Received: 23 May 2018 | Revised: 26 August 2018 | Accepted: 26 September 2018

DOI: 10.1002/spe.2648

EXPERIENCE REPORT

WILEY

Field experience with obfuscating million-user iOS apps in large enterprise mobile development

Pei Wang[1,2] | Dinghao Wu[1] | Zhaofeng Chen[2] | Tao Wei[2]

[1]Pennsylvania State University, University Park, Pennsylvania

[2]Baidu X-Lab, Sunnyvale, USA

Correspondence
Pei Wang, Baidu X-Lab, Sunnyvale, CA 94089.
Email: wangpei10@baidu.com

Funding information
National Science Foundation, Grant/Award Number: CNS-1652790; Office of Naval Research, Grant/Award Number: N00014-16-1-2265, N00014-16-1-2912, and N00014-17-1-2894

Summary

In recent years, mobile apps have become the infrastructure of many popular Internet services. It is now common that a mobile app serves millions of users across the globe. By examining the code of these apps, reverse engineers can learn various knowledge about the design and implementation of the apps. 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. One of the most viable mitigations against malicious reverse engineering is to obfuscate the apps. 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. We especially focus on factors that are unique to mobile software development that may affect the design and deployment of obfuscation techniques. We report the outcome of our obfuscation with empirical experiments. We additionally elaborate on the follow-up case studies about how our obfuscation affected the app publication process and how we responded to the negative impacts. This experience report can benefit mobile developers, security service providers, and Apple as the administrator of the iOS ecosystem.

KEYWORDS

iOS, mobile, obfuscation, reverse engineering, software protection

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

[Correction added on 14 November, after first online publication: the address of the first affiliation has been corrected and the ORCID of Dinghao Wu has been added].

and privacy loopholes in popular mobile software. For example, it was reported the piracy rates of popular mobile apps can reach the alarming level of 95%.[1] A study by Gibler et al indicated that a significant amount of mobile apps have fallen victim to software plagiarism.[2]

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,[3] 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 be applied to mobile apps. Additionally, obfuscation must not affect the regular development, functionality, distribution, and maintenance of mobile apps, which usually requires further customization to be made for the adopted obfuscation techniques.

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,[4-8] 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.

In our previous work,* we reported our experience of obfuscating multiple commercial iOS apps with millions of active users. To help mobile developers form a deeper understanding of software obfuscation and avoid common pitfalls that may appear when obfuscating iOS apps, we discussed 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.

Due to limited space, the discussion in our previous work did not fully cover the experiences we have gathered. Therefore, we extend the preliminary version of the article with the a rich amount of new contents, including the following

- In Section 6, we reported the review feedback received from Apple after the apps were obfuscated by the latest version of our obfuscator. We described the negative impacts caused by the obfuscation and explained how we addressed the issues. The new content can help iOS developers better understand the potential risks of applying obfuscation, even for the benign purpose of software protection. It can also provide developers with references about how the risks should be responded.
- In Section 4.2, we added more content about obfuscation algorithm selection based on the reviews received during the International Conference on Software Engineering submission. The picked algorithms were analyzed in the aspects of runtime cost, spatial cost, implementation complexity, etc. This would help developers who are interested in obfuscating their apps better understand the amount of effort required to achieve certain levels of protection.
- In Section 4.3, we discussed two additional obfuscator implementation pitfalls about redundant instrumentation code and app maintainability, respectively.

*A preliminary version[9] of this article appeared in the Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27 to June 3, 2018, Gothenburg.

- In Section 5, we added new discussions about the reason for the different evaluation results for 32-bit and 64-bit architectures.
- In Section 3, the introduction to theoretical obfuscation methods was enriched with details about indistinguishability obfuscation.
- In Section 8, the related work discussion was expanded to include relevant literature absent in the conference version due to limited space.

Same as its preliminary version, 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 this 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. We further describe our postobfuscation experience with handling the iOS vetting experience in Section 6. Section 7 discusses our prospect of iOS obfuscation and other protection methods, Section 8 reviews related work on obfuscation, and Section 9 concludes this paper.

2 | SOFTWARE OBFUSCATION

2.1 | Theoretical foundation

According to the formalization by Barak et al,[10] 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 nontrivial function property $\phi : \mathcal{F} \to \mathcal{S}$, where $\mathcal{S}$ is the set of all possible analysis results and any program analyzer, $\mathcal{A}_\phi$ that tries to efficiently compute ϕ; if $\phi(f)$ is intractably hard given only access to P as an oracle, the result of $\mathcal{A}_\phi(\mathcal{O}(P))$ is no better than randomly guessing $\phi(f)$. Essentially, the obfuscator in this notion is effectively turning a program into a virtual black box.

To better understand the aforementioned definition, 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 learn k in polynomial time 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.[10] That is, for certain programs, analyzing their code can always reveal at least 1-bit information about *what* the programs compute in time polynomial to program sizes, no matter how complicated the program code is rendered.

Immediately after the result on the impossibility of obfuscating programs into virtual black boxes was discovered, other formal notions of obfuscation were proposed in the hope of finding weaker forms of software protection that are still useful in certain scenarios. One of such notions is called *indistinguishability obfuscation*.[10] Given two equivalent programs, an indistinguishability obfuscator makes the obfuscated versions of the two programs computationally indistinguishable. Applications of such an obfuscator are not as straightforward as those of virtual black box obfuscation. Garg et al[11] described a case where software developers would like to release a demo version of their product. In contrast to a full version, certain functionalities of the demo need to be disabled. To strictly enforce this, code implementing these functionalities should be trimmed off, which can take a lot of additional effort. A convenient solution is to provide a restricted interface to these functionalities such that the code is still represent but cannot be executed by demo users. This is less secure since malicious users may be able to analyze the demo and reactivate the restricted functionalities by tampering with the restricted interfaces. However, if the demo is protected by an indistinguishability obfuscation technique, attackers will not be able to achieve this anymore. Suppose there are two versions of the demo with limited functionalities where D_1 is the version with the unwanted features completely removed from the code and D_2 is the version with the code of unwanted features preserved as unreachable dead code. Note that D_1 and D_2 are equivalent. Given the indistinguishability obfuscator iO, $iO(D_1)$ is indistinguishable from $iO(D_2)$. Since D_2 and therefore $iO(D_2)$ do not contain code implementing the trimmed features, there is no way for attackers to reactivate these features. Consequently, attackers cannot reactivate the features by tampering with $iO(D_2)$, either, even though the implementation of these

features indeed exist in $iO(D_2)$. Otherwise, attackers would successfully distinguish $iO(D_1)$ and $iO(D_2)$, which contradicts with the assumption that iO is an effective implementation of indistinguishability obfuscation.

