Protecting Million-User iOS Apps with Obfuscation: Motivations, Pitfalls, and Experience

Pei Wang*
pxw172@ist.psu.edu
The Pennsylvania State University

Dinghao Wu
dwu@ist.psu.edu
The Pennsylvania State University

Zhaofeng Chen
chenzhaofeng@baidu.com
Baidu X-Lab

Tao Wei
lenx@baidu.com
Baidu X-Lab

ABSTRACT

In recent years, mobile apps have become the infrastructure of many popular Internet services. It is now fairly common that a mobile app serves a large number of users across the globe. Different from web-based services whose important program logic is mostly placed on remote servers, many mobile apps require complicated client-side code to perform tasks that are critical to the businesses. The code of mobile apps can be easily accessed by any party after the software is installed on a rooted or jailbroken device. By examining the code, skilled reverse engineers can learn various knowledge about the design and implementation of an app. Real-world cases have shown that the disclosed critical information allows malicious parties to abuse or exploit the app-provided services for unrightful profits, leading to significant financial losses for app vendors.

One of the most viable mitigations against malicious reverse engineering is to obfuscate the software before release. Despite that security by obscurity is typically considered to be an unsound protection methodology, software obfuscation can indeed increase the cost of reverse engineering, thus delivering practical merits for protecting mobile apps.

In this paper, we share our experience of applying obfuscation to multiple commercial iOS apps, each of which has millions of users. We discuss the necessity of adopting obfuscation for protecting modern mobile business, the challenges of software obfuscation on the iOS platform, and our efforts in overcoming these obstacles. Our report can benefit many stakeholders in the iOS ecosystem, including developers, security service providers, and Apple as the administrator of the ecosystem.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software reverse engineering**;

KEYWORDS

obfuscation, software protection, reverse engineering, mobile, iOS

*Work mostly performed during an internship at Baidu X-Lab.

ICSE-SEIP '18, May 27–June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183524

ACM Reference Format:
Pei Wang, Dinghao Wu, Zhaofeng Chen, and Tao Wei. 2018. Protecting Million-User iOS Apps with Obfuscation: Motivations, Pitfalls, and Experience. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27–June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3183519.3183524

1 INTRODUCTION

During the last decade, mobile devices and apps have become the foundations of many million-dollar businesses operated globally. However, the prosperity has drawn many malevolent attempts to make unjust profits by exploiting the security and privacy loopholes in popular mobile software.

In recent years, we noticed that security breaches targeting mobile apps are becoming more and more prevalent, with both of their scale and severity trending up at a worrying rate. Among all emerging threats, malicious and fraudulent campaigns, conducted through programmatically manipulating a massive number of mobile devices and faking a large volume of user activities [18], are particularly harmful to many large-scale mobile businesses. To minimize the impacts of those campaigns, app developers typically need to place certain hooks into the client code to detect suspicious user activities (see Section 3 for details). Attackers, on the other hand, try to sabotage or circumvent these defenses in order to commence their malicious activities without being noticed. Since most malicious activities targeting mobile apps rely on reverse engineering to tamper with the code, thwarting or weakening the reverse engineering capabilities of the attackers is considered to be a fairly cost-effective protection strategy.

By impeding reverse engineering, developers hold a chance to prevent or delay incoming attacks, buying time for long-term security enhancement and more permanent solutions to various security issues. To this end, software obfuscation plays an important role. The goal of obfuscation is to transform program code into a form that makes reverse engineering ineffective or uneconomical.

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.

In this paper, we report our experience of obfuscating multiple commercial iOS apps with millions of active users. Being one of the dominant mobile operating systems, iOS possesses the common characteristics of a mobile platform but also distinguishes itself from other systems for many unique features. It is known that software obfuscation has been quite prevalent in Android app development, but much less is known or studied for iOS. Many mobile developers now release their apps for both platforms. If the iOS version of an app is not effectively protected, attackers will have a good chance to exploit the app no matter how well the Android version is obfuscated.

To 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.

It should be emphasized that the major focus of this paper is not to propose new obfuscation techniques or evaluate their potency; instead, the point is to introduce how to operationalize obfuscation in real-world mobile app development.

