Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation

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

Qinkun Bao
qub14@ist.psu.edu
The Pennsylvania State University

Li Wang
lzw158@ist.psu.edu
The Pennsylvania State University

Shuai Wang
szw175@ist.psu.edu
The Pennsylvania State University

Zhaofeng Chen
chenzhaofeng@baidu.com
Baidu X-Lab

Tao Wei
lenx@baidu.com
Baidu X-Lab

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

ABSTRACT

The prosperity of smartphone markets has raised new concerns about software security on mobile platforms, leading to a growing demand for effective software obfuscation techniques. Due to various differences between the mobile and desktop ecosystems, obfuscation faces both technical and non-technical challenges when applied to mobile software. Although there have been quite a few software security solution providers launching their mobile app obfuscation services, it is yet unclear how real-world mobile developers perform obfuscation as part of their software engineering practices.

Our research takes a first step to systematically studying the deployment of software obfuscation techniques in mobile software development. With the help of an automated but coarse-grained method, we computed the likelihood of an app being obfuscated for over a million app samples crawled from Apple App Store. We then inspected the top 6600 instances and managed to identify 601 obfuscated versions of 539 iOS apps. By analyzing this sample set with extensive manual effort, we made various observations that reveal the status quo of mobile obfuscation in the real world, providing insights into understanding and improving the situation of software protection on mobile platforms.

CCS CONCEPTS

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

KEYWORDS

obfuscation, reverse engineering, mobile app, empirical study

*Part of the research was done during an internship at Baidu X-Lab.

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

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180169

ACM Reference Format:
Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. 2018. Software Protection on the Go: A Large-Scale Empirical Study on Mobile App Obfuscation. In *ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180169

1 INTRODUCTION

Concerns on security breaches targeting mobile apps have kept rising in past years. It was reported the piracy rates of popular mobile apps can approach to 60–95% [7]. Research by Gibler et al. found that a surprisingly large portion of mobile applications are "copies" of others [19]. Besides these traditional intellectual property theft problems, the industry is also facing new security threats as there are now many businesses heavily relying on mobile devices to operate, among which the fraudulent and malicious campaigns conducted through automatically manipulating a massive number of mobile devices [14] are particularly harmful to the mobile ecosystems. From a technical point of view, reverse engineering mobile apps in general takes less effort than traditional desktop software, due to the wide use of reflective programming languages like Java and Objective-C and the regulated binary structures restricted by the mobile hardware and software environments. The soaring of unprecedented security challenges and the lack of natural defenses call have driven mobile developers to seek additional protections.

One of the most important software protection technologies is software obfuscation, which is a kind of semantics-preserving program transformations that aim to make software code more difficult for automated tools and humans to analyze. Although obfuscation-related research topics have been intensively studied for decades, most previous work focuses on in-lab technical analysis of the effectiveness of new obfuscation techniques [22, 35, 40, 42, 43] or countermeasures against obfuscation when it is misused by malware writers [13, 45]. As far as we have learned, little emphasis is put on investigating how benign software authors take obfuscation as part of their development process in the real world, which is critical for software obfuscation techniques to be practical. To push this line of research forward, we aim to investigate the answers to the following important research questions: **RQ1:** *What are the characteristics of obfuscated mobile apps?*; **RQ2:** *In what patterns are mobile apps typically obfuscated?*; **RQ3:** *How does app review*

affect the adoption of obfuscation?; and **RQ4:** *How resilient are the obfuscated apps to malicious reverse engineering?*

To develop meaningful conclusions, it is most adequate to conduct an empirical study on a reasonably large set of recently developed and supposedly benign mobile apps obfuscated by their vendors. Unfortunately, there is no such a data set available for public access, so we decided to collect samples independently. There are currently two major platforms in mobile software markets, i.e., iOS and Android. Although they share many common characteristics, there are also notable differences. Some previous research has indirectly or implicitly touched the topic of mobile app obfuscation, but the focus is mostly on Android. For example, the study by Zhou and Jiang on Android malware revealed some obfuscated samples [47]. Linares-Vásquez et al. [26] and Glanz et al. [20] investigated Android app repacking, with the potential disturbance of obfuscation considered. On the other hand, the iOS platform received notably less attention which mismatches its share in the market. With over a billion iOS mobile devices sold, there are reportedly millions of software programmers working on iOS app development. In this study, we chose to work on iOS for a dual purpose of filling in the blank of empirical studies on mobile app obfuscation and enriching scientific research on this important mobile platform.