Indistinguishability obfuscation has been proved to be realistic, ie, there are algorithms to implement it.[11] However, the current implementation is not usable due to extreme inefficiency. Experiments by Apon et al[12] showed that it takes a 32-core machine 11 minutes to evaluate a 16-bit point function[†] protected by indistinguishability obfuscation. Nevertheless, new efforts are constantly invested to explore obfuscation techniques that are both theoretically secure and practical. For example, Banescu et al proposed and released an open-source implementation of indistinguishability obfuscation with comprehensive benchmarks to measure its performance and size overhead.[13]

2.2 | Obfuscation in practice

Since theoretically sound solutions to the general problem of software obfuscation are either unfeasible or impractical, 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.[14] 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 runtime. 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[15] 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 runtime. 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.[16] 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.[17] 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. Therefore, the average cost of pirating a single mobile app could be extremely low.

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.[18] 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.[19] By developing information-stealing modules and

[†]An n-bit point function is a function over n-bit integers that returns 1 at a single point and returns 0 everywhere else.

FIGURE 1 Programmatically controlling massive iOS devices as a service (http://shemeitong.com/index.php/anli/show/46.html) [Colour figure can be viewed at wileyonlinelibrary.com]

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.

Frauds, spams, and malicious campaigns. Nowadays, many apps employ anomaly detection to identify suspicious client activities and prevent incidents like fraud, spam, and malicious campaigns. These malicious activities mostly occur within the communities of apps with social networking or information subscription services. Typical examples include sales of illegal products and promotions of fraudulent contents. The detection of these activities 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, 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,[20] the business of "click farming," which refers to the practice of producing massive fraudulent user actions[‡] for certain web and mobile services, has formed a billion-dollar underground economy, in which hundreds of well organized collusive groups have participated. The technical means of these campaigns are also evolving rapidly. 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. Figure 1 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.

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.

[‡]Typical profitable actions including clicking ads, sharing certain contents on social networks, and providing quick ratings for certain products on e-commerce platforms, etc. It is known that many major mobile service providers have been the victims of such malicious campaigns, including Google[21] and Uber.[22]

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,[23-28] 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 patterns of their binary code can be utterly uniform. This allows reverse engineering tools to make aggressive assumptions about the code they are analyzing and "guess" the high-level semantics of the code with specialized heuristics, thus sparing the need of fully reasoning about the program. Indeed, 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,[29] 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, eg, 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 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.

```
1  @implementation TSAnimatedAdapter
2  ...
3
4  - (BOOL)canPerformEditingAction:(SEL)action {
5      return (action == @selector(copy:)
6              || action == NSSelectorFromString(@"save:"));
7  }
8
9  ...
10 @end
```

(A)

```
1  // TSAnimatedAdapter - (bool)canPerformEditingAction:(SEL)
2  bool __cdecl -[TSAnimatedAdapter canPerformEditingAction:]
3  (struct TSAnimatedAdapter *self, SEL a2, SEL a3) {
4    bool result;
5    if ( "copy:" == a3 )
6      result = 1;
7    else
8      result = NSSelectorFromString(CFSTR("save:")) == (_QWORD)a3;
9    return result;
10 }
```

(B)

FIGURE 2 Decompiling an open-source iOS app[30] with IDA Pro [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 1 Partial list of obfuscated iOS apps

Rank	Category	# of Installations
1	Information, Reference	10 000 000 ∼ 50 000 000
2	Productivity	5 000 000 ∼ 10 000 000
3	Navigation, Travel	5 000 000 ∼ 10 000 000
4	Social	1 000 000 ∼ 5 000 000
5	Productivity	1 000 000 ∼ 5 000 000
6	Tools, Reference	500 000 ∼ 1 000 000
7	Entertainment, Life	100 000 ∼ 500 000
8	Tools, Reference	100 000 ∼ 500 000
9	Books, Education	100 000 ∼ 500 000
10	Entertainment, Life	100 000 ∼ 500 000
11	Food and Drink	100 000 ∼ 500 000
12	Food and Drink, Life	100 000 ∼ 500 000
13	Finance	100 000 ∼ 500 000
14	Finance	100 000 ∼ 500 000
15	Tools	100 000 ∼ 500 000

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.[31] 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 (IR), which is even less challenging to analyze than advanced reduced instruction set computing 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. Table 1 shows the detailed information of the top 15 apps ranked by the number of user installations. The apps are anonymized due to company requirements.

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.[§] Therefore, all source code in an iOS project is eventually translated into the LLVM 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 transformations. Compared with other options like source-level and binary-level obfuscation, the IR-level solution provides multiple appealing benefits.

- The IR obfuscation is mostly 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. The 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 swift compiler backend is based on a separately maintained LLVM version, thus slightly different from the standard one.

