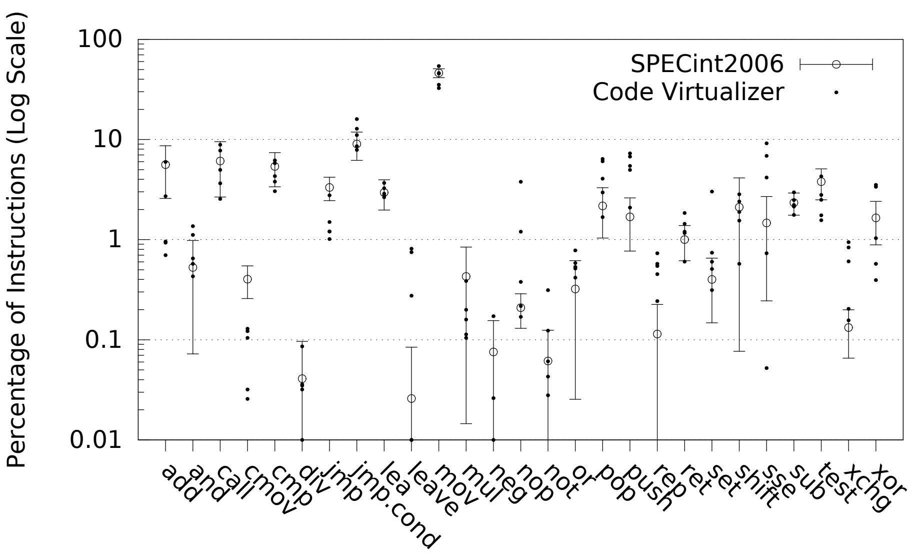


Figure 3.12: Instruction distributions of SPECint2006 programs (mean and standard deviation) and CV-obfuscated programs

mal byte entropy may not be a strong evidence of stealth. Therefore, we employ another statistical feature, the distribution of instructions, to evaluate Babel. This metric has also been employed by previous work [116, 45]. To tell whether Babel-obfuscated programs have abnormal instruction distributions, we need to compare them with normal programs. Since the scale of programs used in our evaluation is relatively small, we select the SPEC2006 benchmarks as the representatives of normal programs. We group common x86 instructions into 27 classes and calculate the means and standard deviations of percentages for each group within SPEC2006 programs. Since integer programs and floating-point programs have different distributions, we only compare bzip2, mcf, regexp, oftpd, and mongoose with SPECint2006.

Figure 3.11 presents the comparisons for integer programs. For the majority of instruction groups, their distributions in Babel-obfuscated programs fall into the interval of normal means minus/plus normal standard deviations. There are some exceptions such as `mov`, `call`, `ret`, `cmp`, and `xchg`. However, their distributions are still bounded by the minimum and maximum of SPEC distributions (not shown in the figure). Hence, we believe these exceptions are not significant enough to conclude that BABEL-obfuscated programs are abnormal in terms of instruction distribution.

Meanwhile for binaries obfuscated by Code Virtualizer, the instruction distributions are significantly more deviant, as shown by Figure 3.12. It should be noted that when we tried to disassemble binaries processed by Code Virtualizer, the disassembler reports hundreds of decoding errors, presumably because Code Virtualizer transforms legal instructions to bytecode which cannot be correctly decoded by the disassembler. Nevertheless, this disassembly anomaly itself can also be strong evidence of obfuscation. Overall, the experiments indicate that BABEL has better stealth performance than Code Virtualizer.

There may be of a concern that solely the existence of a Prolog execution environment can be the evidence of obfuscation. This can be tackled by developing a customized Prolog engine. Previous work has shown that a Prolog engine can be implemented with less than 1,000 lines of Pascal or C code [83, 144].

3.8 Discussion

3.8.1 Generalizing Translingual Obfuscation

Although it is usually quite challenging to translate programs in one language to another language with very different syntax, semantics, and execution models, many of the obstacles can be circumvented when the translation is for obfuscation purposes and not required to be complete. In our translation from C to Prolog, we designate the task of supporting C memory model, which is one of the most challenging issues in translating C, partially to the C execution environment itself. This solution is not feasible in general-purpose language translations. Meanwhile, some of our translation techniques are universally applicable to a class of target languages that share certain similarities. For example, the control flow regularization methods we proposed can be adopted when translating C to many declarative programming languages. We believe that translingual obfuscation has the potential to be made a general framework that supports various source and target languages.

3.8.2 Multithreading Support

Our current implementation of Babel does not support C multithreading, and the main reason is that some components of GNU Prolog are not thread safe. Since GNU Prolog is a Prolog implementation for research and educational use, some language features are not supported. However, many other Prolog implementations that are more mature can indeed support multithreading well [26]. By investing enough engineering effort, we should be able to improve the implementation of GNU Prolog and ensure that it supports concurrent programming. Therefore, we do not view the current limitation as a fundamental one.

3.8.3 Randomness

Some obfuscation techniques improve the security strength by introducing randomness. For example, the virtualization-based obfuscators usually randomize the encoding of their virtual instruction set [62] so that attackers cannot crack all randomized binaries by learning the encoding of a single instance. Although this randomization is ineffective once attackers learned how to systematically crack the virtual machine itself, the idea of randomization does have some value.

Our current design of translingual obfuscation does not explicitly feature any randomness. However, since translingual obfuscation is orthogonal to existing obfuscation techniques, it can be stacked with those techniques that do introduce randomness. Translingual obfuscation itself has the potential to feature randomness as well. One promising direction could be making some of the foreign language compilation strategies undeterministic. Previous research [61] has shown that mutating compilation configurations can effectively disrupt some deobfuscation tools.

3.8.4 Defeating Translingual Obfuscation

In general, translingual obfuscation is open design and does not rely on any secrets, although it can be combined with other secret-based obfuscation methods. All of our justification on the security strength of translingual obfuscation assumes that attackers do possess the knowledge that we have translated C into Prolog. Indeed, with this knowledge attackers can choose to convert the binary to Prolog first rather than directly getting back to C. Either way, attackers will face severe challenges.

We would like to emphasize again that we do not argue it is impossible to defeat translingual obfuscation. Instead, we argue that Prolog is more difficult to crack than C, in the translingual obfuscation context. As long as a Babel-translated Prolog predicate is compiled as native code, recovering it to a high-level program representation faces all the difficulties encountered in C reverse engineering, including the hardness of disassembly and analysis [139]. In Chapter 3.3 we revealed the deep semantics gap between Prolog source code and its low-level implementation. Thanks to this gap, we expect that recovering the computation logic of native code compiled from Prolog-translated C source code will consume a significant amount of reverse engineering effort.

What makes defeating translingual obfuscation even more challenging is that, the obfuscated code is not only a plain combination of normal Prolog plus normal C but a tangled mixture of both. The execution of obfuscated programs will switch back and forth between the two language environments and there will be frequent interleaving of different memory models (see Chapter 3.5). This also imposes challenges to reverse engineering.

There is another point that grants translingual obfuscation the potential to significantly delay reverse engineering attacks. As stated in Chapter 3.1, translingual obfuscation is not limited to Prolog. There are many other programming languages that we can misuse for protection. By mixing these languages in a single obfuscation procedure, the difficulty of reverse engineering will be further increased.

Chapter 4

Status Quo of Obfuscation in Mobile Development

Although obfuscation-related research topics have been intensively studied for decades, most previous work focused on in-lab technical analysis of the effectiveness of new obfuscation techniques [136, 115, 140, 88, 146] or countermeasures against obfuscation when it is misused by malware writers [152, 37]. As far as we have learned, little emphasis is put on investigating how benign software authors take obfuscation as part of their development process in the real world, which is critical for software obfuscation techniques to be practical. To push this line of research forward, we aim to investigate the answers to the following important research questions:

- **RQ1:** *What are the characteristics of obfuscated mobile apps?*
- **RQ2:** *In what patterns are mobile apps typically obfuscated?*
- **RQ3:** *How does app review affect the adoption of obfuscation?*

The work of this chapter is published in *Proceedings of the 40th International Conference on Software Engineering*, 2018 [135].

- **RQ4:** *How resilient are the obfuscated apps to malicious reverse engineering?*

To develop meaningful conclusions, it is most adequate to conduct an empirical study on a reasonably large set of recently developed and supposedly benign mobile apps obfuscated by their vendors. Unfortunately, there is no such a data set available for public access, so we decided to collect samples independently. There are currently two major platforms in mobile software markets, i.e., iOS and Android. Although they share many common characteristics, there are also notable differences. Some previous research has indirectly or implicitly touched the topic of mobile app obfuscation, but the focus is mostly on Android. For example, the study by Zhou and Jiang on Android malware revealed some obfuscated samples [156]. Linares-Vásquez et al. [93] and Glanz et al. [75] investigated Android app repacking, with the potential disturbance of obfuscation considered. On the other hand, the iOS platform received notably less attention which mismatches its share in the market. With over a billion iOS mobile devices sold, there are reportedly millions of software programmers working on iOS app development. Previous research on mobile software engineering revealed that obfuscation has been a common practice on Android [156, 93, 91], yet the figure for iOS is mostly missing. Since iOS is typically considered a more secure system than Android for being more closed, it may be susceptible that obfuscation on iOS could be as prevalent as on Android. However, some recent security incidents have shown that with the help of production-quality binary analysis tools like IDA Pro [15], iOS reverse engineering is not as difficult as it is generally recognized. For example, it is found that iOS developers similarly suffer from severe software piracy issues like Android developers [21]. It is also reported that there have been popular iOS apps being repackaged with malicious payload for stealing sensitive user data [48]. To help iOS developers counter these threats, some reputed software security solution providers have launched their iOS app obfuscation services [25].

In this study, we chose to work on iOS for a dual purpose of filling in the blank of empirical studies on mobile app obfuscation and enriching scientific research on this important mobile platform. For more secure iOS software engineering, it is imperative to obtain a thorough understanding about the current practice of applying obfuscation in iOS app development. The benefits of such an understanding are two-fold: vendors of obfuscation tools can better tune their development based on the status quo, while researchers interested in analyzing iOS app repositories can grasp a sense about when and how obfuscation may affect their analysis. A comprehensive study on iOS apps should be promisingly informative to audience in both the academia and industry.

To obtain a representative sample set, we crawled $1,145,582$ free iOS app instances from the official Apple App Store. We then estimated the likelihood of each instance being obfuscated based on a variant of a statistical language model previously proposed for studying software source code [79, 96]. We picked the top 6600 most likely obfuscated samples and identified 539 that are truly obfuscated with manual verification. For each sample, we further conducted in-depth investigations to understand how obfuscation was applied. In general, effectively analyzing a large amount of obfuscated binary code can be extremely difficult, since most existing program analysis techniques have either scalability or accuracy issues regarding obfuscated code. Moreover, analyzing iOS apps has its own unique challenges, one of which is caused by the wide use of *statically linked third-party libraries* [47]. To overcome these obstacles altogether, our study combined automated analysis with a considerable amount of manual effort from knowledgeable software reverse engineers with industry experience. After examining all the samples, we formulated 8 findings regarding the proposed research questions.

4.1 Background

This section introduces background knowledge about the ARM architecture and iOS mobile operating system, emphasizing platform features that make mobile software obfuscation a problem different from desktop software obfuscation. The section further elaborates on the technical challenges faced by the study.

4.1.1 The ARM Architecture

The ARM processors are currently the most widely used processing units on smartphones. ARM is a RISC architecture, meaning it has a much smaller instruction set compared to the x86 or x64 architectures, which are the major architectures used for desktops. Like most RISC instruction sets, ARM implements a fixed-with encoding for its instructions, making binary disassembly very reliable. For x86 and x64 applications, one of the most effective and widely used obfuscation techniques is manipulating binary layouts so that disassemblers are disrupted by interleaved data and instructions, overlapping instructions, and misleading control flow branches, etc. For ARM applications, however, these anti-disassembly tricks are mostly unfeasible.

There are still some anti-assembly techniques available for 32-bit ARM applications, since ARM processors supports both 32-bit and 16-bit instruction encodings in 32-bit mode and can switch back and forth at run time. By leveraging this feature, it still possible to fool disassemblers by interleaving two kinds of differently encoded instructions. However, such methods are not feasible for 64-bit ARM applications, because ARM processors requires all instructions executed in 64-bit mode to be 32-bit long and 32-bit aligned. Since most iOS devices currently on the market are powered by 64-bit processors, anti-disassembly obfuscation is not as effective as it is for protecting x86 and x64 software.

4.1.2 The iOS Mobile Operating System

The iOS operating system is the system deployed for most Apple mobile devices, including iPhones, iPads, and Apple Watches. By 2015, over 1 billion iOS devices have been sold world wide.

Most iOS applications are written in C, C++, Objective-C, and Swift. Unlike Android, iOS applications can be directly compiled to native ARM code before submitted to Apple App Store and installed on the devices.[1] Therefore, iOS developers can apply obfuscation techniques to source code, LLVM intermediate representation (IR), and binary code simultaneously.

In addition to the underlying hardware, obfuscating iOS software is also subject to iOS security policies, which are much more restrictive than desktop systems like Windows and Linux. One of the prominent iOS security features is code signing. This policy requires every executable page of an iOS application to be statically signed before the application is run on iOS devices, meaning any obfuscation technique featuring self-modification is prohibited for iOS software.[2] The code-signing policy invalidates packing obfuscation, which is considered one of the most powerful obfuscation methods in the desktop environments.

4.1.3 The Objective-C and Swift Programming Languages

Objective-C and Swift are the recommended programming languages for iOS application development. These two languages get official technical support from Apple, including the toolchain and standard libraries. Similar to C and C++, Objective-C and Swift are static languages and programs written in them are compiled to

[1]Apple Watch applications can only be submitted in the form of LLVM intermediate representation. This requirement can be waived for iPhone and iPad applications

[2]Certain applications with Apple-only entitlement can utilize dynamic code-signing to generate executable code at run time. For example, Safari is allowed to execute JavaScript with a just-in-time compiler.

native code. For that reason, iOS applications can be effectively obfuscated at the binary level.

On the other hand, Objective-C and Swift have unique features that make them different from traditional static languages like C and C++, from the perspective of program obfuscation. Essentially, Objective-C is a dynamically typed language with optional static typing. Objects in Objective-C uses a mechanism called message passing to call its member methods. This message passing mechanism resembles the run-time reflection feature in Java, which means class members are accessed and invoked through their names. To implement message passing, the Objective-C compiler has to keep the names of all class members in the binary. These names, however, can be a strong hit to human reverse engineers when they try to understand the program and pinpoint the critical part. Swift, as the latest major development language for iOS, is designed to be fully compatible with its predecessor Objective-C. Therefore, programs written in Swift also need to keep certain symbol names as strings in binaries.

4.1.4 Technical Challenges of the Study

Despite both being mobile platforms, iOS and Android are drastically different in many technical aspects. As a result, our study faces unprecedented challenges that need not to be considered by similar work targeting Android.

4.1.4.1 Obfuscation Detection and Analysis

Detecting and analyzing obfuscated binaries has long been an open research problem and is still being actively studied [105, 36, 118]. To date, the accuracy of automated obfuscation detection is not satisfying enough to fit our demand. Therefore, we decided to undertake manual analysis as the major research methodology of the study, with some light-weight automated methods as assistance.

Unlike Android developers that can use an app obfuscator embedded into the official development toolchain [24], iOS developers do not get any receive official support, thus having to rely on third-party tools or self-made obfuscators. Considering the large number of obfuscation techniques potentially available, it is impractical for an empirical study relying on manual effort to cover all of them. This poses another challenge, requiring us to identify a group of obfuscation techniques analyzable with our limited labor yet representative enough.