To obtain a representative sample set, we crawled 1, 145, 582 free iOS app instances from the official Apple App Store. We then estimated the likelihood of each instance being obfuscated based on a variant of a statistical language model previously proposed for studying software source code [21, 27]. We picked the top 6600 most likely obfuscated samples and identified 539 that are truly obfuscated with manual verification. For each sample, we further conducted in-depth investigations to understand how obfuscation was applied. In general, effectively analyzing a large amount of obfuscated binary code can be extremely difficult, since most existing program analysis techniques have either scalability or accuracy issues regarding obfuscated code. Moreover, analyzing iOS apps has its own unique challenges, one of which is caused by the wide use of *statically linked third-party libraries* [16]. To overcome these obstacles altogether, our study combined automated analysis with a considerable amount of manual effort from knowledgeable software reverse engineers with industry experience. After examining all the samples, we formulated 8 findings regarding the proposed research questions.

In summary, we made the following contributions in this research:

- We are the first to conduct a comprehensive empirical study targeting mobile software obfuscation. Our research focuses on iOS, an influential mobile platform that did not receive enough attention from the academia in contrast to Android.
- We developed a scalable detection algorithm to estimate the likelihood of an iOS app being obfuscated and applied it to a large quantity of apps crawled from App Store. After manually analyzing the 6600 most likely obfuscated instances, we identified 539 truly obfuscated iOS apps with a total of 601 different versions. As far as we know, this is the first scientifically collected sample set of obfuscated iOS mobile apps. We plan to share these samples with the community in the future.
- To overcome the limitations of existing automated software analysis on obfuscated binaries, we invested over 600 manhours in manually examining the obfuscated iOS apps, extracting detailed information about how these apps are protected by different obfuscation algorithms. The human effort assured the accuracy of our analysis and therefore the credibility of our findings.
- We made various observations about the characteristics of obfuscated apps, the obfuscation patterns applied, and their resilience to reverse engineering. Our findings can shed light on future research on mobile software protection.

2 BACKGROUND

2.1 Significance of the Problem

Obfuscation is one of the most important software protection techniques that prevent software from being reverse engineered maliciously. The status of its application and presence among published software is closely related to the state of security in a software ecosystem. Previous research on mobile software engineering revealed that obfuscation has been a common practice on Android [24, 26, 47], yet the figure for iOS is mostly missing.

Since iOS is typically considered a more secure system than Android for being more closed, it may be susceptible that obfuscation on iOS could be as prevalent as on Android. However, some recent security incidents have shown that with the help of production-quality binary analysis tools like IDA Pro [2], iOS reverse engineering is not as difficult as it is generally recognized. For example, it is found that iOS developers similarly suffer from severe software piracy issues like Android developers [7]. It is also reported that there have been popular iOS apps being repackaged with malicious payload for stealing sensitive user data [17]. To help iOS developers counter these threats, some reputed software security solution providers have launched their iOS app obfuscation services [9].

For more secure iOS software engineering, it is imperative to obtain a thorough understanding about the current practice of applying obfuscation in iOS app development. The benefits of such an understanding are two-fold: vendors of obfuscation tools can better tune their development based on the status quo, while researchers interested in analyzing iOS app repositories can grasp a sense about when and how obfuscation may affect their analysis.

2.2 Technical Challenges of the Study

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

2.2.1 Obfuscation Detection and Analysis. Detecting and analyzing obfuscated binaries has long been an open research problem and is still being actively studied [12, 29, 36]. To date, the accuracy of automated obfuscation detection is not satisfying enough to fit our demand. Therefore, we decided to undertake manual analysis as the major research methodology of the study, with some light-weight automated methods as assistance. Unlike Android developers that can use an app obfuscator embedded into the official development toolchain [8], iOS developers do not get any receive official support,

thus having to rely on third-party tools or self-made obfuscators. Considering the large number of obfuscation techniques potentially available, it is impractical for an empirical study relying on manual effort to cover all of them. This poses another challenge, requiring us to identify a group of obfuscation techniques analyzable with our limited labor yet representative enough.

2.2.2 Static Third-Party Libraries. Third-party libraries have been an indispensable part of mobile apps. It is possible that an app "accidentally" got obfuscated due to the inclusion of obfuscated libraries without the awareness of app developers. Our analysis needs to capture such situations to avoid drawing biased conclusions. Unlike Android apps that are written in Java, iOS apps are written in languages that are more static, e.g., C, C++, Objective-C, and Swift. Due to Apple's security policies, iOS apps cannot use dynamic libraries from other vendors until iOS 8, meaning all third-party libraries have to be statically linked into app executables. The consequence is that there is no clear boundary between library code and an app's own code, making library detection in iOS apps a unique challenge [16, 33]. This is completely different from the library identification problem on Android, where application code is naturally assorted through the Java package hierarchy.