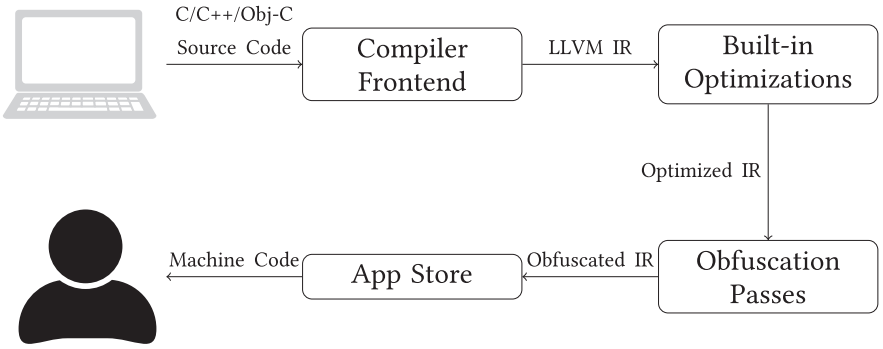

FIGURE 3 Overview of obfuscation workflow. IR, intermediate representation

LLVM is a highly configurable compiler infrastructure. It allows developers to provide their own program analysis and transformation algorithms, which can be combined with the built-in compiler optimizations. An analysis or transformation is called a "pass" in LLVM. Our obfuscator is essentially a series of passes, each of which implements an obfuscation transformation. During compilation, the obfuscation passes are scheduled after all other optimization passes of LLVM to make the effects of obfuscation more predictable. After all transformations are done, the obfuscated LLVM IR is serialized into its binary form (as known as bitcode), which is then submitted to the App Store. Once the bitcode is accepted, it will be translated to machine code on Apple's cloud and made available for users to download. This process is illustrated as Figure 3. At this point, we did not try to find an optimal schedule of the obfuscation passes to maximize the security effects. Nevertheless, this problem has been discussed by Liu et al.[32] The previously proposed method can be applied to our framework without significant modifications.

The current implementation of our obfuscator consists of about 3.8 K lines of C++ code,[¶] plus another 1 K 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 at the translation unit level and the function level through customized compiler flags and extended function attributes.

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 strictly prohibited on iOS, leaving dynamic code rewriting obfuscation unfeasible. For this reason, many packer-based obfuscation techniques that are popular on Android[33] 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.[34] 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,[35,36] which requires integrating a hardware emulator into the app. In some cases, the emulator itself could be more complicated than the code to be obfuscated.

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

[¶]Code statistics in this paper include comments and blanks.

TABLE 2 Cost-effectiveness characteristics of selected obfuscation algorithms

Obfuscation	Runtime cost	Spatial cost	Implementation complexity	Analysis to impede	Impact on maintenance
Symbol name mangling	Negligible	Negligible	Low	Human	Low
String literal encryption	One-time	Negligible	Low	Human	Negligible
Disassembly disruption	One-time	Scale with # of functions	High	Human & Automated	Negligible
Bogus control flow	Accumulative (Configurable)	Scale with # of basic blocks	High	Human & Automated	Low
Control flow flattening	Accumulative	Scale with # of basic blocks	High	Human & Automated	High
Garbage instruction	Accumulative (Configurable)	Scale with # of instructions	Low	Human & Automated	Low

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.[37#]

Considering the factors listed, 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.‖
2. *String literal encryption* that hides the plaintext of the string literals stored in the binary. The protected strings are decrypted at runtime.
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.**[38]
5. *Control flow flattening* that obscures the logic relations between program basic blocks.[39]
6. *Garbage instruction insertion* that injects garbage code that is irrelevant to program functionality.[40]

We present in Table 2 a summary of the characteristics of the selected algorithms, describing their runtime cost, spatial cost, implementation complexity, type of analysis aimming to impede, and negative impacts on app maintainability. We then elaborate on some of the information in Table 2 that we believe is important for readers to better understand why these algorithms fit our demand. We consider this table informative for developers, who are potentially interested in developing their own iOS obfuscators, to understand factors that may affect their decisions on which algorithms to pick according to their unique demands. In particular, while "implementation complexity" does not directly penalize obfuscated apps, it is indeed part of the cost that needs to be considered during obfuscator development, thus should be acknowledged.

Run-time cost. We consider two types of runtime cost, ie, one-time cost that is paid when apps start up and accumulative cost which constantly slows down the apps. For most obfuscations, the cost is accumulative, except for string literal encryption, which can decode all protected strings as apps start and leave them in their original encoding for the rest of the execution. Regarding obfuscations that insert additional executable code fragments into the apps, the cost depends on obfuscation intensity, ie, the ratio of the amount newly inserted code to that of the original code. When implementing those algorithms, we made the obfuscation intensity configurable by app developers so that they can make the trade-off between security and performance based on their own particular needs.

Spatial cost. For most obfuscation algorithms, the spatial cost is positively correlated to the original size and complexity of the code to be protected in terms of the number functions, basic blocks, and instructions. Similar to runtime cost, the spatial cost of obfuscation can also be controlled by adjusting the obfuscation intensity. It is important to note that the

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.

‖ 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. See Section 6 for more details.

** Opaque predicates are predicates that always evaluate to some constant Boolean value at runtime but hard to analyze statically. A typical example of opaque predicate is `x*(x+1)%2==0` whose value is `true` regardless of the value of x.

cost of some obfuscations can be boosted by others. For example, with more bogus control flow paths inserted, control flow flattening becomes more costly as well.

Implementation complexity. The implementation complexity of the obfuscation algorithms should be reviewed not only because developing an obfuscator itself can be a challenging software engineering task, but also because complicated obfuscation algorithms are error prone and can introduce subtle software bugs that are difficult to detect with the regular testing means provided for the apps, eg, the unittest cases. For some algorithms, the implementation complexity is due to the need of inventing a large number of new program constructs. For example, building bogus control flows relies on the availability of opaque predicates, while inserting garbage instructions require predefined patterns to ensure the inserted instructions do not alter program semantics. To make the implementation of these algorithm competent, we need to carefully design these constructs and constantly update them in new versions of the obfuscator, which takes a considerable amount of engineering effort. Some other algorithms do not rely on new program constructs, but require complex program transformations. In Table 2, we consider the implementation complexity to be *high* if the obfuscation requires deep domain knowledge about program analysis, program transformation, or reverse engineering. Considering the implementation complexity factor, we only selected three complicated obfuscation algorithms at this point.