4.1.4.2 Static Third-Party Libraries

Third-party libraries have been an indispensable part of mobile apps. It is possible that an app "accidentally" got obfuscated due to the inclusion of obfuscated libraries without the awareness of app developers. Our analysis needs to capture such situations to avoid drawing biased conclusions.

Due to Apple's security policies, iOS apps cannot use dynamic libraries from other vendors until iOS 8, meaning all third-party libraries have to be statically linked into app executables. As a result, each object of an iOS library is scattered in the entire binary. This differs from Android apps whose third-party libraries are usually grouped as packages as a whole. The consequence is that there is no clear boundary between library code and an app's own code, making library detection in iOS apps a unique challenge [47, 113]. This is completely different from the library identification problem on Android, where application code is naturally assorted through the Java package hierarchy. Previous work on Android app analysis typically relies on Java package information to dissect an app into different parts and identify third-party libraries among them [134, 99, 91]. These methods are not applicable for iOS apps. At this point, distributing static libraries is still the mainstream practice for iOS library providers for legacy and compatibility reasons. Again, we will need manual effort to address this issue.

4.1.5 Inferring Developer Intentions

The primary goal of this research is to learn factors when and how obfuscation is applied to mobile apps, which is ultimately decided by developers subjectively. A survey-based qualitative study can provide direct evidence about developer intentions. However, the information about obfuscation, and software security in general, is very sensitive for most software vendors. Since no obfuscation technique is resilient to extensive reverse engineering [51, 35], learning the details about how obfuscation is performed grants the deobfuscation side significant advantages in the arms race. Our preliminary attempts to reaching developers known to apply obfuscation to their mobile apps mostly failed. As such, a large-scale qualitative study directly targeting developers is extremely difficult.

As such, our study is mainly quantitative and developer intentions have to be inferred through the binary code they distributed. When such cases are inevitable, we try to make as few assumptions as possible when deducing conclusions.

4.2 Methodology

We adopted a three-step process to conduct the empirical study. The first step is to select a representative collection of obfuscation techniques to consider, for reasons explained in Chapter 4.1.4.1. The second step is to search for a reasonably large set of iOS apps that are obfuscated before release. To date, there is no such a publicly available data set. Mining obfuscated samples from benign iOS apps is one of the major contributions of our work. For the third step, we inspect each obfuscated app in more depth and aggregate the harvested information to deduce empirical findings.

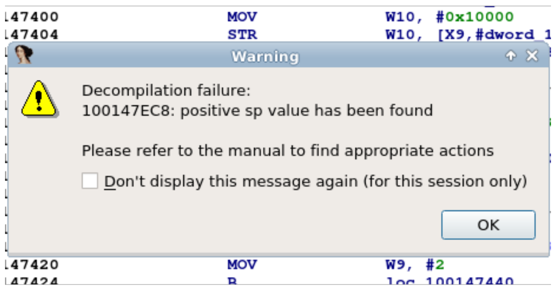
Although the study heavily relies on manual analysis of experienced reverse engineers, we indeed developed some automated techniques to assist the researchers in both sample collection and per-app examination. It should be noted that *these*

```
@interface Person: NSObject
@property NSString *name;
@property int age;
@property NSString *addr;
@end

            ⇓

@interface AlJi09: NSObject
@property NSString *KJihad;
@property int z9kmV;
@property NSString *Nm23d;
@end
```

(a) Symbol renaming

```
const char *str1 = "A␣plain␣string";
                 ⇓

// string xor masked by 0xab
const char *str1 =
  "\xea\x8b\xdb\xc7\xca\xc2\xc5\x8b"
  "\xd8\xdf\xd9\xc2\xc5\xcc\x85";
void decode(const char *s, char *d)
{
  while(*s) *d++ = *s++ ^ 0xab;
  *d = 0;
}
```

(b) Exotic string encoding



```
147400        MOV        W10, #0x10000
147404        STR        W10, [X9,#dword_1
```

Warning

Decompilation failure:
100147EC8: positive sp value has been found

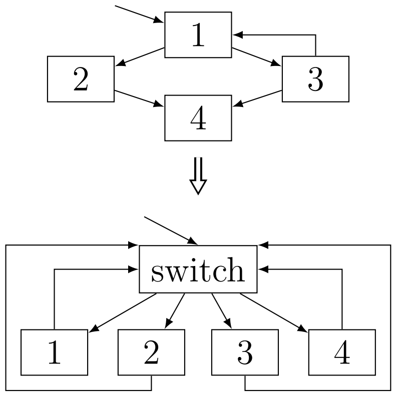
Please refer to the manual to find appropriate actions

Don't display this message again (for this session only)

OK

```
147420        MOV        W9, #2
147424        B          loc_100147440
```

(c) Decompilation disruption



(d) Control flow flattening

Figure 4.1: Illustration of obfuscation techniques considered in the study

techniques are mainly for reducing the workload of our reverse engineers. The performance of these techniques may not be ideal on their own, yet they served our design purposes well and their imperfection should not affect the validity of the final results.

4.2.1 Considered Obfuscations

After decades of development, there are now numerous obfuscation techniques available. A comprehensive review by Schrittwieser et al. [122] included 22 classes of obfuscation methods proposed by previous research. For this study, we would like to focus on obfuscations popular among mobile developers and therefore worthy of in-depth investigation. We used Google to search for commercial and open source tools that can obfuscate iOS applications. By studying the statements and technical white papers of the top 10 results, we identified four families of obfuscations that are most widely supported, i.e., symbol renaming, exotic string encoding, control flow flattening, and decompilation disruption. Compared to all known obfuscation algorithms, this is a relatively small set, with the major reason being that the unique hardware and software environment on iOS devices imposes strict restrictions on the form of executable code. For example, iOS does not allow normal user applications to dynamically generate executable code, rendering self-modifying obfuscation technically impossible to implement. It is also a set quite different from commonly studied Android obfuscations [75], due to the differences of the program languages and execution environments between the two platforms. For instance, the fake type obfuscation available for Android apps, which are written in Java, simply does not apply to iOS software. A graphical illustration of the four obfuscation algorithm families is given by Figure 4.1 while the technical details are briefly introduced as follows.

Symbol Renaming. It is recommended by common software engineering practices that programmers should make sensible names for functions and variables symbols. The preferred programming languages for developing iOS apps, i.e., Objective-C and Swift, are reflective or partially reflective. Therefore, names of many global symbols have to be retained in the distributed binaries to support by-name function dispatching at run time. Symbol renaming scrambles these names to prevent information leakage.

Exotic String Encoding. String literals sometimes disclose important information about the software. Some obfuscation algorithms convert string literals into representations that are not understandable by humans. The converted strings are decoded before use during run time.

Decompilation Disruption. It is common for obfuscations to prevent the recovery of high-level program structures from binary code. Typical methods of this kind include interleaving code and data to disturb disassembly, inserting opaque predicates to forge invalid control flows, and employing certain machine instruction patterns in unconventional ways to confuse decompilers.

Control Flow Flattening. This technique "flattens" the original control flow graph of a function by rewriting the procedure into a huge switch-like structure [89]. This makes the logical links between basic blocks obscure.

It is almost surely certain that obfuscation methods considered in the research do not include all available techniques. Nevertheless, the primary focus of our study is to investigate existing software engineering practices that can form lessons interesting to common developers as well as academic researchers. For that purpose, we have covered a good range of publicly available information when surveying obfuscation techniques accessible by general developers, Although it is possible that some developers can spend extensive effort in developing completely new obfuscation algorithms themselves, we expect that they are not the majority in the industry.

4.2.2 Mining Obfuscated iOS Apps

To obtain a reasonably large sample set without being biased, our collection starts with the entire Apple App Store. However, it should be noted that *we do not aim to find all obfuscated apps in the store.*

From February to October in 2016, we crawled $1,145,582$ free iOS app instances, *including different versions of the same app.* We then try to identify apps that are obfuscated by at least one of the four families of algorithms in Chapter 4.2.1. Ideally, we could run automated detection over all the crawled apps for each obfuscation technique subsumed by the four families. However, obfuscation detection itself is a non-trivial task and is still being actively researched [111, 114, 32, 105]. For many obfuscation algorithms considered by our study, it is prohibitively expensive, if possible at all, to automatically detect their presence in over a million instances.

To tackle this problem, we identify a *baseline obfuscation algorithm* which is supposed to be the most widely adopted in mobile development. If developers indeed consider protecting their products, it is very likely that more than one obfuscation algorithm will be employed. In such cases, detecting the baseline obfuscation can help us identify the heavily obfuscated samples. Based on this insight, we developed an automated method to identify scrambled symbol names, since symbol renaming is considered by a large volume of previous research the most prevalent obfuscation method on mobile platforms [93, 37, 91]. In practice, symbol name scrambling imposes little execution cost while being highly effective in disturbing manual analysis.

Details of the detection algorithm are presented in Chapter 4.3. After running the algorithm for all crawled app instances, we obtained the likelihood of each app being obfuscated by symbol name scrambling. Based on the available man-labor, we examined the top 6600 most likely obfuscated samples, of which 601 are conformed to be true positives by manual verification. These samples, which can be further grouped into 539 applications identified by a unique ID assigned by App Store, are taken as the data set for subsequent study. This sampling process is illustrated by Figure 4.2. We again emphasize that *these 601 samples should not be regarded as all the obfuscated apps among the* 1, 145, 582 *crawled instances*. We set the cut off at 6600 to bound the manual work within a manageable amount.

4.2.3 Per-App Inspection

In addition to symbol scrambling, we need to further confirm what other obfuscation techniques were applied to the apps. This step needs to be conducted manually to achieve the highest possible accuracy. To assure the consistency across the results from different inspectors, we developed a set of elaborate protocols to standardize the inspection process.

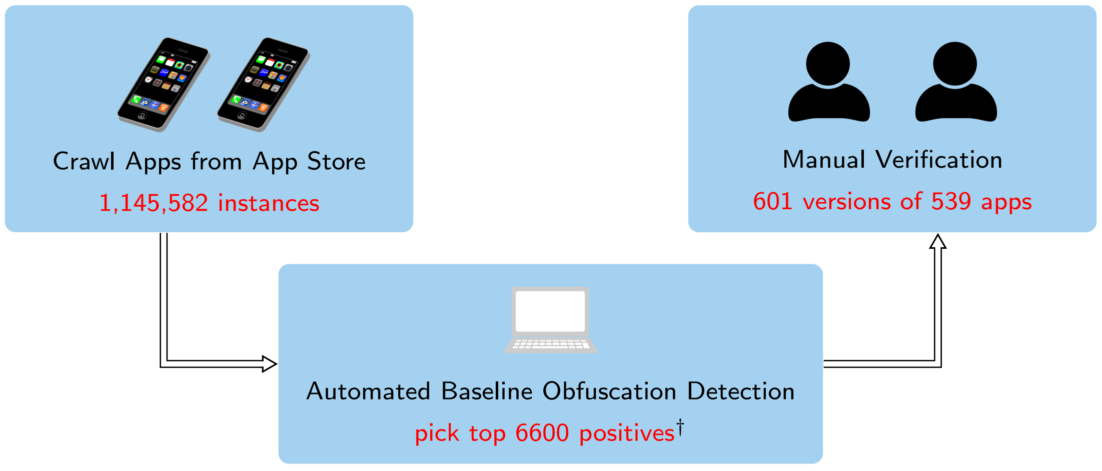

Crawl Apps from App Store
1,145,582 instances
Manual Verification
601 versions of 539 apps
Automated Baseline Obfuscation Detection
pick top 6600 positives†
†The 6600 cut off is based on the maximum labor available for manual verification

Figure 4.2: Workflow for sampling obfuscated iOS apps

4.2.3.1 Detecting Obfuscation

To detect the presence of anti-decompilation obfuscation techniques, we use IDA Pro [15], a commercial integrated reverse engineering environment that has been widely regarded as the de facto industry standard for analyzing binary code. IDA Pro can automatically dissect a binary executable into functions and translate the assembly code of each function to a high-level representation similar to C. We consider that a binary is protected by anti-decompilation techniques if IDA Pro reports too many failures. All results were manually validated.

To identify flattened control flows, we developed a binary analysis framework to disassemble app binaries and construct the control flow graph (CFG) of each function in a binary. If a CFG is flattened, most of its basic blocks will be included by a single loop, which can be captured by a standard loop detection algorithm [107]. Also, the "diameter" of the loop, which is defined as maximum length of the shortest path from the loop header to other basic blocks, should be of the logarithmic order of the total number of all basic blocks in the loop. Based on these two characteristics, we can find functions with flattened control flows.

For exotic string encoding, it is hard to develop automatic detection methods since there is no standard implementation of such techniques. In iOS executable binaries, string literals are stored in dedicated regions. We scan these regions for character sequences that cannot be decoded, or those that can be normally decoded but do not seem to possess reasonable meanings. We then manually investigate how these sequences are utilized in the code and see if they are transformed by an ad hoc decoding procedure at certain program points.

4.2.3.2 Identifying Obfuscated Third-Party Libraries

As introduced in Chapter 4.1.4.2, we need additional manual effort to identify third-party libraries in the examined iOS apps if the library code contains any obfuscation by themselves. We decide if an obfuscated code region belongs to some third-party library by observing whether there are similar code patterns appearing in multiple samples developed by different vendors. Typical signatures of code patterns include control flow graphs, special algorithms, and uncommon data structures. Once a library is detected, we try to identify its origin through public information searching, with clues such as names of library APIs and special string literals, e.g., strings used for logging and generating crash reports. Some libraries do not provide even the most subtle information that can help reveal their identities. In such cases, we extracted the semantic signatures of obfuscated code, e.g., control flow patterns and unique data structures, and check if they appear in different apps.

4.2.4 Cross-Validation

To ensure the accuracy and consistency of manual analysis, the two authors performing per-app inspections were first asked to independently examine the same 50 app instances in the sample set and compare their results. Divergences among results from different authors were discussed until an agreement was reached. The two authors then independently analyzed another 25 apps, based on the regulations made in the previous discussions. For the second round, the inspection results were consistent for all 25 apps. In this way, we established a highly accurate and cross-validated protocols for the manual analysis on obfuscated iOS apps.

4.3 Detecting Symbol Obfuscation

In practice, obfuscation tends to replace human-made symbols with randomly generated gibberish which can be detected by natural language processing (NLP) techniques. Previous research discovered that human-written source code is "natural" in the sense that it can be described by statistical language models [79]. Based on this insight, "unnatural" symbol names are possibly obfuscated.