The rest of the paper is organized as follows. We first introduce the background knowledge about software obfuscation in Section 2. We then explain why we are motivated to protect production iOS apps with obfuscation in Section 3. Our experiences and lessons are presented in Section 4, followed by the evaluation of our obfuscation techniques in Section 5. Section 6 discusses our prospect of iOS obfuscation and other protection methods, Section 7 reviews related work on obfuscation, and Section 8 concludes the paper.

2 SOFTWARE OBFUSCATION

2.1 Theoretical Foundation

According to the formalization by Barak et al. [15], an *effective* obfuscation technique 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 $\phi : \mathcal{F} \to \mathcal{S}$ 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 $\phi(f)$.

To better understand the definition above, take symmetric encryption as an example. Suppose there is a cipher E that takes a key k and consumes plaintext p to compute the ciphertext $E(k, p)$. If k is hard-coded into E, $E(k, \cdot)$ can be considered as a function taking plaintext as the sole argument, denoted by E_k. Let ϕ be the property function that decides the hard-coded key of a cipher, namely $\phi(E_k) = k$. Assuming E is resilient to chosen-plaintext attacks, computing ϕ will be prohibitively expensive if attackers can only access E as a black box oracle. However, it is possible to recover k in a reasonable amount of time by directly looking at P which is the code of an implementation of E_k. In that case, if a perfect obfuscator $\mathcal{O}$ exists and the implementation of E_k is released as $\mathcal{O}(P)$, any attempt to efficiently learn k by analyzing $\mathcal{O}(P)$ will fail.

It has been proven that a perfect obfuscator does not exist, even if the properties to hide are limited to $\{0, 1\}$-valued functions [15]. That is, analyzing the code of a program can always reveal at least 1 bit information about *what* the program computes without spending too much time, no matter how complicated the program code is rendered. Therefore, some theorists relaxed the security requirement for obfuscation instead of trying to develop a technique that is universally effective [27].

2.2 Obfuscation in Practice

Since a theoretically perfect solution to the generalize problem of software obfuscation is unfeasible, practitioners usually set limits to problem characteristics so that the problems can be addressed within a reasonable scope. In industry, the goal of obfuscation is not to make reverse engineering impossible but to increase the cost of it such that attacks can be delayed or diverted to relatively poorly protected targets.

A recent literature review classified obfuscation algorithms into three categories according to how they are implemented [40]. The first kind is data obfuscation that alters the structures in which data are stored in binaries. One typical data obfuscation technique is to statically encrypt the string literals and decode them at run time. The second kind is static code rewriting that transforms the executable code into a semantically equivalent but syntactically obscuring form. For example, a static code rewriting technique called movfuscator [26] can transform an x86 binary into a form that only contains *mov* instructions, making it difficult for reverse engineering tools to reconstruct the original control flow. The third kind is dynamic code rewriting, also known as self-modifying obfuscation. For programs protected by self-modifying obfuscation, the statically observable code is different from what is actually executed at run time. One of the most widely used dynamic rewriting obfuscation techniques is the packing method. A packer encodes the original code of the obfuscated program into data and dynamically decodes the data back into code during execution.

3 MOTIVATION

Before sharing our experience with iOS obfuscation, we would like to discuss the reasons that drove us to consider employing obfuscation in the first place. The rationale is twofold. We first explain the important role played by reverse engineering in malicious activities targeting mobile software. We then introduce why these threats are particularly realistic on iOS due to the lack of technical challenges in analyzing unprotected iOS apps.

3.1 Threats of Reverse Engineering

Many malevolent attempts to exploit mobile apps for illegal benefits heavily depend on reverse engineering. 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 four common kinds of them.

Intellectual property theft. This is a longstanding problem bothering commercial software developers. The piracy of desktop software causes millions of dollars of yearly economic loss [13]. On mobile platforms, the problem may be even more severe, since the digital right management of mobile apps is usually delegated to centralized app publishers and significantly relies on the security of the underlying mobile operating systems. If these systems are cracked (known as "root" for Android and "jailbreak" for iOS), attackers can easily pirate a large number of mobile apps in a short period.

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 [31]. 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 [20]. 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, we have encountered many incidents described above, among which the most concerning threat is the prevalence of large-scale malicious campaigns, as mentioned in Section 1. According to a report on the status of malicious campaigns in China [2], 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 of these campaigns are also evolving quickly. Campaign runners can now programmatically control hundreds of mobile devices without the need of human labor, while each device can host over 50 instances of the same mobile app. Figure1 shows an example of such technology.