Type of analysis to impede. 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. It is hard to predict what reverse engineering methods will be available to our adversaries, but, in general, both human and automated analysis should be taken into account. Among the obfuscations we picked, symbol name mangling and string literal encryption are mainly for misleading human perceptions while the others can additionally confuse most automated tools.

Negative impacts on app maintenance. It is common that 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. Obfuscation can render crash traces unreadable in two aspects. First, the symbol names appearing in the stack traces, especially the function names, are scrambled into strings meaningless to humans. Secondly, obfuscations like garbage instruction, bogus control flow, and control flow flattening inject additional code into the apps, which cannot be correlated to actual locations in the source files. Ideally, obfuscator-instrumented 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. In that case, crash reports will be confusing to developers unaware of the details of obfuscation. We consider three types of effort required to amend the postobfuscation crash reports. The impact on app maintenance is "negligible" for an obfuscation if text processing is sufficient to render crash reports readable again. The impact is "low" if new LLVM IR debug info needs to be inserted, and the impact is "high" if the debug info requires both insertion and rearrangement. Section 4.3.3 discusses the details about how we addressed those issues.

We ensure that all selected obfuscation algorithms well abide by Apple's security policies. According to a previous empirical study on obfuscated iOS apps found in the App Store, these algorithms or their variants have been previously employed by legit app developers,[41] indicating that they are unlikely to affect the review process. Regarding the concern about exceeding the binary size limit, the first three algorithms of the selected ones (ie, symbol name mangling, string literal encryption, and disassembly disruption) barely introduce 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, eg, 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 disassembly disruption, 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.[42-44] We intentionally implemented mutant versions of the previously proposed obfuscation algorithms by introducing new minor features and randomizing the patterns of the generated binary code. As a consequence, attackers will need more sophisticated techniques to nullify the mutated obfuscation effects.[45]

```
; @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 intermediate representation inline assembly [Colour figure can be viewed at wileyonlinelibrary.com]

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 these issues may not be of significant concern at the first glance, but can subtly affect the potency and practicality of our work if not properly addressed. 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 advanced reduced instruction set computing machine (ARM) architecture. Compared with the complex instruction set computer architectures x86 and x64, where binary-level obfuscation is quite prevalent, ARM is reduced instruction set computer and employs the fixed-length instruction encoding. This invalidates many obfuscation techniques that exploit the variable-length encoding of instructions, such as overlapping instructions.[46]

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, even though the 32-bit platforms are much less popular than the 64-bit ones. 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

To undo the negative impact of symbol name mangling on crash analysis, the obfuscator needs 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. To achieve this, we generate the mapping before compilation and feed the mapping as auxiliary information to the compiler. Obfuscation passes in the compiler will scramble the symbol names as directed by this predefined mapping.

To tackle the problem of inconsistent code and debug information, 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.

4.3.4 | Redundant instrumentation code and data

For many obfuscation methods, it is necessary to instrument the code to be protected with additional helper routines. For example, when applying string literal encryption, a decryption routine has to be injected and scheduled to run before the obfuscated strings are used.

Typically, to avoid symbol name clashes, the linkage of functions and global variables inserted by the compiler should be set as "private" and invisible to other translation units, which is equivalent to declaring and defining the symbols to be `static` in the C or C++ source code. This, however, could lead to unnecessarily bloated code, since oftentimes it is sufficient to keep a single copy of the helper code and data in the entire program instead of keeping one copy in each obfuscated translation unit.

Our solution to this problem is to set the linkage of inserted code and data as "weak." In this way, the linker will ensure that only one copy of the definitions to the same symbol will be preserved in the final executable. Indeed, this requires the obfuscator to ensure that names of inserted symbols are exotic enough to avoid any possible clash with developer-defined functions and variables. Typically, this will not be a problem, since both LLVM IR and the linker for iOS accept all ASCII strings as symbol names, many of which are not legal identifiers in most programming languages.

4.3.5 | Releasing obfuscated code as libraries

As previously mentioned, Apple now advises developers to submit their apps to the store in the form of LLVM IR.[47] The apps will be compiled on Apple's cloud to native code of different targets. In this way, Apple can reoptimize the app binary when there is an update for the compilation toolchain, without developers needing to upload a new binary themselves. By the time of writing, uploading apps by IR is still optional for iOS apps, but mandatory for watchOS and tvOS apps.

Since our obfuscation is performed at the IR level, all obfuscation effects should be retained even if the final binary code is generated by Apple. However, there is a new security concern to developers who make iOS libraries for other apps. Submitting an app as IR requires all dependencies of that app, ie, libraries the app links against, to be available in the IR form. As more and more app developers are following the new practice of distribution, third-party library developers have to switch to IR release if they want to maximize their user base. From a security point of view, however, IR code is more "vulnerable" than machine code for being substantially easier to reverse engineer, just like source code is easier to analyze than machine code. This is against the goal of heuristic obfuscation which aims to make reverse engineering as uneconomical as possible.