4.3.1 An NLP-Based Detection Model

In NLP, the perplexity measure is used to quantify how "surprising" it is for a sequence of words to appear within a statistical language model. Oftentimes, the log-transformed version of perplexity, called cross-entropy, is more preferable in the literature. Given a word sequence $s = x_1 \cdots x_k$ of length k and a language model $\mathcal{M}$, the cross-entropy of s within $\mathcal{M}$ is defined as

$$H_{\mathcal{M}}(s) = -\frac{1}{k}\sum_{i=1}^{k} \log_2 P(x_i | x_1, \cdots, x_{i-1}) \tag{4.1}$$

We use cross-entropy to capture the naturalness of an identifier. Intuitively, lower $H_{\mathcal{M}}(s)$ means s is more natural within $\mathcal{M}$. In particular, we adopt the n-gram language model that assumes the word sequences suit an $(n-1)$-order Markov process. Historically, n-gram has been utilized in various software engineering applications, including automated code completion [79] and bug detection [138]. Within an n-gram model, the definition of cross-entropy can be further formulated as

$$H_{n\text{-gram}}(s) = -\frac{1}{k}\sum_{i=1}^{k} \log_2 P(x_i|x_{i-(n-1)}, \cdots, x_{i-1}) \tag{4.2}$$

A notable difference between our method and previous work is that our statistical language model is applied to individual identifiers rather than sequences of terms. As a consequence, we need to first segment an identifier into several parts before fitting it to an n-gram model. Naturally, we adopt the segmentation that makes most sense within the n-gram model by enumerating all possibilities. Therefore, the likelihood of an identifier I being "surprising", or obfuscated, can be defined by the following formula

$$L(I) = \min_{s \in S_I} H_{n\text{-gram}}(s) \tag{4.3}$$

where S_I is the set of all possible word sequences obtained by segmenting I in different ways. Given an empirically decided threshold H, we deem I as an obfuscated symbol name if $L(I) > H$.

4.3.2 Implementation

Considering that identifiers are usually not too lengthy, we can efficiently compute $L(I)$ in equation (4.3) using the Viterbi algorithm with the complexity of $O(nl^2)$, where n is length of the identifier and l is the length of the longest possible word in the language [123]. In fact, the worst cases can often be avoided, since most normal symbol names are already naturally segmented by programmers with underscores or the camel case scheme. We first compute the cross-entropy of an identifier by assuming the symbol is naturally segmented. If the entropy computed this way is already low enough, we can skip the relatively expensive Viterbi segmentation.

Our n-gram corpus contains two parts, i.e., the natural language corpus and the software source code corpus. Most identifiers in the crawled apps are named in English, but there are also many written in Chinese pinyin or even a mixture of English and Chinese. For English, we use a portion of the Google web trillion word corpus introduced by Franz and Brants [67] and derived by Norvig [112]. For Chinese, we employ the Lancaster Corpus of Mandarin Chinese (LCMC) [19]. As for the source code part, we crawled all identifiers appearing in iOS official APIs, which are all naturally segmented. Each identifier is then turned into a word sequence, thus forming a n-gram corpus.

The probability of occurrence for an n-gram is defined as the average of its probabilities in three corpora. If an n-gram does not appear in any corpus, we assign it a low probability penalized by its length. This is a necessary heuristic since there are a large number of unlisted words in program identifiers. Formally, the occurrence probability of an n-gram s is defined as

$$p(s) = \begin{cases} \dfrac{p_{\mathrm{EN}}(s) + p_{\mathrm{CN}}(s) + p_{\mathrm{code}}(s)}{3} & p_{\mathrm{EN}}(s) + p_{\mathrm{CN}}(s) + p_{\mathrm{code}}(s) > 0 \\ 20^{-(|s|-1)} \cdot 2^{-(H+1)} & p_{\mathrm{EN}}(s) + p_{\mathrm{CN}}(s) + p_{\mathrm{code}}(s) = 0 \end{cases} \tag{4.4}$$

where $|s|$ is the number of characters in the n-gram and H is the threshold defined earlier in this section.

When deciding the value of n, we observed that patterns of word sequences in different applications are quite unique and rarely occur in the corpus. The consequence is that any n greater than one leads to too many false positives. Therefore, the best option for the problem is to set n to 1, namely to adopt the unigram model.

In this study, the threshold H is set to 32.5. With this configuration, a total of 6600 positives were reported. Potentially, we could find more positives by employing a larger H, but the results then will exceed the maximum number of samples we can afford to verify. After manually examining symbols in the 6600 initial positives, we confirmed that 601 of them are truly obfuscated. The false positives are mostly caused by uses of non-English language and out-of-vocabulary abbreviations. After all, our purpose is to find a fairly large sample set of obfuscated apps as the tarting point of the empirical study rather than identifying all the true positives.

4.4 Findings

In this section we present 8 findings of our empirical study, grouped by their relevance to research questions raised at the beginning of the chapter.

4.4.1 Characteristics of Obfuscated Apps

We first discuss what factors might lead to the adoption of obfuscation in mobile app development.

Finding A.1. *A considerable portion of apps containing obfuscation are "passively" obfuscated due to the inclusion of obfuscated third-party libraries.*

As previously mentioned, we paid special attention to third-party libraries when inspecting the obfuscated apps. The examination shows that these libraries indeed make a major source of obfuscation. In total, we captured 35 third-party libraries. The major functionality of each library, inferred by analyzing their code and retrieving publicly available information on the Web, is presented in Table 4.1.

Figure 4.3 shows the breakdown of the origins of obfuscated code in the samples. Among the 539 apps employing obfuscation, 404 (75%) of them include at least one obfuscated third-party library. In particular, for 344 (63.8%) apps, the obfuscation is solely introduced by libraries. The popularity of these libraries can be further demonstrated in two aspects. Figure 4.4a shows for each library the number of including apps and Figure 4.4b shows the distribution of apps including obfuscated third-party libraries regarding the number of libraries.

Figure 4.3 indicates that the occurrences of obfuscation are mainly caused by the practice of depending on third-party libraries rather than app developers

Table 4.1: Obfuscated Libraries Grouped by Functionality

Functionality	Count	Including Apps
Advertising & Promotion	9	259
Security & Authentication	7	17
Digital Right Management	6	53
Payment & Banking	5	101
Location	2	11
Visualization	2	11
Analytics	1	19
Fraud Detection	1	17
Peripheral Control	1	3
Speech-to-Text	1	8

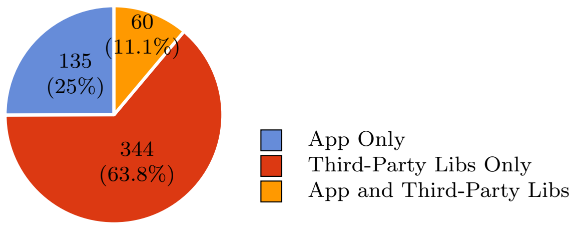
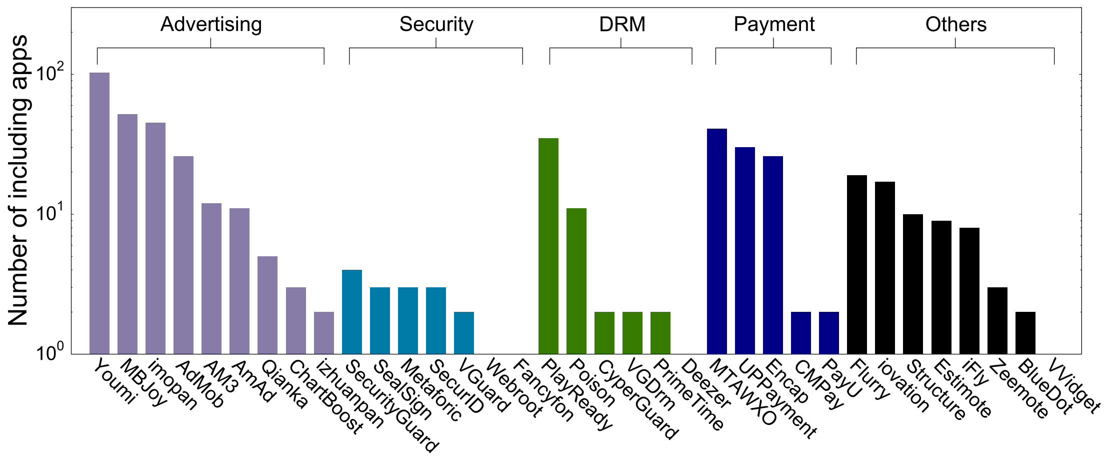


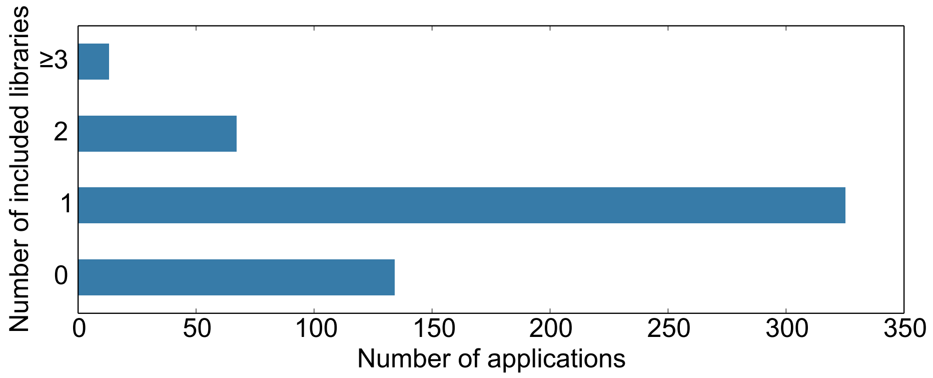
Figure 4.3: Origins of obfuscation in 539 obfuscated apps

actively considering software protection. Based on the observation, we believe that it is important to consider the impact of third-party libraries for empirical software engineering research whenever app obfuscation is involved. To distinguish different sources of obfuscation, we henceforth call an app is *actively obfuscated* if its obfuscation is *not* entirely contributed by third-party libraries; otherwise it is called *passively obfuscated*.

The most notable kind of third-party libraries is for advertising purposes with both metrics being the highest in Table 4.1. Our preliminary analysis on some of these libraries shows that the obfuscated parts are used for communicating with the back-end ad servers. It is known that mobile advertising has been bothered by reverse engineering, through which a malicious party instruments advertising libraries to forge fake advertisement display or user clicks and tricks ad providers into paying in vain [92]. For ad providers, obfuscating their libraries is a reasonable response to such malicious attempts.



(a) Number of apps including each third-party library



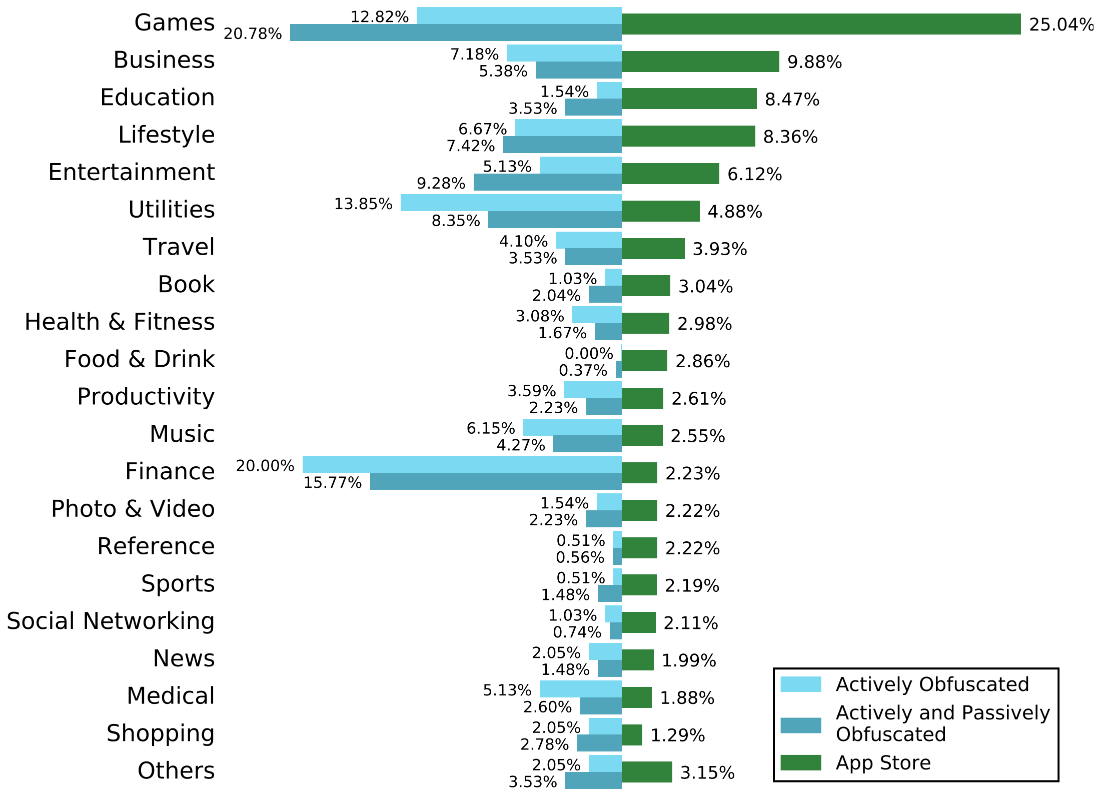
(b) Distribution of apps regarding the number of obfuscated libraries included

Figure 4.4: Popularity of obfuscated third-party libraries

Finding A.2. *The likelihood of apps and libraries being obfuscated is strongly correlated to their categories of functionality.*

We found that in contrast to the distribution of all apps in App Store regarding their categories, the distribution of obfuscated apps has a vastly different pattern. This pattern varies further when the impact of third-party libraries is considered. Figure 4.5 shows the differences between these distributions, leading to the following key observations:

- The proportions of obfuscated apps in certain categories are exceptionally high compared to the shares of all apps in these categories across App Store,



Data for App Store from Statista [2]

Figure 4.5: Distributions of apps regarding their categories

no matter whether passive obfuscation is taken into account. These categories are Finance (20.00%/15.77% vs. 2.23%), Utilities (13.85%/8.35% vs. 4.88%), Music (6.15%/4.27% vs. 2.55%), and Medical (5.13%/2.60% vs. 1.88%). According to our investigation, most of the obfuscated Music apps provide streaming services for copyrighted musical contents. The inspected Utilities apps are mainly toolkit software providing assistance to daily activities, the majority of which regularly record user data that may be closely tied to personal privacy or enterprise secrets.

- For some other categories, the situation is flipped, namely the proportions of apps carrying obfuscated code are significantly lower than the store-wide ratios. Categories of such include Education (1.54%/3.53% vs. 8.47%), Book

(1.03%/2.04% vs. 3.04%), Food & Drink (0.00%/0.37% vs. 2.86%), and Reference (0.51%/0.56% vs. 2.22%).

- The distributions of obfuscated apps in the Games, Finance, and Utilities categories are heavily influenced by obfuscated third-party libraries. Apps in the Games category are easily passively tainted by libraries. The Finance and Utilities apps, on the other hand, have a relatively higher rate for being actively obfuscated.

The first two points suggest that mobile apps related to health, finance, privacy, and intellectual property safety are more likely to get obfuscated, both actively or passively. Despite being a fairly expected phenomenon, it informs us that software obfuscation at this point is still not a general interest to mobile development. We may infer that although developers working on security-sensitive business sectors do view malicious reverse engineering as a non-neglectable threat, the obfuscation applied to their works is mostly for protecting *the information encapsulated in the apps rather than the design and implementation of the software.*

Regarding the third point, it turns out that among the 112 Games apps with obfuscation, 87 are passively obfuscated and 82 of them are solely tainted by obfuscated advertising libraries. The statistics fit the general perceptions of the mobile game business model in which publishing third-party advertisements is the major monetization method for free game apps. For Finance and Utilities apps, the fractions of passively obfuscated ones are comparatively lower (46 out of 85 and 18 out of 45, respectively), suggesting that software protection is more seriously considered in these sectors.

4.4.2 Obfuscation Patterns

Before presenting our findings regarding **RQ2**, we first present an overview on the obfuscation patterns extracted from the samples. We studied the pattern of obfuscation in three aspects:

- How many and what kinds of obfuscation techniques are found in the code;
- In what scopes the obfuscation algorithms are applied to the code, i.e., at the function level, class level, or module[3] level;
- Whether multiple obfuscation methods are applied to the same code region to achieve a synergy effect, which we call *synergistic obfuscation*.