Since the third quarter of 2016, we 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.

Figure 1: Programmatically controlling massive iOS devices as a service (http://shemeitong.com/index.php/anli/show/46.html).

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.

3.2 Reverse Engineering on iOS

From the research point of view, there exist various challenges in automated reverse engineering that cannot be easily addressed, which may lead 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, especially on iOS.

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 2 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 [8], 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

```objc
@implementation TSAnimatedAdapter
...

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

...
@end
```

(a) Original Objective-C source code

```objc
// 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 2: Decompiling an open source iOS app [7] with IDA Pro

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 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. 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 [11]. 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.

4 EXPERIENCE WITH iOS OBFUSCATION

Regarding obfuscation, our major objective is to protect a common code base shared by a group of commercial iOS apps. These apps span a wide range of functionality categories, including news, utility, navigation, payment, social networking, and shopping.

4.1 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 that can be arbitrarily combined per developer demands. The granularity of obfuscation is configurable through customized compiler flags and extended function attributes. Figure 3 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.

4.2 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

[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 paper include comments and blanks.

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

void foo() {
  ...
}

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

(a) Obfuscate the whole compilation unit

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

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

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

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

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

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

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

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

Figure 3: Obfuscation configuration examples

strictly prohibited on iOS, leaving dynamic code rewriting obfuscation unfeasible. For this reason, many packer-based obfuscation techniques that are popular on Android [47] 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 [10]. 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 cannot afford obfuscation algorithms that bloat the software size too much. That includes virtualization-based obfuscation [24, 41], 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 [9].[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 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 [23].
(5) *Control flow flattening* that obscures the logic relations between program basic blocks [21].
(6) *Garbage instruction insertion* that injects garbage code that is irrelevant to program functionality [22].

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*, a feature supported by many implementations of C-family languages and LLVM itself. Figure 4 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 Section 4.3, 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 [5, 6, 32]. We intentionally made our implementation different from the public ones

[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 paper 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 paper 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.

```
; @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 4: Example of obfuscation utilizing LLVM IR inline assembly

by altering code patterns and introducing new features. Attackers will need more sophisticated techniques to nullify the mutated obfuscation effects [44]. 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 the paper is not developing or evaluating new obfuscation methods, but maximizing the value of existing techniques in practical engineering.

4.3 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.

4.3.1 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 [16].

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.

4.3.2 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. This is similar to what we have emphasized in Section 1 about protecting iOS and Android apps with comparable effort.

4.3.3 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 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. [23], resilience indicates how well the obfuscation can withstand *automated* reverse engineering. As for overhead measurement, we focus on binary size expansion and execution slowdown.

5.1 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 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 [27], which is still not practical for protecting real-world software [36]. 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 [14, 37, 42], 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 [14, 17], 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 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 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 Section 4.3, it is crucial for an iOS obfuscator to protect binaries of different architectures equally well. When interpreting the results in Table 1 and Figure 5, 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.

5.2 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 in Section 4, 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 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.

Table 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

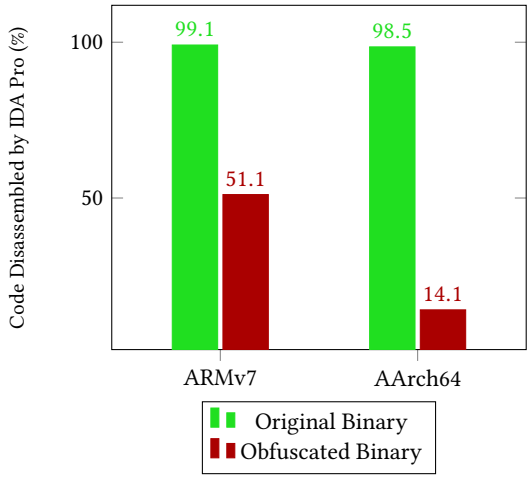

Figure 5: Effectiveness of disassembly disruption

5.2.1 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 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 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%)

5.2.2 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.

6 DISCUSSION

6.1 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.

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

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 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.

6.2 Other Protections

Obfuscation is not a panacea for combating the security threats targeting mobile apps and there have been many deobfuscation techniques proposed [24, 35, 45, 46]. 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 [3] and Frida [4] 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. As explained in Section 3, 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 [25].

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.