At this point, the code base we obfuscated is only for internal use. Therefore, the IR form of obfuscated code is never revealed to outsiders. As such, we have not developed a method to mitigate the problem. However, we believe this is worth considering in the long run. We have a dedicated discussion on this topic in Section 7.

5 | EVALUATION

We now report the outcome of our obfuscation effort. The protected iOS code base consists of 23 K lines of Objective-C and C code, which roughly takes 0.5% to 2% of each including app. Note that the proportion is low because the scale of apps including the obfuscated code base is extremely large. Most of those apps contain millions of lines of code.

We evaluated the obfuscation in two aspects, ie, resilience and overhead. According to the definition by Collberg et al,[38] 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. For cryptographic obfuscation, the evaluation methodology is clear, ie, reducing deobfuscation to a computational problem with provable or conjectural intractability.[48] In contrast, evaluation through empirical experiments always raises 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,[49-51] 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,[49,52] the early steps are fairly standard. Typically, the testers will first use automated reverse engineering tools to transform 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 3 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.

Notably, the difference between results of the two architectures is significant. To understand the cause, we manually analyzed the binaries within IDA Pro and found that it is related to the heuristics used by IDA Pro to identify constant pools and infer whether the constants are memory addresses. Since IDA Pro uses a recursive disassembler, any newly recognized memory addresses lead to additional code areas to be disassembled. The 32-bit iOS binaries use the unaligned THUMB instruction encoding and has a smaller address space. Therefore, there is a higher probability for the value of a 4-byte data chunk to be a valid code pointer value. IDA Pro falsely identified a lot more memory addresses than it did for the 64-bit binary, thus leading to a much higher disassembly rate.

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 3 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.

TABLE 3 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

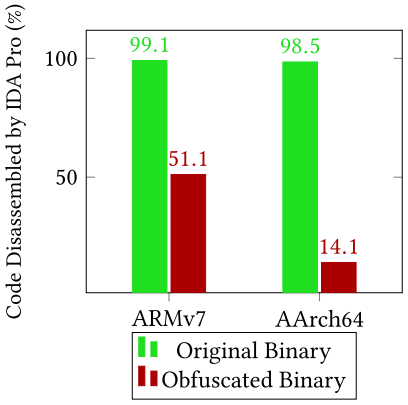

FIGURE 5 Effectiveness of disassembly disruption [Colour figure can be viewed at wileyonlinelibrary.com]

There exist more advanced deobfuscation techniques. For example, AntiProGuard is a tool that analyzes Android functions whose names have been scrambled and try to guess their semantics by matching them with a labeled code database[53]; Udupa et al proposed to detect and remove control flow flattening[54]; Sharif et al,[35] Coogan et al,[36] and Udupa et al,[54] developed methods to unpack code protected by virtualization-based obfuscation. Indeed, some of our obfuscation algorithms are potentially vulnerable to some of these deobfuscation techniques, if they are revised to take our modifications to the obfuscations into account. As we have introduced in Section 2.2, most practically usable obfuscation algorithms are not theoretically safe and there is always a chance that each of them can be cracked by a particular countermeasure. This is why we consider developing and applying multiple obfuscations together. To fully deobfuscate the code, attackers will have to nullify every single layer of deployed protections, which greatly raises the bar of successful reverse engineering. Nevertheless, it is typically believed that no practical obfuscation can withstand extensive reverse engineering effort. We therefore discuss the possibility of combining obfuscation with other types of software protection methods to achieve more comprehensive defense. See Section 7.2 for details.

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 only functionality of this new app is to sequentially run the unittests for the obfuscated code base. During compilation, all obfuscation algorithms were enabled for every suitable translation unit and function.[††] 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.

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 4 shows the code segment sizes of the original and obfuscated iOS code. Only the expansion of code sections was considered by the measurement. 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 aforementioned, 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 reduced instruction set computer and has a limited instruction length, loading a large constant integer into a register usually takes more than one instruction to accomplish. The 32-bit ARM backend of LLVM materializes relocatable memory addresses by employing constant pools, whereas the 64-bit backend uses dedicated instructions like `adrp`. Constant pools are placed in the code section, thus contributing to the size overhead. Overall, the 32-bit backend requires slightly more bytes than the 64-bit backend to load a code block address into a register. See Figure 6 for an illustration. Since our obfuscator emits a large amount of code that refers to the memory addresses of nonentry basic blocks, the difference between the size efficiency of the two backends is amplified, leading to different size expansion rates. This may further affect the behavior of compiler backend when it decides whether or not to inline a function at link time, but we have not confirmed this since it requires diving into the implementation details of LLVM. Once we can fully understand the causes, it may be possible to improve the size efficiency of the obfuscation for 32-bit binaries in the future.

[††]Compilation and obfuscation settings in the experiments are identical to those for actual development and deployment of the apps.

TABLE 4 Binary size expansion due to obfuscation

Target Architecture	Code Segment Size in Bytes		
	Original	Obfuscated	Increase
ARMv7 (32 bit)	286 304	1 070 656	784 352 (+ 307%)
AArch64 (64 bit)	333 376	1 165 456	832 080 (+ 221%)