We performed this pattern analysis on actively obfuscated apps and obfuscated third-party libraries separately. The results are presented in Table 4.2 and Table 4.3, respectively.

Due to limited space, we only list categories with significant relevance to the discussions in Finding A.2. It may cause confusion that a small number of apps or libraries do not employ symbol renaming even though it is the baseline obfuscation method in sample collection. The reason is that we detect symbol scrambling in obfuscated app instances as a whole. In some cases we "accidentally" detect obfuscated apps or libraries without scrambled symbols because they are "mingled" with obfuscated parts developed by others that indeed contain such symbols. Nevertheless, such cases are rarely seen among actively obfuscated apps (9 out of 195).

Interestingly, all five third-party libraries that did not scramble their symbols are developed by Internet giants like Google, Amazon, Yahoo, Tencent, and Alibaba, suggesting that *large-scale enterprises and smaller mobile development teams may favor quite different obfuscation patterns*, which is worth further investigation.

[3] A module is defined as functionality-related classes coupled through method calls.

Table 4.2: Numbers of Actively Obfuscated Apps Employing Different Obfuscation Patterns

Category	Total	Applied Obfuscation Families				# of Families				Scope of Obfuscation			Synergic Obfuscation
		Symbol	String	Anti-Decomp.	Flattening	1	2	3	4	Function	Class	Module	
Finance	39	39	17	12	0	19	11	9	0	4	10	25	18
Utilities	27	27	10	2	3	15	10	1	1	2	4	21	4
Games	25	22	7	6	0	15	10	0	0	2	3	20	3
Music	12	11	4	1	0	9	2	1	0	1	0	11	3
Medical	10	9	2	0	0	9	1	0	0	2	7	1	0
Others	82	78	18	6	2	66	11	4	1	16	35	31	9
All	195	186	58	27	5	133	45	15	2	27	59	109	37

Table 4.3: Numbers of Third-Party Libraries Employing Different Obfuscation Patterns

Category	Total	Applied Obfuscation Families				# of Families				Scope of Obfuscation			Synergic Obfuscation
		Symbol	String	Anti-Decomp.	Flattening	1	2	3	4	Function	Class	Module	
Advertising	9	7	3	2	0	6	3	0	0	1	1	7	2
Security	7	6	5	1	2	2	3	2	0	0	0	7	3
DRM	6	6	2	1	1	4	1	0	1	1	2	3	1
Payment	5	4	3	1	0	2	3	0	0	1	1	3	1
Others	8	7	3	0	0	6	2	0	0	2	1	5	1
All	35	30	16	5	3	20	12	2	1	5	5	25	8

Finding B.1. *Mobile apps are mostly obfuscated at a large scale, suggesting a wide adoption of automated obfuscation tools.*

In theory, obfuscation can be manually conducted without the aid from automated tools [16]. Nevertheless, we believe this is not the case in mobile development. For actively obfuscated apps, the proportion of those employing module-level obfuscation is 55.90% (109 out of 195). For third-party libraries, the rate is even higher, reaching 71.43% (25 out of 35). Compared to function-level and class-level obfuscation, the workload of protecting one or more modules is significantly heavier, implying that *most mobile developers rely on automated tools for obfuscation.*

On the other hand, it is extremely rare that an entire app or library is obfuscated. Throughout the inspection, we only identified two actively protected apps that are fully covered by symbol scrambling obfuscation. For all other apps and libraries, the obfuscation covers only a small portion of the code. This phenomenon shows that applying obfuscation to mobile software comes with non-negligible cost even if the process can be automated. Presumably, the cost of obfuscation can include but not limited to,

- Increased configuration effort, increased compilation time, and run-time performance penalty,
- Additional cost of software crash forensics due to scrambled symbol names and obscure control flows, and
- Risks of apps being rejected by software publisher for bloated or unanalyzable code (see Finding C.1 for more discussions).

Although it is hard to confirm these items without contacting the developers, we can still get some hints by analyzing other aspects of the obfuscation patterns, as demonstrated by the following finding.

Finding B.2. *The popularity of obfuscation method families decreases as the implementation and performance cost grows.*

It is made clear by Table 4.2 and 4.3 that the popularity of the four obfuscation families vastly differs. The number of apps and libraries containing decompilation disruption and control flow flattening is remarkably smaller than the number of apps and libraries protected with scrambled symbol names and exotic string encoding. Due to our sampling methodology, symbol scrambling is naturally the most popular obfuscation technique across the data set. However, even without symbol scrambling considered, it is still true for the other three families of techniques that, the more costly it is to implement and deploy an obfuscation algorithm, the less widely it is adopted. To elaborate on this trend, we roughly discuss the difficulty of automating obfuscation each method and their impacts on run-time performance, in an increasing order.

Automatically scrambling symbol names is relatively easy and can be implemented through various options like preprocessor macros, compiler instrumentation, and even binary rewriting. Renaming symbols can be implemented in a way that it causes almost no performance degradation during program execution.

Re-encoding string literals in an automated manner requires more effort since it changes program semantics. However, the obfuscation only needs to operate on strings and therefore light-weight program transformations are sufficient. At run time, the obfuscated strings need to be decoded before use, but it is one-time cost and only manifests when programs launch.

Compared with the first two families of obfuscation, decompilation disruption is significantly more difficult to implement, for obfuscator writers need reverse engineering experience to understand how to disrupt a decompiler. It is hard to analyze the run-time cost of this obfuscation since techniques in this family can vary a lot. Nevertheless, the performance penalty is not constant and will keep accumulating as programs run.

Implementing control flow flattening requires deep customization of the compiler which falls out of the skill sets of most common mobile developers. Same as decompilation disruption, each execution of flattened control flows takes an additional amount of time. It is also worth noting that control flow flattening can increase the size of obfuscated binaries significantly.

Currently, we are unable to confirm whether the difference of popularity results from exact one of the two factors, i.e., implementation cost and performance penalty, or both of them. Theoretically, if the obfuscation is conducted with third-party tools, the technical challenges in implementing each obfuscation method should not be a problem, leaving performance to be the primary concern. Otherwise, if the intention of apply software protection is really blocked by technical issues, there will be many opportunities for obfuscation toolkit providers to improve their products and attract more mobile developers to embed advanced obfuscation techniques into their apps and libraries. It would be interesting future work to investigate which is the case.

Finding B.3. *Apps and libraries of certain categories tend to adopt more complicated obfuscation patterns than others.*

Finding A.2 shows that apps serving life-, money-, and privacy-critical purposes are more likely to be obfuscated. It is further suggested by Table 4.2 and Table 4.3 that the security strength of obfuscation applied to apps and libraries of these kinds is also notably stronger. In general, the Finance, Utilities, Games, and Music apps, if obfuscated, are more willing to employ expensive obfuscation techniques, i.e., decompilation disruption and control flow flattening. These apps also tend to employ more different families of obfuscation techniques. For example, over half (20 out of 39) of the actively obfuscated Finance apps contain plural kinds of obfuscation. Moreover, in many cases (18 out of 20), these different methods were applied to the same part of the code, achieving synergistic obfuscation. Also, the scope of obfuscation in these apps is often larger, mostly reaching module-level protection.

The observation above applies to obfuscated third-party libraries as well. Overall, the obfuscation patterns found in libraries are very similar to those in actively obfuscated apps in most aspects. Therefore, *it can be difficult to distinguish actively and passively obfuscated mobile apps by simply analyzing their obfuscation patterns.*

Finding B.4. *An increasing number of mobile apps start to integrate obfuscation into the development process.*

As aforementioned, our sample crawling was continuous and lasted for nine months. For apps getting version updates during the crawling period, we were able to analyze the temporal evolution of their obfuscation patterns. With these historical versions and some additional examinations, we confirmed that 27 of the 195 actively obfuscated apps were unobfuscated at the beginning of the crawling period. It is very likely that developers of these apps were newly attracted by the benefits of software protection and started to employ it as part of their software engineering routines. Note that 27 is a possibly untight lower bound because the recorded version histories may be incomplete because of the limited workload capacity of our crawler.

Unfortunately, the same analysis does not apply to passively obfuscated apps, since they may include different third-party libraries in different versions. The change of obfuscation status in these apps may not reflect the intention of their developers. The analysis is also not applicable to third-party libraries, because we were unable to obtain the development dates of each version of the same library.

4.4.3 Impact of Distributor Code Review

Centralized software distribution usually features a vetting process in which an app must be reviewed by the distributor before allowed for publication. Through this vetting process, software publishers aim to filter out malicious or misbehaving applications that can hurt user experience or security, thus affecting the healthiness of the ecosystem. Both iOS and Android employ this centralized model.

Hypothetically, this mandatory app review process can affect developer incentive to obfuscate their products in two opposite ways. Firstly, although software obfuscation is a legit approach to protecting apps from undesired reverse engineering, it hinders distributor reviews as well. If the reviewer acts conservatively and considers unanalyzable code malicious, the obfuscated apps may be constantly rejected, making developers reluctant to adopt heavy-weight obfuscation algorithms. On the other hand, some developers may be stimulated to obfuscate their code so that they are able to circumvent certain checks, allowing their apps to possess features forbidden by publisher policies. We have encountered two cases supporting both possibilities, respectively. Although not qualified as solid evidence to validate our hypotheses, these case studies can indeed provide valuable insight on the problem.

Finding C.1. *Code reviews enforced by mobile software publishers may influence the adoption of obfuscation in different directions.*

The first case is a heavily obfuscated app developed by a reputed commercial iOS security service provider, which only published that single app in App Store. Judged from the simplicity of its functionality, this app is merely a minimal working example of iOS development, whereas it is protected by all four kinds of obfuscation techniques considered by our study. Only two among the 195 actively obfuscated apps are obfuscated in this pattern. We speculate that the security solution provider submitted this app to address the concerns that their obfuscation algorithms may cause distributor review alarms, to the detriment of the sales of their services. It is known that App Store have various constraints on submitted apps, some of which may not be clearly documented. For example, each slice of an executable file in iOS apps must not exceed 60 MB [18] if the app is to be compatible with older versions of iOS, limiting the use of code transformations that bloat binary size too much. These constraints intrigue obfuscator writers to test the boundaries of acceptable obfuscation techniques. This case suggests that *developing new mobile obfuscation algorithms has to take the app vetting process into account to be practical.*

In the second case, we found that a third-party advertising library contains code for calling private iOS APIs, which is strictly forbidden by Apple App Store security policies. To circumvent store reviews, the library writer uses the `dlopen` system call to avoid direct linkage to internal iOS frameworks providing private APIs. The library then uses exotic string encoding to obfuscate the string literals provided to `dlopen` as parameters. In this way, Apple's vetting analysis failed to detect this violation. By searching related information on the Internet, we learned that this library was once caught using private iOS APIs in 2015 [17], long before we started crawling samples from App Store. Shortly after the incident was reported, Apple announced that it had removed all apps contaminated by this library from App Store. Yet our findings show that either authors of the library managed to bypass the app review process for another time or Apple failed to detect all apps including this library. Whichever is the case, this finding serves as empirical evidence that *obfuscation is not only employed to repel malicious reverse engineering but also for infiltrating publisher inspection*, even though this practice is previously regarded as a signature of malware.

By nature, ad providers are impelled to collect as much client data as possible for developing more effective ad distributing strategies, potentially placing themselves on the verge of infringing user privacy. Considering the large quantity of obfuscated third-party advertising libraries and their wide spread in the sample set, we are concerned by the possibility that abusing obfuscation to bypass publisher security policy enforcement is becoming a common practice for aggressive adware on the mobile. Mobile apps falling within a "gray area" that are controversially benign or malicious, aka. "grayware," has drawn attention from the security community [29].

4.4.4 Effectiveness of Obfuscation

We now present our findings regarding the effectiveness of real-world obfuscation for mobile apps. It should be emphasized that our goal is not to access the security strength of obfuscation techniques themselves like previous literature review did [122] but to investigate whether iOS developers are able to appropriately utilize these techniques and optimize the protection effects.

With limited labor, we cannot afford to conduct comprehensive penetration tests for all apps in our sample set. Even though, we found that a modest amount of reverse engineering effort is enough to reveal some information that possibly leads to security breaches. We inspected the actively obfuscated apps in two aspects. Firstly, we scanned all symbol names, searching for common key phrases related to security, such as "private key" and "secret". Secondly, during the detection of exotic string encoding, we payed attention to string literals that are not obfuscated and seem to leak sensitive information.

Finding D.1. *A considerable portion of obfuscated apps remain vulnerable to low-effort reverse engineering, which could have been avoided if the obfuscation was performed more appropriately.*

With preliminary reverse engineering effort, we found that among the 195 actively obfuscated apps, there are 33 that may leave certain sensitive information unprotected due to lack of certain obfuscation techniques or insufficient coverage by the right techniques. There are mainly three kinds of such information:

- Tokens assigned to apps for accessing third-party services. Some enterprise entities provide APIs for mobile apps to retrieve proprietary information or upload app usage data for analytics, usually at a price. Requests for accessing these services has to be sent with tokens issued by service providers to prove the identities of requesting clients. We found that some apps store these tokens as plaintext in variables whose names are not scrambled.

- In-app secrets. Apps may encrypt their private data such as execution logs and intermediate results before storing them on mobile devices. Some poorly obfuscated apps store encryption keys in plaintext as string literals.

- Information about back-end servers connected with the apps and the corresponding communication protocols. *In particular, we found* 4 *apps, which are the mobile clients of some financial institutions, leaking the URLs or IP addresses of their back-end testing infrastructures.* Surprisingly, accessing these infrastructures does not require any authentication. The communication protocols and even internal documentations are exposed to anyone knowing the URLs or IPs.

It is true that information leaked above does not necessarily lead to exploitable security vulnerabilities. Per common software security principles, however, such information should not be exposed to unauthorized parties in the first place. Although leakages discovered by our study were caused by series of inappropriate software engineering practices, the problem will be less severe if the apps are more properly obfuscated. In our opinion, the current status of software protection on mobile platforms is far from satisfactory. Both the academia and the industry should invest some effort in improving the developer practices of utilizing obfuscation techniques.

4.5 Implications of the Results

Through this empirical study, we learned that third-party libraries play a significant role in iOS app obfuscation, which is consistent with the situation on

Android [91]. Being a major source of obfuscated code, third-party libraries affect software attributes in various aspects without app developers being aware. We urge that future studies on iOS app repositories to take obfuscated third-party libraries into consideration and develop dedicated analysis techniques to handle them.

We have found a posteriori evidence indicating the correlation between the likelihood of mobile apps being obfuscated and their functionality. Particularly, apps related to finance, privacy, intellectual properties, and monetization are more likely to be obfuscated. It may be worthwhile for obfuscation service providers to take an in-depth study on the characteristics of these apps and specialize their products to better fit the demands of their vendors.

Our study suggests that the adoption of obfuscation on mobile platforms may be affected by mandatory code reviews from app distributors. Since obfuscation is inherently unfriendly to code reviews and may causes disapproval from the reviewer, app developers will likely face the dilemma between improved security and shorter time to market of their products. This factor needs to be considered when developing or advocating new obfuscation techniques for mobile platforms, particularly iOS whose vetting process is much more strict that Android.

We noticed an increasing trend in the number of mobile apps getting obfuscated. For a notable portion of these apps, however, the obfuscation was not appropriately conducted, leaving them still vulnerable to certain low-effort reverse engineering techniques. As such, we believe that future efforts on software protection should not only focus on developing new obfuscation techniques but also proposing accessible policies and strategies that can guide mobile developers to maximize the efficacy of existing techniques.