7 RELATED WORK

Software obfuscation has been intensively researched from both theoretical and practical perspectives. It has been formally proved that a universally effective obfuscator is not possible to implement [15, 28]. It is however feasible to build secure obfuscation constructs with limited generality, such as indistinguishability obfuscation [15] for polynomial-sized circuits [27] and the best possible obfuscation that tries to minimize rather than eliminate information leakage [29].

On the practical side, various obfuscation techniques have been developed and some of them have been shown helpful to software protection. Compared with early inventions like the ones introduced in Section 4, recently proposed solutions employ more advanced system and language features for obfuscation purposes. Popov et al. [38] proposed to replace branches with exceptions and reroute the control flows with customized exception handlers. Chen et al. [19] employed the architectural support of Itanium processors for information flow tracking, i.e., utilizing the deferred exception tokens embedded in Itanium registers to encode opaque predicates. Wang et al. [42], Lan et al. [33], and Wang et al. [43] obfuscated C programs by translating them into declarative languages or abstract computation models, making imperative-oriented analysis heuristics much less effective. There are also obfuscation methods that are less dependent on special software and hardware features but utilize more general techniques like compression, encryption, and virtualization [30, 34, 39, 48].

Most practically usable obfuscation tools supporting iOS are commercial. Obfuscator-LLVM [32] is an open source project that provides the implementation of several well known obfuscation algorithms within LLVM, thus suitable for iOS app protection. Unfortunately, the tool is no longer actively maintained. Tigress [12] is a source-level obfuscator supporting the C programming language. Protections provided by Tigress are mostly heavy weight, e.g., virtualization-based obfuscation and self modification, some of which are not applicable to iOS apps.

8 CONCLUSION

In this paper, we shared our experience with applying software obfuscation to iOS mobile apps in realistic software development settings. We revealed the threats of various malicious activities targeting mobile apps, including those conducted by well organized groups of underground economy practitioners. We then discussed why iOS app vendors should seriously consider protecting their apps by software obfuscation and what efforts need to be made for obfuscation to be practical when applied to complicated apps with large user bases. In particular, we summarized the major pitfalls that may prevent iOS developers from utilizing obfuscation effectively and efficiently, together with our responses to these challenges. We presented quantitative results on the resilience and cost of our iOS obfuscator. The evaluation is conducted on a common code base included by multiple commercial iOS apps serving a large number of users. The results show that software obfuscation, being

an technique accessible to common mobile developers, can indeed provide reasonably effective protection with modest cost.

ACKNOWLEDGMENTS

The work was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-1-2912, N00014-16-1-2265, and N00014-17-1-2894.