```
0x0d9066: df f8 78 09  ldr.w r0, [pc, #0x978]
0x0d906a: 78 44        add   r0, pc        ; load address 0xd9074
...
0x0d99e0: 16 00 00 00  (32-bit data: 0x16)
```

(A)

```
0x0730ac: 80 00 00 90  adrp  x8, #0x73000
0x0730b0: 08 01 03 91  add   x8, x8, #0xc0 ; load address 0x730c0
```

(B)

FIGURE 6 Code sequences emitted for loading relocatable memory addresses into registers [Colour figure can be viewed at wileyonlinelibrary.com]

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 1 GB RAM. The obfuscated code performs both synchronous and asynchronous tasks inside host apps. Each task was executed for 10 times and the performance is measured by the built-in debugger and profiler of Xcode, which is the official iOS app development environment. 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,[‡‡] 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 | IMPACTS ON PUBLISHER REVIEW

One of the most important principles that direct our design and implementation of the obfuscator is that the obfuscation should not have negative influence on the vetting process of obfuscated apps. After finishing the initial development of the obfuscator, we have been closely tracking the submission process of apps containing our obfuscated code. In this section, we report the details of postobfuscation app reviews, including the problems encountered and how we addressed them.

6.1 | Basics of app review

Getting the approval for an app to be published into the App Store can be an error-prone procedure. The review takes many factors into consideration, including security, digital rights, objectionable content, monetization, and overall app quality. An app can easily get rejected if the developers are careless enough to violate any of the guidelines documented by Apple.

The time needed for a submission to receive a decision varies. Although there is no official description about how exactly decisions are made, it is likely that both automated and human inspections are required. The availability of human reviewers are unpredictable. The first submission of an app typically takes 5 to 10 days to get a decision, while reviews for minor version updates can be much faster. However, once a submission is rejected, subsequent reviews can take significantly longer.

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

In case of a rejection, Apple will provide reasons. However, the first batch of feedback can be very brief and sometimes confusing. Developers can request for more details if they find the reasons attached to the rejection are not clear.

6.2 | Postobfuscation review feedback

The first working version of our obfuscator was delivered in August 2016. Since then, we have been subimitting iOS applications with obfuscation applied. Until August 2017, we did not notice any significant delay for Apple to accept our app updates, nor did our apps get rejected due to the inclusion of obfuscated code. However, starting from September 2017, three of our apps got rejected successively. The initial feedback indicated that the rejected apps "contain hidden or undocumented features," which is unclear to the development team. Therefore, the submitters requested additional details about the rejections and forwarded them to us. In the enriched comments, reviewers from Apple expressed their concerns about the Objective-C functions whose names are obfuscated. The reviewers further suggested that it would be appropriate to remove all symbol renaming obfuscations.

Review feedback indicates that obfuscating function names is indeed an effective way to hinder human binary analysis. However, getting approvals is usually the top priority to app developers and obfuscation must not come with the cost of delaying app updates. Therefore, we immediately disabled symbol name mangling and retreated to the original function names. We then resubmitted the updated apps and got approved shortly after.

6.3 | Implications

Our app submission experience suggests that obfuscation techniques are not only faced with challenges from the technical side but are also constrained by the current centralized distribution model for mobile software. 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, developers will have to carefully take the balance themselves. We discuss this dilemma in more details in Section 7.1.

Another finding we obtained through the failed submissions is that, even without considering the advancement of reverse engineering techniques, employing obfuscation in practical mobile app development requires long-term efforts and constant investment. According to a recent empirical study on obfuscated mobile software,[41] iOS apps submitted between January and October in 2016 were still allowed for symbol name mangling, which is consistent with our experience, while the situation has now substantially changed and led to the rejection discussed above. As such, we speculate that Apple has been updating its internal guidelines about how transparent an app should be regarding the vetting process. A previously allowed obfuscation is not guaranteed to remain valid in the future. Hence, the design and implementation of an obfuscator should keep up with distributor policies to ensure seamless updates of the protected apps.

7 | DISCUSSION

7.1 | 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.

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 postreview 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 toward more effective iOS software protection.

7.2 | Other protections

Obfuscation is not a panacea for combating the security threats targeting mobile apps and there have been many deobfuscation techniques proposed.[36,55-57] 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 runtime to circumvent certain static protections provided by obfuscation. Reverse engineering tools like cycript[58] and Frida[59] 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 runtime, 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.[60]

Various techniques are available for preventing software from being dynamically debugged by unauthorized parties. However, antidebugging 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 antidebugging facilities before dynamically inspecting them.

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

8 | 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.[10,61] It is however feasible to build secure obfuscation constructs with limited generality, such as indistinguishability obfuscation[10] for polynomial-sized circuits[11] and the best possible obfuscation that tries to minimize rather than eliminate information leakage.[62] In addition to obfuscation algorithms, researchers have been proposing and implementation hardware tokens that can securely execute obfuscation programs such that even the low-level program behavior like memory access does not leak additional information. Ascend,[63] GhostRider,[64] Phantom,[65] and Raccoon[66] are implementations of secure processors or hardware-software systems that achieves achieves memory obliviousness when executing obfuscated programs. HOP[48] is another secure processor that provides oblivious memory access; moreover, it is resilient to rewinding attacks by making the processor stateless.