In conclusion, we empirically investigated the status of software obfuscation in the mobile software industry. We collected a large set of obfuscated iOS applications in the real world and performed in-depth analysis on these samples. With the information gathered in the study, we revealed factors potentially affecting the deployment of obfuscation techniques in mobile apps and typical obfuscation patterns adopted by mobile developers. We believe that findings developed in this research will shed light on future research that aims to understand and improve the state of art of software protection.

Chapter 5

A Case Study on Real-World Mobile Obfuscation

In Chapter 1.1, we discussed why iOS software is particularly vulnerable to reverse engineering and the threats faced by enterprise iOS developers. In this chapter, we apply our knowledge about advanced obfuscation techniques to protecting a group of commercial iOS apps which serve millions of users. These apps span a wide range of functionality categories, including news, utility, navigation, payment, social networking, and shopping. To be specific, our objective is to protect a common code base shared by these apps. The protected iOS code base consists of 23K lines of Objective-C and C code, which roughly takes 0.5% to 2% of each including app.

Practicability has been a major concern of research on new obfuscation techniques. This dissertation shares the field experience of operationalizing obfuscation as part of the real-world software engineering procedures. By analyzing the benefits, pitfalls, and costs of obfuscation in massive production settings, we deliver a deeper understanding on the role of obfuscation in secure software engineering, particularly in the development of large-scale mobile software.

The work of this chapter is published in *Proceedings of the 40th International Conference on Software Engineering, the Software Engineering In Practice Track*, 2018 [137].

5.1 Tools

iOS apps can be developed in several different programming languages, including C, C++, Objective-C, and Swift. Apple provides different frontends for each language, while all backends are based on the LLVM compiler infrastructure.[1] Therefore, all source code in an iOS project is eventually translated into the LLVM intermediate representation (IR). Most of the compiler assets for iOS development have been made open source. This allows other software vendors to develop new features for the compilers.

Considering the iOS app build process, we decided to implement our obfuscation tool as a series of LLVM IR transformation passes. Compared with other options like source-level and binary-level obfuscation, the IR-level solution provides multiple appealing benefits:

- IR obfuscation is language independent. A single IR transformation module can process most part of an iOS app, which is not the case for source-level obfuscation.

- Apple now advises app developers to submit their products in the form of LLVM IR rather than binary. IR-level obfuscation fits this practice better than binary-level obfuscation.

- A compiler-based obfuscator is mostly transparent to app developers, minimizing the interference to the normal development process.

The current implementation of our obfuscator consists of about 3.8K lines of C++ code,[2] plus another 1K lines of third-party code for random number generation and security hashes. The obfuscator provides different obfuscation algorithms

[1]The swift compiler backend is based on a separately maintained LLVM version, thus slightly different from the standard one.

[2]Code statistics in this chapter include comments and blanks.

that can be arbitrarily combined per developer demands. The granularity of obfuscation is configurable through customized compiler flags and extended function attributes. Figure 5.1 shows how app developers can control the granularity of obfuscation at the compilation unit level and function level. In actual development, each obfuscation algorithm can be configured separately.

5.2 Obfuscation Algorithms

Choosing the appropriate obfuscation algorithms is the first step to effective protection of iOS apps. In addition to effectiveness, obfuscation in real-world software engineering also needs to take many other factors into account. On iOS, there are several issues that may not exist on other platforms. We discuss these factors with more details below.

Platform-wide security policies iOS is considered to be one of the most secure mobile systems, for it enforces extremely restrictive security policies on its apps. The policy affecting obfuscation the most is called *code signing*. To counter software tampering, iOS ensures that every executable page owned by a third-party app must be signed and checked for integrity before code in that page is executed for the first time after the process starts. On the other hand, changing the execution permission of a memory page is not allowed for third-party apps. This means self-modifying code is strictly prohibited on iOS, leaving dynamic code rewriting obfuscation unfeasible. For this reason, many packer-based obfuscation techniques that are popular on Android [153] are not viable options for iOS.

Binary size For apps that need to support all living iOS versions (including 7 and above), Apple imposes a 60 MB limit on the size of the code section in each executable [20]. Since many popular apps have large code bases, this limit is very tight. Even if the code to be obfuscated is only a small part of the apps, developers

```c
// source.c, compiled with -obf flag

void foo() {
  ...
}

void bar() {
  ...
}
```

(a) Obfuscate the whole compilation unit

```c
// source.c, compiled with -obf flag

__attribute__((no_obf)) void foo() {
  ...
}

void bar() {
  ...
}
```

(b) Obfuscate the whole compilation unit excluding `foo`

```c
// source.c, compiled without -obf flag

__attribute__((obf)) void foo() {
  ...
}

void bar() {
  ...
}
```

(c) Obfuscate only `foo` in the compilation unit

Figure 5.1: Obfuscation configuration examples

cannot afford obfuscation algorithms that bloat the software size too much. That includes virtualization-based obfuscation [126, 54], which requires integrating a full-fledged hardware emulator into the app.

LLVM IR compatibility Since our obfuscator operates on LLVM IR, it can be challenging, if possible at all, to implement certain obfuscation algorithms that require extensive manipulations of low-level machine instructions.

App review All iOS apps are reviewed by Apple App Store before allowed to be published. This is a necessary procedure for minimizing the number of low-quality and malicious apps delivered to users. While the details of app reviews are kept confidential, it is likely that both humans and automated analyzers are participating in the process. It is imperative that our obfuscation does not have adverse impact on the review. In particular, we must make sure that the applied obfuscation algorithms strictly abide by the iOS developer regulations [18].[3]

Considering the factors listed above, we made a careful selection of obfuscation algorithms, listed as follows.

1. *Symbol name mangling* that turns understandable human-written identifiers into strings that do not indicate program semantics.[4]

2. *String literal encryption* that hides the plaintext of the string literals stored in the binary. The protected strings are decrypted at run time.

3. *Disassembly disruption* that confuses instruction decoding and function recognition in binary analysis. Typical methods of disruption include interleaving

[3] It is known that some iOS developers have tried to misuse obfuscation to disrupt and mislead the review process such that the apps can secretly possess features disallowed by Apple. We emphasize that techniques discussed in this chapter are not meant to advocate such behavior, nor any app obfuscated by us ever seeks to bypass Apple's review through obfuscation.

[4] Although symbol name mangling was valid obfuscation on iOS by the time of writing, our latest communication with Apple suggests that it may not be acceptable any more. Readers interested in adopting this method should carefully consult with Apple about their possibly undocumented regulations.

data with code and forging code patterns that code analyzers recognize as special hints for disassembly.

4. *Bogus control flow insertion* that constructs unfeasible code paths guarded by opaque predicates [52].

5. *Control flow flattening* that obscures the logic relations between program basic blocks [49].

6. *Garbage instruction insertion* that injects garbage code that is irrelevant to program functionality [50].

Among these obfuscations, symbol name mangling and string literal encryption are mainly for misleading human perception while the others are meant to confuse both humans and automated tools. The major focus of our solution is to impede automated binary disassembly and decompilation, which are the early steps of most malicious activities conducted by the practitioners of underground economy targeting iOS apps.

We ensure that all selected obfuscation algorithms well abide by Apple's security policies. By analyzing other obfuscated iOS apps found in the App Store, we have confirmed that these algorithms or their variants have been previously employed by legit app developers, indicating that they are unlikely to affect the review process. Regarding the limit for binary size, obfuscation (1), (2), and (3) barely introduces spatial overhead into the obfuscated binaries. For the other three algorithms, the expanded binary size can be controlled within an acceptable rate by carefully tuning the configurable obfuscation parameters, e.g., the ratio of inserted opaque predicates and garbage instructions to the amount of the original code.

Through our implementation, we have confirmed that all selected algorithms are fully compatible with LLVM IR, except for (3), which needs to directly manipulate machine code. We partially addressed this problem with the use of *inline assembly*,

```
; @foo: A function computing foo(a, b) = a + b
define i32 @foo(i32 %a, i32 %b) #0 {
entry:
  ; %x: uninitialized 32-bit integer variable
  %x = alloca i32, align 4
  %0 = load i32, i32* %x, align 4
  %1 = load i32, i32* %x, align 4
  %add = add nsw i32 %1, 1
  %mul = mul nsw i32 %0, %add
  %rem = srem i32 %mul, 2
  ; %tobool: opaque predicate 'x*(x+1)%2 != 0' (constantly false)
  %tobool = icmp ne i32 %rem, 0
  br i1 %tobool, label %if.then, label %if.else

; %if.then: unreachable block guarded by %tobool
if.then:
  ; insert 4-byte data 0xdeadbeaf with inline asm
  call void asm sideeffect ".long␣0xdeadbeaf", ""()
  br label %if.end

if.else:
  %add1 = add nsw i32 %a, %b
  br label %if.end

if.end:
  %2 = phi i32 [%x, %if.then], [%add1, %if.else]
  ret i32 %4
}
```

Figure 5.2: Example of obfuscation utilizing LLVM IR inline assembly

a feature supported by many implementations of C-family languages and LLVM itself. Figure 5.2 shows an example of interleaving data and code at the LLVM IR level. The inserted data are used for disrupting disassembly. The data chunks are guarded by an opaque predicate so that they are never reached and thus do not compromise normal execution. In Chapter 5.3, we will discuss implementing binary obfuscation at the IR level in more depth.

Many obfuscation methods we employed have reference implementations from the open source community [13, 12, 82]. We intentionally made our implementation different from the public ones by altering code patterns and introducing new features. Attackers will need more sophisticated techniques to nullify the mutated obfuscation effects [146]. Indeed, most of the mutations we made are supplementary and it is questionable whether they render the obfuscations fundamentally more difficult to defeat. Ideally, a reliable defensive measure should be secure even if its technical details are known to attackers. This is however a standard not met by most obfuscation techniques used in practice. As a consequence, keeping the obfuscation details confidential is one of the few advantages that benign developers can hold over adversaries. Regardless, the customized obfuscation techniques can at least make reverse engineering much more tedious and frustrating, since reverse engineers will have to undo the customization before reducing the mutated obfuscation to its baseline form. Again, we would like to note that the main contribution of this part of the dissertation is not developing or evaluating new obfuscation methods, but maximizing the value of existing techniques in practical engineering.

5.3 Implementation Pitfalls

We have encountered a series of technical issues when trying to implement the aforementioned algorithms, many of which are quite stealthy and lead to subtle problems affecting the potency and practicality of our work. Some of the issues are generally relevant to software obfuscation, but more of them are unique to iOS.

5.3.1 Inline Assembly

As previously mentioned, the inline assembly feature of LLVM allows IR transformations to manipulate machine instructions. To the best of our knowledge, this is the only solution that makes binary-level obfuscation possible if we are to follow the currently recommended iOS app development procedure.

Since directly manipulating or adjusting machine instructions after compiling the source code is not possible, the capability of our solution is significantly limited. In principle, inline assembly can only perform instruction insertion but not code modification or deletion. Moreover, at the time of IR transformation, most machine code is not yet generated by the LLVM backend, making it extremely difficult to construct complicated binary transformations solely with LLVM IR manipulation. Another factor to consider is the characteristics of the ARM architecture. Compared with the CISC architectures x86 and x64 where binary-level obfuscation is quite prevalent, ARM is RISC and employs the fixed-length instruction encoding. This invalidates many obfuscation techniques that exploit the variable-length encoding of instructions, such as overlapping instructions [38].

According to our experience, the following obfuscation-oriented transformations can be correctly implemented with LLVM inline assembly:

- Insert junk instructions.
- Interleave data and code in unreachable basic blocks.
- Perform control flow transfers that are consistent with the IR-level control flows.
- Diversify stack frame layouts by manipulating the stack and frame pointer registers.

It should be noted that the correctness of these transformations cannot be guaranteed for concurrent code, due to the lack of support for volatile inline assembly in LLVM. In certain cases, aggressive compiler optimization may also make binary-level obfuscation problematic. As such, it is extremely crucial to thoroughly test the obfuscator in real app development and production settings. Because of this potential instability of binary-level obfuscation in LLVM IR, developers should take deliberation to make appropriate trade-offs among security, reliability, and maintainability when designing an iOS obfuscator.

5.3.2 Heterogeneous Hardware

In contrast to Android, iOS runs on a very limited set of models of hardware, therefore hardware fragmentation is much less of an issue for most iOS developers. For obfuscation, however, heterogeneous architectures is still a factor that needs to be considered, especially when obfuscation aims to hinder binary disassembly, which is heavily architecture dependent.

iOS and its variants support both 32-bit and 64-bit ARM architectures. For iPhone apps, 32-bit binaries are no longer supported since iOS 11, while other Apple mobile devices like smart watches and smart TVs will keep supporting 32-bit binaries for a much longer time. If a developer intends to release its apps on all active iOS devices and the code fragments to be protected are shared by apps on different platforms, obfuscation should guarantee that code for the two architectures are equally protected. If code for one architecture is less well obfuscated than that for the other, attackers will simply choose to breach the weaker spot, leaving the more effective protection on the other architecture meaningless.

5.3.3 App Maintainability

In most cases, when commodity software crashes, the only information available to software developers for investigating the root causes are the core dumps and stack traces collected at crash sites. This applies to iOS apps as well. Developers can either embed a third-party crash reporting library into their apps or periodically receive diagnosis reports from Apple. In either case, the readability of the stack traces will be affected by obfuscation, potentially making app maintenance troublesome.

Obfuscation can render crash traces unreadable in two aspects. Firstly, the symbol names appearing in the stack traces, especially the function names, are scrambled into strings meaningless to humans. To undo this effect at the time of crash analysis, the obfuscator need to memorize the mapping from original symbol names to the mangled ones during app compilation and revert the obfuscated names before app maintainers read crash reports. Secondly, obfuscation inserts additional code into the software, which cannot be correlated to any location in the source files. Ideally, if the obfuscator is correctly implemented, obfuscation-specific code should not cause crashes. However, since iOS apps are mostly written in unsafe programming languages that are prone to memory errors, faults caused by defective genuine app code may propagate to surrounding locations, possibly reaching code introduced by the obfuscator. To tackle this problem, we make the obfuscator generate extra debug information for the inserted code. In order to minimize the confusion caused to crash analysts, we adopt a "nearby principle" that maps obfuscator-generated code to the source location of the nearest genuine app code within the same lexical scope.

On iOS, the debug information of an executable is collected into a dedicated metadata file and is only accessible to app developers. Therefore, enriching debug information will not accidentally help reverse engineers better understand the app.

5.4 Evaluation

We now report the outcome of our obfuscation effort. The protected iOS code base consists of 23K lines of Objective-C and C code, which roughly takes 0.5% to 2% of each including app. We evaluated the obfuscation in two aspects, i.e., resilience and overhead. According to the definition by Collberg et al. [52], resilience indicates how well the obfuscation can withstand *automated* reverse engineering. As for overhead measurement, we focus on binary size expansion and execution slowdown.