REFERENCES

[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] Black mirror investigation: A report on the bussiness of "click-farming". http://image.3001.net/uploads/pdf/4aa87c46888173995c295a873c2aa682.pdf.
[3] Cycript. http://www.cycript.org/.
[4] Frida · A world-class dynamic instrumentation framework. www.frida.re/.
[5] GitHub - pjebs/Obfuscator-iOS: Secure your app by obfuscating all the hard-coded security-sensitive strings. https://github.com/pjebs/Obfuscator-iOS.
[6] GitHub - preemptive/PPiOS-Rename: Symbol obfuscator for iOS apps. https://github.com/preemptive/PPiOS-Rename.
[7] GitHub - WhisperSystems/Signal-iOS: A private messenger for iOS. https://github.com/WhisperSystems/Signal-iOS.
[8] IDA: About. https://www.hex-rays.com/products/ida/.
[9] iTunes Connect Developer Guide. https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/About.html.
[10] Maximum build file sizes. https://help.apple.com/itunes-connect/developer/#/dev611e0a21f.
[11] Shrink Your Code and Resources | Android Studio - Android Developers. https://developer.android.com/studio/build/shrink-code.html.
[12] The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/.
[13] Unlicensed Software Use Still High Globally Despite Costly Cybersecurity Threats. http://globalstudy.bsa.org/2016/index.html.
[14] Sebastian Banescu, Martín Ochoa, and Alexander Pretschner. 2015. A Framework for Measuring Software Obfuscation Resilience Against Automated Attacks. In *Proceedings of the 1st International Workshop on Software Protection (SPRO '15)*. 45–51.
[15] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2012. On the (Im)Possibility of Obfuscating Programs. *J. ACM* 59, 2 (May 2012), 6:1–6:48.
[16] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. 2015. 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 (CCS '15)*. 745–756.
[17] Mariano Ceccato, Massimiliano Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. 2014. A Family of Experiments to Assess the Effectiveness and Efficiency of Source Code Obfuscation Techniques. *Empirical Softw. Engg.* 19, 4 (Aug. 2014), 1040–1074.
[18] Hao Chen, Daojing He, Sencun Zhu, and Jingshun Yang. 2017. Toward Detecting Collusive Ranking Manipulation Attackers in Mobile App Markets. In *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security (ASIACCS '17)*. 58–70.
[19] Haibo Chen, Liwei Yuan, Xi Wu, Binyu Zang, Bo Huang, and Pen-Chung Yew. 2009. Control Flow Obfuscation with Information Flow Tracking. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '42)*. 391–400.
[20] Zhaofeng Chen. iOS Masque Attack Weaponized: A Real World Look. https://www.fireeye.com/blog/threat-research/2015/08/ios_masque_attackwe.html.
[21] Stanley Chow, Yuan Gu, Harold Johnson, and Vladimir A. Zakharov. 2001. An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. In *Proceedings of the 4th International Conference on Information Security (ISC '01)*. 144–155.
[22] Frederick B. Cohen. 1993. Operating System Protection Through Program Evolution. *Comput. Secur.* 12, 6 (Oct. 1993), 565–584.
[23] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. 184–196.
[24] Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. 275–284.
[25] Vacha Dave, Saikat Guha, and Yin Zhang. 2012. Measuring and Fingerprinting Click-spam in Ad Networks. *SIGCOMM Comput. Commun. Rev.* 42, 4 (Aug. 2012), 175–186.
[26] Christopher Domas. Turning 'mov' Into a Soul-Curshing RE Nightmare. In *Proceeding of the 2015 Annual Reverse Engineering and Security Conference (REcon '15)*.
[27] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. 2013. Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits. In *Proceedings of the 54th IEEE Annual Symposium on Foundations of Computer Science (FOCS '13)*. 40–49.
[28] Shafi Goldwasser and Yael Tauman Kalai. 2005. On the Impossibility of Obfuscation with Auxiliary Input. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS '05)*. 553–562.
[29] Shafi Goldwasser and Guy N. Rothblum. 2007. On Best-possible Obfuscation. In *Proceedings of the 4th Conference on Theory of Cryptography (TCC '07)*. 194–213.
[30] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. 2008. A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID '08)*. 98–115.
[31] John Hubbard, Ken Weimer, and Yu Chen. 2014. A Study of SSL Proxy Attacks on Android and iOS Mobile Applications. In *Proceedings of the 11th IEEE Consumer Communications and Networking Conference (CCNC '14)*. 86–91.
[32] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO'15)*. 3–9.
[33] Pengwei Lan, Pei Wang, Shuai Wang, and Dinghao Wu. 2017. Lambda Obfuscation. In *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks (SECURECOMM '17)*.
[34] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. 2007. OmniUnack: Fast, Generic, and Safe Unpacking of Malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference*. 431–441.
[35] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. 757–768.
[36] Kartik Nayak, Christopher Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. 2017. HOP: Hardware Makes Obfuscation Practical. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS '17)*.
[37] Andre Pawlowski, Moritz Contag, and Thorsten Holz. 2016. Probfuscation: An Obfuscation Approach using Probabilistic Control Flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 165–185.
[38] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. 2007. Binary Obfuscation Using Signals. In *Proceedings of 16th USENIX Security Symposium (USENIX Security '07)*.
[39] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. 2006. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*. 289–300.
[40] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1 (2016), 4:1–4:37.
[41] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P '09)*. 94–109.
[42] Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. 2016. Translingual Obfuscation. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P '16)*.
[43] Yan Wang, Shuai Wang, Pei Wang, and Dinghao Wu. 2017. Turing Obfuscation. In *Proceedings of the 13th EAI International Conference on Security and Privacy in Communication Networks (SECURECOMM '17)*.
[44] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2016. Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method. In *Proceedings of the 19th Information Security Conference (ISC '16)*. 323–342.
[45] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*. 921–937.
[46] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive Unpacking of Android Apps. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. 358–369.
[47] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. 2015. AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID '15)*. 359–381.
[48] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards Obfuscation-resilient Mobile Application Repackaging Detection. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks (WiSec '14)*. 25–36.