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[67] proposed to replace branches with exceptions and reroute the control flows with customized exception handlers. Mimimorphism[68] 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[69] employed the architectural support of Itanium processors for information flow tracking, ie, utilizing the deferred exception tokens embedded in Itanium registers to encode opaque predicates. Domas[15] developed a compiler, which generates a binary employing only the mov family instructions, based on the fact that x86 mov is Turing complete. Wang et al,[50]

Lan et al,[70] and Wang et al[71] 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.[72-75] Liu et al used a method based on stochastic processes to make JavaScript source code more perplexing.[32] Hindle et al suggested to measure the perplexity of obfuscated source code with statistical language models.[76]

Most practically usable obfuscation tools supporting iOS are commercial. Obfuscator-LLVM[44] 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[77] is a source-level obfuscator supporting the C programming language. Protections provided by Tigress are mostly heavy weight, eg, virtualization-based obfuscation and self-modification, some of which are not applicable to iOS apps. In addition to obfuscation frameworks, there are also open-source implementations of individual obfuscation algorithms for iOS, such as string literal encryption[43] and identifier name mangling.[42]

9 | CONCLUSION

In this paper, we have 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 large-scale apps with humongous 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.

ACKNOWLEDGEMENTS

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.

ORCID

Pei Wang http://orcid.org/0000-0001-9066-3948
Dinghao Wu https://orcid.org/0000-0002-0741-5511

REFERENCES

1. Monument Valley apparently has a 95% piracy rate on Android, 60% on iOS. https://goo.gl/TkfCIK
2. Gibler C, Stevens R, Crussell J, Chen H, Zang H, 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); 2013; Taiwan.
3. Chen H, He D, Zhu S, Yang J. Toward detecting collusive ranking manipulation attackers in mobile app markets. In: Proceedings of the 12th ACM Asia Conference on Computer and Communications Security (ASIACCS); 2017; Abu Dhabi, United Arab Emirates.
4. Zhou Y, Jiang X. Dissecting Android malware: characterization and evolution. In: Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P); 2012; San Francisco, CA.
5. Linares-Vásquez M, Holtzhauer A, Bernal-Cárdenas C, 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 (MSR); 2014; Hyderabad, India.
6. Wang H, Guo Y, Ma Z, 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 (ISSTA); 2015; Baltimore, MD.
7. Glanz L, Amann S, Eichberg M, et al. CodeMatch: obfuscation won't conceal your repackaged app. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE); 2017; Paderborn, Germany.
8. Li M, Wang W, Wang P, et al. LibD: scalable and precise third-party library detection in Android markets. In: Proceedings of the 39th ACM/IEEE International Conference on Software Engineering (ICSE); 2017; Buenos Aires, Argentina.

9. Wang P, Wu D, Chen Z, 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 Track, ICSE Companion; 2018; Gothenburg, Sweden.
10. Barak B, Goldreich O, Impagliazzo R, et al. On the (im)possibility of obfuscating programs. *J ACM*. 2012;59(2). Article No 6.
11. Garg S, Gentry C, Halevi S, Raykova M, Sahai A, Waters B. Candidate indistinguishability obfuscation and functional encryption for all circuits. In: Proceedings of the 54th IEEE Annual Symposium on Foundations of Computer Science (FOCS); 2013; Berkeley, CA.
12. Apon D, Huang Y, Katz J, Malozemoff AJ. Implementing cryptographic program obfuscation. Cryptology ePrint Archive: Report 2014/779. 2014. https://eprint.iacr.org/2014/779
13. Banescu S, Ochoa M, Kunze N, Pretschner A. Idea: benchmarking indistinguishability obfuscation–a candidate implementation. In: *Engineering Secure Software and Systems: 7th International Symposium, ESSoS 2015, Milan, Italy, March 4-6, 2015. Proceedings*. Cham, Switzerland: Springer International Publishing; 2015.
14. Schrittwieser S, Katzenbeisser S, Kinder J, Merzdovnik G, Weippl E. Protecting software through obfuscation: can it keep pace with progress in code analysis? *ACM Comput Surv*. 2016;49(1). Article No 4.
15. Domas C. Turning 'mov' into a soul-curshing RE nightmare. In: Proceeding of the 2015 Annual Reverse Engineering and Security Conference (REcon); 2015.
16. Prakash A. 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
17. Unlicensed software use still high globally despite costly cybersecurity threats. Washington, DC: BSA Worldwide. http://globalstudy.bsa.org/2016/index.html
18. Hubbard J, Weimer K, 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 (CCNC); 2014; Las Vegas, NV.
19. Chen Z. iOS masque attack weaponized: a real world look. Milpitas, CA: FireEye. https://www.fireeye.com/blog/threat-research/2015/08/ios_masque_attackwe.html
20. Black mirror investigation: a report on the bussiness of "click-farming". http://image.3001.net/uploads/pdf/4aa87c46888173995c295a873c2aa682.pdf
21. The flourishing business of fake youtube views. https://www.nytimes.com/interactive/2018/08/11/technology/youtube-fake-view-sellers.html
22. Uber fights off scammers every day. here's how it learned the tricks. https://www.cnet.com/news/uber-fights-off-scammers-every-day-heres-how-it-learned-the-tricks/
23. Sagiv M, Reps T, Wilhelm R. Solving shape-analysis problems in languages with destructive updating. *ACM Trans Program Lang Syst*. 1998;20(1):1-50.
24. Balakrishnan G, Reps T. Analyzing memory accesses in x86 executables. In: *Compiler Construction: 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2004.
25. Brumley D, Jager I, Avgerinos T, Schwartz EJ. BAP: a binary analysis platform. In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2011.
26. Schwartz EJ, Lee J, Woo M, Brumley D. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In: Proceedings of the 22nd USENIX Security Symposium; 2013; Washington, DC.
27. Wang S, Wang P, Wu D. Reassembleable disassembling. In: Proceedings of the 24th USENIX Security Symposium, USENIX Security; 2015; Washington, DC.
28. Bauman E, Lin Z, Hamlen KW. Superset disassembly: statically rewriting x86 binaries without heuristics. In: Proceedings of the 25th Annual Network and Distributed Systems Security Symposium (NDSS); 2018; San Diego, CA.
29. IDA: About. Liège, Belgium: Hex-Rays. https://www.hex-rays.com/products/ida/
30. Github - WhisperSystems/Signal-iOS: a private messenger for iOS. San Francisco, CA: GitHub Inc. https://github.com/WhisperSystems/Signal-iOS
31. Shrink your code and resources | Android studio - Android developers. https://developer.android.com/studio/build/shrink-code.html
32. Liu H, Sun C, Su Z, Jiang Y, Gu M, Sun J. Stochastic optimization of program obfuscation. In: Proceedings of the 39th International Conference on Software Engineering (ICSE); 2017; Buenos Aires, Argentina.
33. Yang W, Zhang Y, Li J, et al. AppSpear: bytecode decrypting and DEX reassembling for packed Android malware. In: *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan,November 2-4, 2015. Proceedings*. Cham, Switzerland: Springer International Publishing; 2015.
34. Maximum build file sizes. Cupertino, CA: Apple Inc. https://help.apple.com/itunes-connect/developer/#/dev611e0a21f