5.4.1 Resilience

Although software obfuscation has been actively researched for quite some time, how to systematically assess the security strength of an obfuscator remains an open

Table 5.1: Performance of IDA Pro Function Recognition

Target Architecture	Number of Functions					
	Original			Obfuscated		
	Ground Truth	IDA Reported	False Positives	Ground Truth	IDA Reported	False Positives
ARMv7 (32 bit)	993	1152	159	1069	3516	2447
AArch64 (64 bit)	991	1121	130	1065	1778	713

problem. The theoretically solid evaluation methodology is to reduce deobfuscation to a computational problem with provable or conjectural intractability. To date, this has only been done for indistinguishability obfuscation [70], which is still not practical for protecting real-world software [109]. On the other hand, evaluation through empirical experiments always raise concerns about the possibility that the obfuscation can be effectively nullified by some unknown or future deobfuscation methods not considered by the evaluation. Some recent effort has tried to establish standards for assessing the security strength of obfuscation techniques [34, 136, 115], but it remains unclear how well they can fit the demands of practical software protection.

Practitioners in industry mostly evaluate the resilience of an obfuscation technique through white-hat penetration tests. Although the procedures of these tests are exceedingly subjected to human intuition and experience [44, 34], the early steps are fairly standard. Typically, the testers will first use automated reverse engineering tools to reduce binary code into a form that is much more convenient for humans to inspect. Our internal penetration test also follows this scheme. In the section, we report the effectiveness of our obfuscator by showing its resilience to IDA Pro, the de facto industrial standard of binary disassembler and decompiler.

Our obfuscation delivers two major disrupting effects on the efficacy of IDA Pro. The first one is that IDA Pro will report significantly more false positives when trying to recognize the starting addresses of functions in an obfuscated binary, due to the confusing code patterns we inserted. Table 5.1 displays the true numbers of functions, the numbers of functions recognized by IDA Pro, and the numbers of false positives, counted before and after obfuscation. Note that the ground truths of function numbers before and after obfuscation are slightly different because the obfuscator inserted some helper functions during IR transformations.

The other disrupting effect is that IDA Pro will fail to disassemble a large portion of the binary code, due to the garbage instructions, intrusive binary data, and unfeasible control flows forged by the obfuscator. Figure 5.3 presents the performance of IDA Pro regarding the original and obfuscated binaries in terms of the proportion of successfully disassembled code. Before obfuscation, IDA Pro is able to disassemble almost all binary code for both 32-bit and 64-bit architectures. After obfuscation, the disassembler can only process 51.1% of the 32-bit binary and 14.1% of the 64-bit binary.

As discussed in Chapter 5.3, it is crucial for an iOS obfuscator to protect binaries of different architectures equally well. When interpreting the results in Table 5.1 and Figure 5.3, it is important to note that the two metrics used in the evaluation are complementary. Since a recognized function must have a body, more falsely recognized functions naturally lead to more disassembled binary chunks. Considering that IDA Pro reports much more false positives in 32-bit binary function recognition, a disassembly rate higher than the result for the 64-bit version is plausible. In other words, despite that IDA Pro can disassemble more code in the 32-bit binary, the additionally decoded instructions are incorrectly promoted to functions that do not exist in the source code, which actually has a negative effect on further analysis. According to our internal penetration tests, although the two versions of obfuscated binaries confused IDA Pro in different ways, the end effects are about the same.

5.4.2 Overhead

To measure the obfuscation overhead, we implemented the evaluated code base as a standalone iOS app by adding necessary initialization procedures and a minimal GUI. The newly added code is negligible for the purpose of measurement. We report both the spatial and temporal overhead caused by obfuscation. As discussed at the beginning of this chapter, the protected part of the code is small compared to apps including it. Since the routines provided by this part are mostly decoupled from the main functionality of the apps and typically run in the background, the

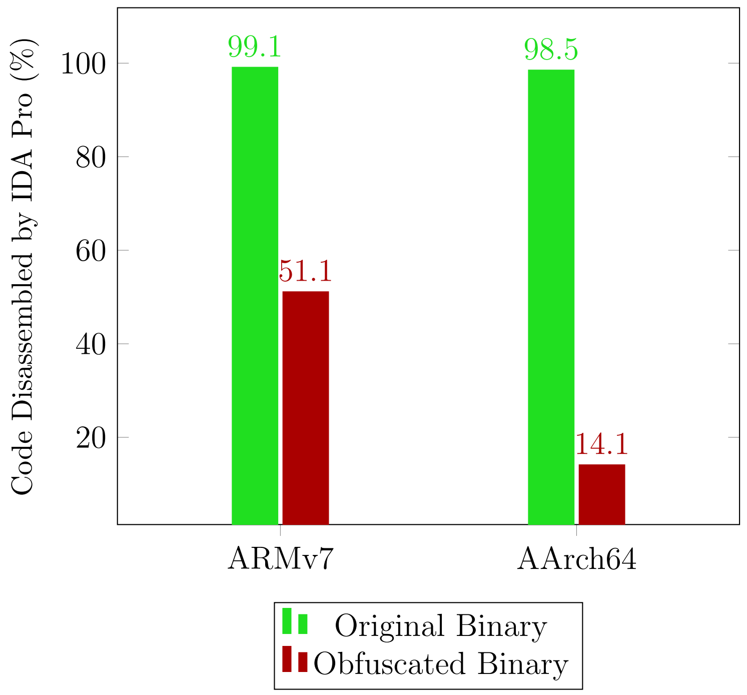


Figure 5.3: Effectiveness of disassembly disruption

impact of obfuscation on the overall execution speed is expected to be modest. In contrast, the bloated binary size is more of a concern due to the strict size limit on the code segments of iOS apps.

5.4.2.1 Size Expansion

For most of our obfuscated apps, the 64-bit binaries suffer more from the limited quota of binary size, because the 32-bit iOS binaries are usually smaller than their 64-bit counterparts. The main reason is that 32-bit binaries are composed of THUMB2 instructions whose encoding is more compact than that of 64-bit instructions. Meanwhile, the size limits for the two architectures are the same, meaning the obfuscated part by itself is allowed to consume more quota on the 32-bit platform.

Table 5.2 shows the code segment sizes of the original and obfuscated iOS apps. As can be seen, the obfuscation can cause 3 to 4 times of binary inflation, suggesting that whole-app obfuscation is likely inapplicable to large-sized iOS apps.

Another observation is that the obfuscation bloats the 64-bit binary less than the 32-bit version, in terms of proportion. As mentioned above, this is a somewhat desirable outcome since the size problem is more troublesome for 64-bit binaries. We conducted a preliminary investigation to explore the causes of this phenomenon. We found one of the reasons is that the 32-bit and 64-bit ARM backends of LLVM handle relocatable memory addresses differently. Since ARM is RISC and has a limited instruction length, loading a large constant integer into a register usually takes more than one instruction to accomplish. According to our observation, the 32-bit ARM backend of LLVM materializes relocatable memory addresses by employing constant pools, while the 64-bit backend uses dedicated instructions like `adrp`, which are slightly more efficient than the 32-bit solution in terms of the total bytes of instructions generated. Since our obfuscator emits a lot of large constants to represent basic block addresses, the difference between the size efficiency of the two backends is significantly amplified.

Table 5.2: Binary Size Expansion Due to Obfuscation

Target Architecture	Code Segment Size in Bytes		
	Original	Obfuscated	Increase
ARMv7 (32 bit)	286304	1070656	784352 (+307%)
AArch64 (64 bit)	333376	1165456	832080 (+221%)

5.4.2.2 Execution Slowdown

We tested the decrease in execution speed after obfuscation on an Apple iPad Air, an iOS device released in 2013, which has a 1.4 GHz dual-core ARM CPU and 1GB RAM. The obfuscated code performs both synchronous and asynchronous tasks inside host apps. The asynchronous tasks are scheduled sparsely during app execution and we did not detect any notable slowdown after obfuscation was applied. As for the synchronous part, the execution penalty is from 5% to 10% for both 32-bit and 64-bit builds,[5] while the app-wide slowdown is mostly negligible. This result indicates that performance degradation is not necessarily the primary blocker that prevents obfuscation to be applied to real-world mobile apps.

5.5 Discussion

5.5.1 Dilemma of Security and Transparency

In our experience, one of the most challenging factor that prevents thorough software protection on iOS, and potentially on all platforms featuring centralized software distribution, is the conflict between seeking more securely obfuscated code and retaining the transparency to app reviews. Naturally, the more effectively an app is obfuscated, the more difficult it makes the distributor to review the functionality of the code, even though the purpose of obfuscation is to prevent reverse engineering only from the malicious parties. Since Apple does not provide official support for iOS app protection, the developers will have to carefully take the balance themselves.

An adequate solution to the dilemma is to let the app distributor perform obfuscation after the review is completed and before the app is published. Indeed, this solution will shift the burden of protection from iOS developers to App Store, which may not be practical in the near future. However, we believe that it could significantly benefit the entire iOS ecosystem in long terms.

Although it is unclear whether post-review obfuscation can be expected by iOS developers at this stage, there are indeed other more realistic measures that iOS can take to improve app code security. For example, some library developers would like their products to be freely downloaded by any developers who are interested, yet

[5] The precise measurement results are confidential per app developer requirements.

they also wish to keep the actual content of the code confidential from potentially malicious clients and competitors. Since iOS app code generation can now be conducted remotely on Apple's cloud, it is technically feasible for iOS to provide encryption facilities for third-party library code such that only the programming interface can be seen by other developers while the actual library content is only revealed to Apple. Although this cannot prevent the code from being analyzed after apps containing the libraries are released, it is still a step forward towards more effective iOS software protection.

5.5.2 Other Protections

Obfuscation is not a panacea for combating the security threats targeting mobile apps and there have been many deobfuscation techniques proposed [150, 147, 54, 105]. A comprehensive defense requires a synergy among various countermeasures. At this point, obfuscation techniques available on iOS are mostly designed for hindering static analysis, while reverse engineering can also be conducted dynamically. Given a jailbroken iOS device, reverse engineers can tamper with an app by injecting third-party code into its process. In this way, adversaries can debug the app at run time to circumvent certain static protections provided by obfuscation. Reverse engineering tools like cycript [8] and Frida [11] have made it quite convenient to perform on-device debugging for arbitrary iOS apps. There are at least two effective dynamic tampering attacks:

- *Sensitive information pry.* Depending on the objective of an attack, it is sometimes sufficient for attackers to place hooks at critical program points of an app and dynamically monitor what types of data are being exchanged. Such information leakage is extremely severe for data-driven defenses like anomaly detection.

- *Replay attacks.* On jailbroken devices, attackers is capable of dynamically invoking arbitrary Objective-C methods of an app after injecting the debugging module at run time, which allows them to replay certain communications between apps and servers. It is known that attackers have used replay to counterfeit users clicks so that they can trick ad providers into paying them for nothing [56].

Various techniques are available for preventing software from being dynamically debugged by unauthorized parties. However, anti-debugging faces a problem similar to obfuscation regarding its security guarantee. In the case of iOS, since attackers are able to gain full control over the app and the system altogether, code integrity can be easily breached. In theory, attackers can rewrite app binaries and remove all anti-debugging facilities before dynamically inspecting them.

Although neither obfuscation nor anti-debugging is comprehensive by themselves, there is a chance that they can be combined to patch the weaknesses of each other. To disable anti-debugging, attackers will have to gain certain knowledge about the defenses in static means. On the other hand, before removing the anti-debugging facilities, attackers cannot circumvent obfuscation via dynamic analysis. Therefore, when obfuscation and anti-debugging are deployed together, they can form an all round defense against reverse engineering.

Chapter 6

Conclusion

In this dissertation, we presented translingual obfuscation, a novel software obfuscation scheme based on programming language translation. By utilizing certain unique features of the target language, we are able to protect the original program against reverse engineering. We implemented Babel, a tool that translates part of a C program into Prolog and makes use of Prolog's highly abstract computation model and its implementation to turn program into a much more obscure form. We evaluated Babel with respect to potency, resilience, cost, and stealth on real-world C programs of different categories. The experiment results show that translingual obfuscation is an adequate and practical software protection technique in desktop environments.

We also empirically investigated the status of software obfuscation in the mobile software industry. We collected a large set of obfuscated iOS applications in the real world and performed in-depth analysis on these samples. With information gathered from the study, we revealed the factors that potentially affect the deployment of obfuscation techniques in mobile app development and typical obfuscation patterns adopted by mobile developers. We believe that these findings can shed light on future research that aims to understand and improve the state of art of software protection.

Finally, we shared our experience with applying software obfuscation to iOS mobile apps in realistic software development settings. In particular, we discussed what efforts are required to make obfuscation a practical technique when applied to complicated apps with large user bases. We summarized the major pitfalls that may prevent iOS developers from utilizing obfuscation effectively and efficiently. We then presented how we addressed these challenges when designing and implementing an industry-quality iOS obfuscator, followed by quantitative evaluations about the resilience and cost of the obfuscator. The evaluation was conducted on a code base included by multiple commercial iOS apps, each of which serves millions of users. The results show that software obfuscation, being an technique accessible to common mobile developers, can indeed provide reasonably effective protection against malicious reverse engineering with affordable cost.

Appendix A

Additional Potency Evaluation Data For Babel

In this appendix, we present the program complexity at obfuscation levels of 10%, 20%, 30%, 40%, and 50%, for both Babel (Table A.1) and Code Virtualizer (Table A.2). We would like to remind readers that for the Babel potency data, all values are obtained by IDA Pro. Since Babel generates many indirect control flow (see Chapter 3.3.2 and Figure 3.4), it is hard to evaluate how accurate IDA Pro is when analyzing indirect control flows in our case, the reported values can be interpreted as lower bounds of the corresponding metrics.

Table A.1: Program Complexity of Babel-Obfuscated Binaries at Different Obfuscation Levels

Program	Obfuscation Level	# of Call Graph Edges		# of CFG Edges		# of Basic Blocks		Cyclomatic Number		Knot Count	
		Value	Ratio	Value	Ratio	Value	Ratio	Value	Ratio	Value	Ratio
bzip2	No Obf.	353	1.0	5382	1.0	3528	1.0	1856	1.0	3120	1.0
	10%	5609	15.9	18539	3.4	15445	4.4	3096	1.7	12488	4.0
	20%	5719	16.2	18909	3.5	15788	4.5	3123	1.7	12166	3.9
	30%	5964	16.9	19771	3.7	17078	4.8	2695	1.5	12396	4.0
	40%	6386	18.1	19630	3.6	17907	5.1	1725	0.9	12027	3.9
	50%	6617	18.7	19829	3.7	18210	5.2	1621	0.9	12110	3.9
mcf	No Obf.	78	1.0	854	1.0	583	1.0	273	1.0	153	1.0
	10%	5159	66.1	13352	15.6	11759	20.2	1595	5.8	8761	57.3
	20%	5302	68.0	13500	15.8	12079	20.7	1423	5.2	8761	57.3
	30%	5449	69.9	14233	16.7	13086	22.4	1149	4.2	8792	57.5
	40%	5519	70.8	13922	16.3	12926	22.2	998	3.7	8739	57.1
	50%	5697	73.0	14076	16.5	13464	23.1	614	2.2	8686	56.8
regexp	No Obf.	72	1.0	855	1.0	591	1.0	266	1.0	1135	1.0
	10%	5053	70.2	13082	15.3	11447	19.4	1637	6.2	9675	8.5
	20%	5101	70.8	12964	15.2	11428	19.3	1538	5.8	9491	8.4
	30%	5276	73.3	13290	15.5	11802	20.0	1490	5.6	9530	8.4
	40%	5309	73.7	13064	15.3	11704	19.8	1362	5.1	9405	8.3
	50%	5375	74.7	13292	15.5	11940	20.2	1354	5.1	9393	8.3
svm	No Obf.	511	1.0	5375	1.0	3545	1.0	1832	1.0	2972	1.0
	10%	5734	11.2	19156	3.6	15777	4.5	3381	1.8	11729	3.9
	20%	6343	12.4	19912	3.7	17368	4.9	2546	1.4	11658	3.9
	30%	6739	13.2	20752	3.9	18533	5.2	2221	1.2	11521	3.9
	40%	7052	13.8	20680	3.8	19049	5.4	1633	0.9	11547	3.9
	50%	7661	15.0	21119	3.9	20135	5.7	986	0.5	11552	3.9
oftpd	No Obf.	455	1.0	2035	1.0	1667	1.0	370	1.0	1277	1.0
	10%	5541	12.2	14591	7.2	13011	7.8	1582	4.3	9856	7.7
	20%	5710	12.5	15110	7.4	13812	8.3	1300	3.5	9923	7.8
	30%	5810	12.8	15501	7.6	14422	8.7	1081	2.9	9911	7.8
	40%	5853	12.9	15875	7.8	15086	9.0	791	2.1	9858	7.7
	50%	6048	13.3	16493	8.1	16108	9.7	387	1.0	9954	7.8
mongoose	No Obf.	1027	1.0	2788	1.0	2086	1.0	704	1.0	493	1.0
	10%	6288	6.1	15981	5.7	14262	6.8	1721	2.4	9495	19.3
	20%	6525	6.4	16464	5.9	15102	7.2	1364	1.9	9474	19.2
	30%	6762	6.6	17115	6.1	16079	7.7	1038	1.5	9491	19.3
	40%	6784	6.6	17597	6.3	16924	8.1	675	1.0	9447	19.2
	50%	7024	6.8	18470	6.6	18369	8.8	103	0.1	9450	19.2

Table A.2: Program Complexity of CV-Obfuscated Binaries at Different Obfuscation Levels

Program	Obfuscation Level	# of Call Graph Edges		# of CFG Edges		# of Basic Blocks		Cyclomatic Number		Knot Count	
		Value	Ratio	Value	Ratio	Value	Ratio	Value	Ratio	Value	Ratio
bzip2	No Obf.	353	1.0	5382	1.0	3528	1.0	1856	1.0	3120	1.0
	10%	424	1.2	4079	0.8	2962	0.8	1119	0.6	645	0.2
	20%	385	1.1	3988	0.7	2906	0.8	1084	0.6	630	0.2
	30%	261	0.7	3868	0.7	2826	0.8	1044	0.6	713	0.2
	40%	248	0.7	3675	0.7	2684	0.8	993	0.5	699	0.2
	50%	242	0.7	3652	0.7	2684	0.8	970	0.5	696	0.2
mcf	No Obf.	78	1.0	854	1.0	583	1.0	273	1.0	153	1.0
	10%	36	0.5	531	0.6	377	0.6	156	0.6	71	0.5
	20%	36	0.5	520	0.6	370	0.6	152	0.6	71	0.5
	30%	34	0.4	461	0.5	329	0.6	134	0.5	68	0.4
	40%	24	0.3	372	0.4	270	0.5	104	0.4	54	0.4
	50%	33	0.4	308	0.4	223	0.4	87	0.3	46	0.3
regexp	No Obf.	72	1.0	855	1.0	591	1.0	266	1.0	1135	1.0
	10%	69	1.0	589	0.7	410	0.7	181	0.7	619	0.5
	20%	67	0.9	578	0.7	411	0.7	169	0.6	618	0.5
	30%	66	0.9	525	0.6	377	0.6	150	0.6	603	0.5
	40%	58	0.8	330	0.4	243	0.4	89	0.3	117	0.1
	50%	57	0.8	326	0.4	253	0.4	75	0.3	117	0.1
svm	No Obf.	511	1.0	5375	1.0	3545	1.0	1832	1.0	2972	1.0
	10%	403	0.8	3612	0.7	2576	0.7	1038	0.6	324	0.1
	20%	382	0.7	3431	0.6	2460	0.7	973	0.5	313	0.1
	30%	357	0.7	3267	0.6	2358	0.7	911	0.5	302	0.1
	40%	337	0.7	3129	0.6	2259	0.6	872	0.5	293	0.1
	50%	311	0.6	2971	0.6	2147	0.6	826	0.5	282	0.1
oftpd	No Obf.	455	1.0	2035	1.0	1667	1.0	370	1.0	1277	1.0
	10%	444	1.0	1923	0.9	1582	0.9	343	0.9	1097	0.9
	20%	411	0.9	1786	0.9	1454	0.9	334	0.9	1065	0.8
	30%	390	0.9	1727	0.8	1435	0.9	294	0.8	542	0.4
	40%	333	0.7	1500	0.7	1237	0.7	265	0.7	972	0.8
	50%	307	0.7	1384	0.7	1158	0.7	228	0.6	944	0.7
mongoose	No Obf.	1027	1.0	2788	1.0	2086	1.0	704	1.0	493	1.0
	10%	717	0.7	2489	0.9	1934	0.9	557	0.8	526	1.1
	20%	644	0.6	2239	0.8	1786	0.9	455	0.6	462	0.9
	30%	585	0.6	2063	0.7	1638	0.8	427	0.6	442	0.9
	40%	532	0.5	1954	0.7	1555	0.7	401	0.6	430	0.9
	50%	467	0.5	1787	0.6	1424	0.7	365	0.5	400	0.8

Appendix B

Publications During Ph.D.

- *Protecting Million-User iOS Apps with Obfuscation: Motivations, Pitfalls, and Experience.* **Pei Wang**, Dinghao Wu, Zhaofeng Chen, and Tao Wei. In the 40th International Conference on Software Engineering, the Software Engineering In Practice Track, 2018. (ICSE '18, SEIP)

- *Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation.* **Pei Wang**, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. In the 40th International Conference on Software Engineering, 2018. (ICSE '18)

- *Binary Code Retrofitting and Hardening Using SGX.* Shuai Wang, Wenhao Wang, Qinkun Bao, **Pei Wang**, XiaoFeng Wang, and Dinghao Wu. In the 2nd Workshop on Forming an Ecosystem Around Software Transformation, 2017. (FEAST '17, co-located with CCS '17)

- *Lambda Obfuscation.* Pengwei Lan, **Pei Wang**, Shuai Wang, and Dinghao Wu. In the 13th EAI International Conference on Security and Privacy in Communication Networks, 2017. (SecureComm '17)

- *Turing Obfuscation.* Yan Wang, Shuai Wang, **Pei Wang**, and Dinghao Wu. In the 13th EAI International Conference on Security and Privacy in Communication Networks, 2017. (SecureComm '17)

- *Semantics-Aware Machine Learning for Function Recognition in Binary Code.* Shuai Wang, **Pei Wang**, and Dinghao Wu. In the 33rd IEEE International Conference on Software Maintenance and Evolution, 2017. (ICSME '17)

- *Composite Software Diversification.* Shuai Wang, **Pei Wang**, and Dinghao Wu. In the 33rd IEEE International Conference on Software Maintenance and Evolution, 2017. (ICSME '17)

- *CacheD: Identifying Cache-Based Timing Channels in Production Software.* Shuai Wang, **Pei Wang**, Xiao Liu, Danfeng Zhang, and Dinghao Wu. In the 26th USENIX Security Symposium, 2017. (USENIX Security '17)

- *LibD: Scalable and Precise Third-party Library Detection in Android Markets.* Menghao Li, Wei Wang, **Pei Wang**, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. In the 39th ACM/IEEE International Conference on Software Engineering, 2017. (ICSE '17)

- *CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump.* Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, **Pei Wang**, and Peng Liu. In the 23rd ACM Conference on Computer and Communications Security, 2016. (CCS '16)

- *Translingual Obfuscation.* **Pei Wang**, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. In the 1st IEEE European Symposium on Security and Privacy, 2016. (EuroS&P '16)

- UROBOROS*: Instrumenting Stripped Binaries with Static Reassembling.* Shuai Wang, **Pei Wang**, and Dinghao Wu. In the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, 2016. (SANER '16)

- *Reassembleable Disassembling.* Shuai Wang, **Pei Wang**, and Dinghao Wu. In the 24th USENIX Security Symposium, 2015. (USENIX Security '15)

Bibliography

[1] Anand Prakash: How anyone could have used Uber to ride for free! http://www.anandpraka.sh/2017/03/how-anyone-could-have-used-uber-to-ride.html.

[2] Apple: most popular app store categories 2017 — statistic. https://www.statista.com/statistics/270291/popular-categories-in-the-app-store/.

[3] Binary Diff (bdiff). http://sourceforge.net/projects/bdiff/.

[4] BinDiff. http://www.zynamics.com/bindiff.html.

[5] Black mirror investigation: A report on the bussiness of "click-farming". http://image.3001.net/uploads/pdf/4aa87c46888173995c295a873c2aa682.pdf.

[6] C++ to Java converter. http://www.tangiblesoftwaresolutions.com/Product_Details/CPlusPlus_to_Java_Converter_Details.html.

[7] Code Virtualizer: Total obfuscation against reverse engineering. http://oreans.com/codevirtualizer.php.

[8] Cycript. http://www.cycript.org/.

[9] Darpa-baa-10-36, cyber genome program. https://www.fbo.gov/index?id=d477d43219a5a189e02e190c6b81ac6a&_cview=1.

[10] DarunGrim: A patch analysis and binary diffing tool. http://www.darungrim.org/.

[11] Frida · a world-class dynamic instrumentation framework. www.frida.re/.

[12] GitHub - pjebs/Obfuscator-iOS: Secure your app by obfuscating all the hard-coded security-sensitive strings. https://github.com/pjebs/Obfuscator-iOS.

[13] GitHub - preemptive/PPiOS-Rename: Symbol obfuscator for iOS apps. https://github.com/preemptive/PPiOS-Rename.

[14] GitHub - WhisperSystems/Signal-iOS: A private messenger for iOS. https://github.com/WhisperSystems/Signal-iOS.

[15] IDA: About. https://www.hex-rays.com/products/ida/.

[16] The international obfuscated C code contest. http://www.ioccc.org.

[17] iOS apps caught using private APIs. http://sourcedna.com/blog/20151018/ios-apps-using-private-apis.html.

[18] iTunes connect developer guide. https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/About.html.

[19] The Lancaster corpus of mandarin Chinese. http://www.lancaster.ac.uk/fass/projects/corpus/LCMC/.

[20] Maximum build file sizes. https://help.apple.com/itunes-connect/developer/#/dev611e0a21f.

[21] Monument Valley apparently has a 95% piracy rate on Android, 60% on iOS. https://goo.gl/TkfCIK.

[22] Protecting better with Code Virtualizer. http://www.oreans.com/Release/ProtectBetter.pdf.

[23] Seizing opportunity through license compliance. http://globalstudy.bsa.org/2016/downloads/studies/BSA_GSS_US.pdf.

[24] Shrink your code and resources — Android studio - Android developers. https://developer.android.com/studio/build/shrink-code.html.

[25] Smart obfuscation for iOS apps — PreEmptive Protection. https://www.preemptive.com/products/ppios.

[26] SWI-Prolog manual. http://www.swi-prolog.org/man/threads.html.

[27] VirusTotal - free online virus, malware and URL scanner. https://www.virustotal.com/.

[28] VMProtect software protection. http://vmpsoft.com.

[29] Andow, B., Nadkarni, A., Bassett, B., Enck, W., and Xie, T. A study of grayware on google play. In *Proceedings of the 2016 IEEE Workshop on Mobile Security Technologies* (2016), MoST '16.

[30] Apon, D., Huang, Y., Katz, J., and Malozemoff, A. J. Implementing cryptographic program obfuscation. Cryptology ePrint Archive, Report 2014/779, 2014. https://eprint.iacr.org/2014/779.

[31] Appel, A. W., and MacQueen, D. B. Standard ML of New Jersey. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming* (1991), PLILP '91.

[32] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., and Rieck, K. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the 2014 Network and Distributed System Security Symposium* (2014), NDSS '14.

[33] At-Kaci, H. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[34] Banescu, S., Ochoa, M., and Pretschner, A. A framework for measuring software obfuscation resilience against automated attacks. In *Proceedings of the 1st International Workshop on Software Protection* (2015), SPRO '15, pp. 45–51.

[35] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. On the (im)possibility of obfuscating programs. *J. ACM 59*, 2 (May 2012), 6:1–6:48.

[36] Bardin, S., David, R., and Marion, J.-Y. Backward-bounded DSE: Targeting infeasibility questions on obfuscated codes. In *Proceedings of the 38th IEEE Symposium on Security and Privacy* (2017), SP '17, pp. 633–651.

[37] Bichsel, B., Raychev, V., Tsankov, P., and Vechev, M. Statistical deobfuscation of Android applications. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security* (2016), CCS '16, pp. 343–355.

[38] Bonfante, G., Fernandez, J., Marion, J.-Y., Rouxel, B., Sabatier, F., and Thierry, A. CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS '15, pp. 745–756.

[39] Brumley, D., Jager, I., Avgerinos, T., and Schwartz, E. J. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (2011), CAV '11, pp. 463–469.

[40] Brumley, D., Poosankam, P., Song, D., and Zheng, J. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy* (2008), SP '08.

[41] Buddrus, F., and Schödel, J. Cappuccino—A C++ to Java translator. In *Proceedings of the 1998 ACM Symposium on Applied Computing* (1998), SAC '98, pp. 660–665.

[42] Caliskan, A., Yamaguchi, F., Dauber, E., Harang, R., Rieck, K., Greenstadt, R., and Narayanan, A. When coding style survives compilation: De-anonymizing programmers from executable binaries. In *Proceedings of 25th Network and Distributed System Security Symposium* (2018), NDSS '18.

[43] Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., and Greenstadt, R. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Security Symposium* (2015), USENIX Security '15, pp. 255–270.

[44] Ceccato, M., Penta, M., Falcarin, P., Ricca, F., Torchiano, M., and Tonella, P. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Softw. Engg. 19*, 4 (Aug. 2014), 1040–1074.

[45] Chen, H., Yuan, L., Wu, X., Zang, B., Huang, B., and Yew, P.-C. Control flow obfuscation with information flow tracking. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), MICRO '42, pp. 391–400.

[46] Chen, K., Liu, P., and Zhang, Y. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering* (2014), ICSE '14, pp. 175–186.

[47] Chen, K., Wang, X., Chen, Y., Wang, P., Lee, Y., Wang, X., Ma, B., Wang, A., Zhang, Y., and Zou, W. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on Android and iOS. In *Proceedings of the 37th IEEE Symposium on Security and Privacy* (2016), S&P '16, pp. 357–376.

[48] Chen, Z. iOS masque attack weaponized: A real world look. https://www.fireeye.com/blog/threat-research/2015/08/ios_masque_attackwe.html.

[49] Chow, S., Gu, Y., Johnson, H., and Zakharov, V. A. An approach to the obfuscation of control-flow of sequential computer programs. In *Proceedings of the 4th International Conference on Information Security* (2001), ISC '01, pp. 144–155.

[50] Cohen, F. B. Operating system protection through program evolution. *Comput. Secur. 12*, 6 (Oct. 1993), 565–584.

[51] Collberg, C., Thomborson, C., and Low, D. A taxonomy of obfuscating transformations. Tech. rep., The University of Auckland, 1997.

[52] Collberg, C., Thomborson, C., and Low, D. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1998), POPL '98, pp. 184–196.

[53] Conte, S. D., Dunsmore, H. E., and Shen, V. Y. *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co., Inc., 1986.

[54] Coogan, K., Lu, G., and Debray, S. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS '11, pp. 275–284.