35. Sharif M, Lanzi A, Giffin J, Lee W. Automatic reverse engineering of malware emulators. In: Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P); 2009; Berkeley, CA.
36. Coogan K, Lu G, Debray S. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS); 2011; Chicago, IL.
37. iTunes connect developer guide. Cupertino, CA: Apple Inc. https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/About.html
38. Collberg C, Thomborson C, Low D. Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL); 1998; San Diego, CA.

39. Chow S, Gu Y, Johnson H, Zakharov VA. An approach to the obfuscation of control-flow of sequential computer programs. In: *Information Security: 4th International Conference, ISC 2001 Malaga, Spain, October 1-3, 2001 Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2001.
40. Cohen FB. Operating system protection through program evolution. *Comput Secur*. 1993;12(6):565-584.
41. Wang P, Bao Q, Wang L, et al. Software protection on the go: a large-scale empirical study on mobile app obfuscation. In: Proceedings of the 40th International Conference on Software Engineering (ICSE); 2018; Gothenburg, Sweden.
42. Github - preemptive/PPiOS-rename: Symbol obfuscator for iOS apps. San Francisco, CA: GitHub Inc. https://github.com/preemptive/PPiOS-Rename
43. Github - pjebs/obfuscator-iOS: Secure your app by obfuscating all the hard-coded security-sensitive strings. San Francisco, CA: GitHub Inc. https://github.com/pjebs/Obfuscator-iOS
44. Junod P, Rinaldini J, Wehrli J, Michielin J. Obfuscator-LLVM -- software protection for the masses. In: Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO); 2015; Florence, Italy.
45. Xu D, Ming J, Wu D. Generalized dynamic opaque predicates: a new control flow obfuscation method. In: *Information Security: 19th International Conference, ISC 2016, Honolulu, HI, USA, September 3-6, 2016. Proceedings*. Cham, Switzerland: Springer International Publishing; 2016.
46. Bonfante G, Fernandez J, Marion J-Y, Rouxel B, Sabatier F, 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 (CCS); 2015; Denver, CO.
47. What is app thinning? (iOS, tvOS, watchOS). Cupertino, CA: Apple Inc. https://help.apple.com/xcode/mac/current/#/devbbdc5ce4f
48. Nayak K, Fletcher CW, Ren L, et al. HOP: hardware makes obfuscation practical. In: Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS); 2017; San Diego, CA.
49. Banescu S, Ochoa M, Pretschner A. A framework for measuring software obfuscation resilience against automated attacks. In: Proceedings of the 1st International Workshop on Software Protection (SPRO); 2015; Florence, Italy.
50. Wang P, Wang S, Ming J, Jiang Y, Wu D. Translingual obfuscation. In: Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P); 2016; Saarbrücken, Germany.
51. Pawlowski A, Contag M, Holz T. Probfuscation: an obfuscation approach using probabilistic control flows. In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*. Cham, Switzerland: Springer International Publishing; 2016.
52. Ceccato M, Di Penta M, Falcarin P, Ricca F, Torchiano M, Tonella P. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empir Softw Eng*. 2014;19(4):1040-1074.
53. Baumann R, Protsenko M, Müller T. Anti-proguard: towards automated deobfuscation of Android apps. In: Proceedings of the 4th Workshop on Security in Highly Connected IT Systems (SHCIS); 2017; Neuchâtel, Switzerland.
54. Udupa SK, Debray SK, Madou M. Deobfuscation: reverse engineering obfuscated code. In: Proceedings of the 12th Working Conference on Reverse Engineering (WCRE); 2005; Pittsburgh, PA.
55. Xue L, Luo X, Yu L, Wang S, Wu D. Adaptive unpacking of Android apps. In: Proceedings of the 39th International Conference on Software Engineering (ICSE); 2017; Buenos Aires, Argentina.
56. Xu D, Ming J, 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; San Jose, CA.
57. Ming J, Xu D, Wang L, 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 (CCS); 2015; Denver, CO.
58. Cycript. http://www.cycript.org/
59. Frida · a world-class dynamic instrumentation framework. www.frida.re/
60. Dave V, Guha S, Zhang Y. Measuring and fingerprinting click-spam in ad networks. *ACM SIGCOMM Comput Commun Rev*. 2012;42(4):175-186.
61. Goldwasser S, Kalai YT. On the impossibility of obfuscation with auxiliary input. In: Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS); 2005; Pittsburgh, PA.
62. Goldwasser S, Rothblum GN. On best-possible obfuscation. In: *Theory of Cryptography: 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2007.
63. Fletcher CW, Dijk MV, Devadas S. A secure processor architecture for encrypted computation on untrusted programs. In: Proceedings of the 7th ACM Workshop on Scalable Trusted Computing (STC); 2012; Raleigh, NC.
64. Liu C, Harris A, Maas M, Hicks M, Tiwari M, Shi E. Ghostrider: a hardware-software system for memory trace oblivious computation. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS); 2015; Istanbul, Turkey.
65. Maas M, Love E, Stefanov E, et al. PHANTOM: practical oblivious computation in a secure processor. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, (CCS); 2013; Berlin, Germany.
66. Rane A, Lin C, Tiwari M. Raccoon: closing digital side-channels through obfuscated execution. In: Proceedings of the 24th USENIX Security Symposium, USENIX Security; 2015; Washington DC.
67. Popov IV, Debray SK, Andrews GR. Binary obfuscation using signals. In: Proceedings of 16th USENIX Security Symposium, USENIX Security; 2007; Boston, MA.

68. Wu Z, Gianvecchio S, Xie M, Wang H. Mimimorphism: a new approach to binary code obfuscation. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS); 2010; Chicago, IL.
69. Chen H, Yuan L, Wu X, Zang B, Huang B, Yew P-C. Control flow obfuscation with information flow tracking. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO); 2009; New York, NY.
70. Lan P, Wang P, Wang S, Wu D. Lambda obfuscation. In: *Security and Privacy in Communication Networks: 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22-25, 2017, Proceedings*. Cham, Switzerland: Springer; 2017.
71. Wang Y, Wang S, Wang P, Wu D. Turing obfuscation. In: *Security and Privacy in Communication Networks: 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22-25, 2017, Proceedings*. Cham, Switzerland: Springer; 2017.
72. Guo F, Ferrie P, Chiueh T-C. A study of the packer problem and its solutions. In: *Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2008:98-115.
73. Martignoni L, Christodorescu M, Jha S. OmniUnpack: fast, generic, and safe unpacking of malware. In: Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC); 2007; Miami Beach, FL.
74. Royal P, Halpin M, Dagon D, Edmonds R, Lee W. PolyUnpack: automating the hidden-code extraction of unpack-executing malware. In: Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC); 2006; Miami Beach, FL.
75. Zhang F, Huang H, Zhu S, Wu D, Liu P. ViewDroid: towards obfuscation-resilient mobile application repackaging detection. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks (WiSec); 2014; Oxford, UK.
76. Hindle A, Barr ET, Su Z, Gabel M, Devanbu P. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering (ICSE); 2012; Zürich, Switzerland.
77. The tigress c diversifier/obfuscator. http://tigress.cs.arizona.edu/

How to cite this article: Wang P, Wu D, Chen Z, Wei T. Field experience with obfuscating million-user iOS apps in large enterprise mobile development. *Softw: Pract Exper*. 2018;1–22. https://doi.org/10.1002/spe.2648