[55] Cozzie, A., Stratton, F., Xue, H., and King, S. T. Digging for data structures. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), OSDI'08, pp. 255–266.

[56] Dave, V., Guha, S., and Zhang, Y. Measuring and fingerprinting click-spam in ad networks. *SIGCOMM Comput. Commun. Rev. 42*, 4 (Aug. 2012), 175–186.

[57] Demaine, E. D. C to Java: converting pointers into references. *Concurrency - Practice and Experience 10*, 11-13 (1998), 851–861.

[58] Diaz, D., and Codognet, P. Design and implementation of the GNU Prolog system. *Journal of Functional and Logic Programming 2001*, 6 (2001).

[59] Domas, C. Turning 'mov' into a soul-curshing RE nightmare. In *Proceeding of the 2015 Annual Reverse Engineering and Security Conference*, REcon '15.

[60] Drewry, W., and Ormandy, T. Flayer: Exposing application internals. In *Proceedings of the 1st USENIX Workshop on Offensive Technologies* (2007), WOOT '07.

[61] Egele, M., Woo, M., Chapman, P., and Brumley, D. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceeding of the 23rd USENIX Security Symposium* (2014), USENIX Security '14, pp. 303–317.

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

[63] Flake, H. Structural comparison of executable objects. In *Proceedings of the 1st SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment* (2004), DIMVA '04, pp. 161–173.

[64] Fletcher, C. W., Ren, L., Kwon, A., Van Dijk, M., Stefanov, E., Serpanos, D., and Devadas, S. A low-latency, low-area hardware oblivious RAM controller. In *Proceedings of the 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines* (2015), FCCM '15, pp. 215–222.

[65] Foket, C., Sutter, B. D., and Bosschere, K. D. Pushing java type obfuscation to the limit. *IEEE Transactions on Dependable and Secure Computing 11*, 6 (Nov 2014), 553–567.

[66] Foket, C., Sutter, B. D., and Bosschere, K. D. Pushing java type obfuscation to the limit. *IEEE Transactions on Dependable and Secure Computing 11*, 6 (Nov 2014), 553–567.

[67] Franz, A., and Brants, T. All our n-gram are belong to you. https://research.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html.

[68] Ganesh, V., and Dill, D. L. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (2007), CAV'07, pp. 519–531.

[69] Gao, D., Reiter, M. K., and Song, D. BinHunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 4th International Conference on Information and Communications Security* (2008), ICICS '08.

[70] Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., and Waters, B. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science* (2013), FOCS '13, pp. 40–49.

[71] Ghiya, R., and Hendren, L. J. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1996), POPL '96.

[72] Ghosh, S., Hiser, J., and Davidson, J. W. Replacement attacks against vm-protected applications. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (2012), VEE '12, pp. 203–214.

[73] Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., and Choi, H. AdRob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pp. 431–444.

[74] Gilb, T. *Software Metrics*. Winthrop computer systems series. Winthrop Publishers, 1977.

[75] Glanz, L., Amann, S., Eichberg, M., Reif, M., Hermann, B., Lerch, J., and Mezini, M. CodeMatch: Obfuscation won't conceal your repackaged app. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering* (2017), ESEC/FSE '17, pp. 638–648.

[76] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious rams. *J. ACM 43*, 3 (May 1996), 431–473.

[77] Guo, F., Ferrie, P., and Chiueh, T. A study of the packer problem and its solutions. In *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection* (2008), RAID '08, pp. 98–115.

[78] Hada, S. Zero-knowledge and code obfuscation. In *Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology* (London, UK, UK, 2000), ASIACRYPT '00, Springer-Verlag, pp. 443–457.

[79] Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. On the naturalness of software. In *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering* (2012), ICSE '12, pp. 837–847.

[80] Hubbard, J., Weimer, K., and Chen, Y. A study of SSL proxy attacks on Android and iOS mobile applications. In *Proceedings of the 11th IEEE Consumer Communications and Networking Conference* (2014), CCNC '14, pp. 86–91.

[81] Jones, N. D., and Muchnick, S. S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1982), POPL '82, pp. 66–74.

[82] Junod, P., Rinaldini, J., Wehrli, J., and Michielin, J. Obfuscator-LLVM – software protection for the masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection* (2015), SPRO'15, pp. 3–9.

[83] Kamin, S. N. *Programming Languages: An Interpreter-Based Approach.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[84] Kettle, N., and King, A. Bit-precise reasoning with affine functions. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning* (2008), SMT '08/BPR '08, pp. 46–52.

[85] Knuth, D. E., and Floyd, R. W. Notes on avoiding 'go to' statements. *Information Processing Letters 1*, 1 (1971), 23–31.

[86] Laffra, C. C2J: A C++ to Java translator. *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips* (2001).

[87] Lakhotia, A., Boccardo, D. R., Singh, A., and Manacero, Jr., A. Context-sensitive analysis without calling-context. *Higher Order Symbol. Comput. 23*, 3 (Sept. 2010), 275–313.

[88] Lan, P., Wang, P., Wang, S., and Wu, D. Lambda obfuscation. In *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks* (2017), SecureComm '17.

[89] László, T., and Kiss, Á. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica 30* (2009), 3–19.

[90] Launchbury, J. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1993), POPL '93, pp. 144–154.

[91] Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., Xue, R., and Huo, W. LibD: Scalable and precise third-party library detection in Android markets. In *Proceedings of the 39th ACM/IEEE International Conference on Software Engineering* (2017), ICSE '17.

[92] Li, W., Li, H., Chen, H., and Xia, Y. AdAttester: Secure online mobile advertisement attestation using TrustZone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (2015), MobiSys '15, pp. 75–88.

[93] Linares-Vásquez, M., Holtzhauer, A., Bernal-Cárdenas, C., and Poshyvanyk, D. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (2014), MSR '14.

[94] Linn, C., and Debray, S. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (2003), CCS '03, pp. 290–299.

[95] Liu, C., Harris, A., Maas, M., Hicks, M., Tiwari, M., and Shi, E. GhostRider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ASPLOS '15, pp. 87–101.

[96] Liu, H., Sun, C., Su, Z., Jiang, Y., Gu, M., and Sun, J. Stochastic optimization of program obfuscation. In *Proceedings of the 39th International Conference on Software Engineering* (2017), ICSE '17, pp. 221–231.

[97] Luo, L., Ming, J., Wu, D., Liu, P., and Zhu, S. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), FSE '14, pp. 389–400.

[98] Lyda, R., and Hamrock, J. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy 5*, 2 (2007), 40–45.

[99] Ma, Z., Wang, H., Guo, Y., and Chen, X. LibRadar: Fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion* (2016), ICSE '16 Companion, pp. 653–656.

[100] Marlow, S., Yakushev, A. R., and Peyton Jones, S. Faster laziness using dynamic pointer tagging. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (2007), ICFP '07.

[101] Martignoni, L., Christodorescu, M., and Jha, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference* (2007), ACSAC '07, pp. 431–441.

[102] Martin, J., and Muller, H. Strategies for migration from C to Java. In *Fifth European Conference on Software Maintenance and Reengineering, 2001* (2001), pp. 200–209.

[103] McCabe, T. J. A complexity measure. *IEEE Trans. Softw. Eng. 2*, 4 (July 1976), 308–320.

[104] Ming, J., Pan, M., and Gao, D. iBinHunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th Annual International Conference on Information Security and Cryptology* (2012), ICISC '12.

[105] Ming, J., Xu, D., Wang, L., and Wu, D. LOOP: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), CCS '15, pp. 757–768.

[106] Moser, A., Kruegel, C., and Kirda, E. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference* (2007), ACSAC '07.

[107] Muchnick, S. S. *Advanced compiler design implementation.* Morgan Kaufmann, 1997.

[108] Nagra, J., and Collberg, C. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Pearson Education, 2009.

[109] Nayak, K., Fletcher, C., Ren, L., Chandran, N., Lokam, S., Shi, E., and Goyal, V. HOP: Hardware makes obfuscation practical. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium* (2017), NDSS '17.

[110] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction* (2002), CC '02.

[111] Ngo, M. N., and Tan, H. B. K. Detecting large number of infeasible paths through recognizing their patterns. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (2007), ESEC-FSE '07, pp. 215–224.

[112] Norvig, P. Natural language corpus data: Beautiful data. http://norvig.com/ngrams/.

[113] Orikogbo, D., Büchler, M., and Egele, M. CRiOS: Toward large-scale iOS application analysis. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices* (2016), SPSM '16, pp. 33–42.

[114] Paleari, R., Martignoni, L., Roglia, G. F., and Bruschi, D. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies* (2009), WOOT '09.

[115] Pawlowski, A., Contag, M., and Holz, T. Probfuscation: An obfuscation approach using probabilistic control flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '16. Springer, 2016, pp. 165–185.

[116] Popov, I. V., Debray, S. K., and Andrews, G. R. Binary obfuscation using signals. In *Proceedings of 16th USENIX Security Symposium* (2007), USENIX Security '07.

[117] Ramshaw, L. Eliminating go to's while preserving program structure. *Journal of the ACM 35*, 4 (1988), 893–920.

[118] Rasthofer, S., Arzt, S., Miltenberger, M., and Bodden, E. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *Proceedings of 23rd Network and Distributed System Security Symposium* (2016), NDSS '16.

[119] Rolles, R. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies* (2009), WOOT '09.

[120] Royal, P., Halpin, M., Dagon, D., Edmonds, R., and Lee, W. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference* (2006), ACSAC '06, pp. 289–300.

[121] Sagiv, M., Reps, T., and Wilhelm, R. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst. 20*, 1 (Jan. 1998), 1–50.

[122] Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., and Weippl, E. Protecting software through obfuscation: Can it keep pace with progress in code analysis? 4:1–4:37.

[123] Segaran, T., and Hammerbacher, J. *Beautiful data: the stories behind elegant data solutions.* "O'Reilly Media, Inc.", 2009.

[124] Sepp, A., Mihaila, B., and Simon, A. Precise static analysis of binaries by extracting relational information. In *Proceedings of the 18th Working Conference on Reverse Engineering* (2011), WCRE '11, pp. 357–366.

[125] Sharif, M., Lanzi, A., Giffin, J., and Lee, W. Impeding malware analysis using conditional code obfuscation. In *Proceedings of 15th Network and Distributed System Security Symposium* (2008), NDSS '08.

[126] Sharif, M., Lanzi, A., Giffin, J., and Lee, W. Automatic reverse engineering of malware emulators. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy* (2009), SP '09, pp. 94–109.

[127] Sikorski, M., and Honig, A. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software.* No Starch Press, 2012.

[128] Smith, J. E., and Nair, R. *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufmann, 2005.

[129] Szor, P. *The Art of Computer Virus Research and Defense.* Addison-Wesley Professional, February 2005.

[130] Tärnlund, S.-Å. Horn clause computability. *BIT Numerical Mathematics 17*, 2 (1977), 215–226.

[131] Thomas, D., and Rolf, R. Graph-based comparison of executable objects. In *Proceedings of the Symposium sur la Securite des Technologies de l'Information et des Communications* (2005), SSTIC '05.

[132] Trudel, M., Furia, C., Nordio, M., Meyer, B., and Oriol, M. C to O-O translation: Beyond the easy stuff. In *Proceedings of the 19th Working Conference on Reverse Engineering* (2012), WCRE '12, pp. 19–28.

[133] Wang, C., Hill, J., Knight, J., , and Davidson, J. Software tamper resistance: Obstructing static analysis of programs.

[134] Wang, H., Guo, Y., Ma, Z., and Chen, X. WuKong: A scalable and accurate two-phase approach to Android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015), ISSTA '15, pp. 71–82.

[135] Wang, P., Bao, Q., Wang, L., Wang, S., Chen, Z., Wei, T., and Wu, D. Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *Proceedings of the 40th International Conference on Software Engineering* (2018), ICSE '18.

[136] Wang, P., Wang, S., Ming, J., Jiang, Y., and Wu, D. Translingual obfuscation. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy* (2016), EuroS&P '16, pp. 128–144.

[137] Wang, P., Wu, D., Chen, Z., and Wei, T. Protecting million-user iOS apps with obfuscation: Motivations, pitfalls, and experience. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (2018), pp. 235–244.

[138] Wang, S., Chollak, D., Movshovitz-Attias, D., and Tan, L. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (2016), ASE '16, pp. 708–719.

[139] Wang, S., Wang, P., and Wu, D. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium* (2015), USENIX Security '15.

[140] Wang, Y., Wang, S., Wang, P., and Wu, D. Turing obfuscation. In *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks* (2017), SecureComm '17.

[141] Warren, D. H. D. An abstract Prolog instruction set. Tech. Rep. 309, SRI International, October 1983.

[142] Williams, M. H., and Chen, G. Restructuring pascal programs containing goto statements. *The Computer Journal 28*, 2 (1985), 134–137.

[143] Woodward, M., Hennell, M., and Hedley, D. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering SE-5*, 1 (Jan. 1979), 45–50.

[144] Wu, D., Appel, A. W., and Stump, A. Foundational proof checkers with small witnesses. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming* (2003), PPDP '03, pp. 264–274.

[145] Wu, Z., Gianvecchio, S., Xie, M., and Wang, H. Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (2010), CCS '10, pp. 536–546.

[146] Xu, D., Ming, J., and Wu, D. Generalized dynamic opaque predicates: A new control flow obfuscation method. In *Proceedings of the 19th Information Security Conference* (2016), ISC '16, pp. 323–342.

[147] Xu, D., Ming, J., and Wu, D. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *Proceedings of the 38th IEEE Symposium on Security and Privacy* (2017), pp. 921–937.

[148] Xu, W., Zhang, F., and Zhu, S. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In *Proceedings of the 7th International Conference on Malicious and Unwanted Software* (2012), MALWARE '12, pp. 9–16.

[149] Xue, L., Luo, X., Yu, L., Wang, S., and Wu, D. Adaptive unpacking of Android apps. In *Proceedings of the 39th International Conference on Software Engineering* (2017), ICSE '17, pp. 358–369.

[150] Xue, L., Luo, X., Yu, L., Wang, S., and Wu, D. Adaptive unpacking of android apps. In *Proceedings of the 39th International Conference on Software Engineering* (2017), ICSE '17, pp. 358–369.

[151] Yadegari, B., and Debray, S. Bit-level taint analysis. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation* (2014), SCAM '14, pp. 255–264.

[152] Yadegari, B., Johannesmeyer, B., Whitely, B., and Debray, S. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy* (2015), SP '15.

[153] Yang, W., Zhang, Y., Li, J., Shu, J., Li, B., Hu, W., and Gu, D. AppSpear: Bytecode decrypting and DEX reassembling for packed Android malware. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses* (2015), RAID '15, pp. 359–381.

[154] Zakai, A. Emscripten: An LLVM-to-JavaScript compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (2011), OOPSLA '11, pp. 301–312.

[155] Zhang, F., Huang, H., Zhu, S., Wu, D., and Liu, P. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless and Mobile Networks* (2014), WiSec '14, pp. 25–36.

[156] Zhou, Y., and Jiang, X. Dissecting Android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (2012), S&P '12, pp. 95–109.

Vita

Pei Wang finished his Ph.D. at the College of Information Sciences and Technology, The Pennsylvania State University. He was advised by Dr. Dinghao Wu. His research interest subsumes computer security, software engineering, and programming language. Before he went to Penn State, he received his master's degree from University of Waterloo (2012-2013) in Electrical and Computer Engineering, and his bachelor's degree from Peking University (2008-2012) in Computer Sciences and Technology.