3 METHODOLOGY

We adopted a three-step process to conduct the empirical study. The first step is to select a representative collection of obfuscation techniques to consider, for reasons explained in Section 2.2.1. The second step is to search for a reasonably large set of iOS apps that are obfuscated before release. To date, there is no such a publicly available data set. Mining obfuscated samples from benign iOS apps is one of the major contributions of our work. For the third step, we inspect each obfuscated app in more depth and aggregate the harvested information to deduce empirical findings.

3.1 Considered Obfuscations

After decades of development, there are now numerous obfuscation techniques available. A comprehensive review by Schrittwieser et al. [37] included 22 classes of obfuscation methods proposed by previous research. For this study, we would like to focus on obfuscations popular among mobile developers and therefore worthy of in-depth investigation. We used Google to search for commercial and open source tools that can obfuscate iOS applications. By studying the statements and technical white papers of the top 10 results, we identified four families of obfuscations that are most widely supported, i.e., symbol renaming, exotic string encoding, control flow flattening, and decompilation disruption. Compared to all known obfuscation algorithms, this is a relatively small set, with the major reason being that the unique hardware and software environment on iOS devices imposes strict restrictions on the form of executable code. For example, iOS does not allow normal user applications to dynamically generate executable code, rendering self-modifying obfuscation technically impossible to implement. A graphical illustration of the four obfuscation algorithm families is given by Figure 1 while the technical details are briefly introduced as follows.

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

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

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

Control Flow Flattening. This technique "flattens" the original control flow graph of a function by rewriting the procedure into a huge switch-like structure [23]. This makes the logical links between basic blocks obscure.

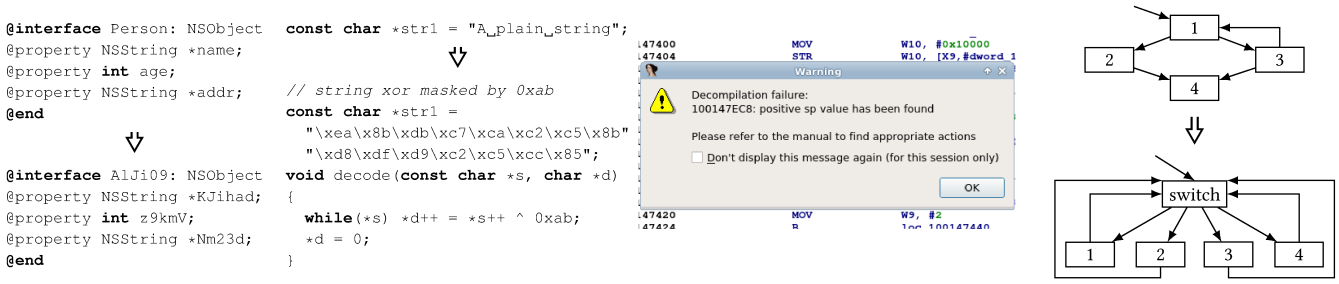
3.2 Mining Obfuscated iOS Apps

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

From February to October in 2016, we crawled 1, 145, 582 free iOS app instances, *including different versions of the same app.* We then try to identify apps that are obfuscated by at least one of the four families of algorithms in Section 3.1. Ideally, we could run automated detection over all the crawled apps for each obfuscation technique subsumed by the four families. However, obfuscation detection itself is a non-trivial task and is still being actively researched [11, 29, 31, 34]. For many obfuscation algorithms considered by our study, it is prohibitively expensive, if possible at all, to automatically detect their presence in over a million instances.

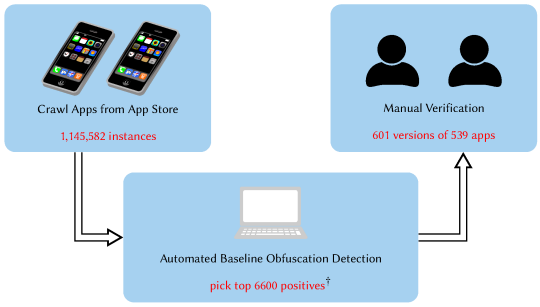
To tackle this problem, we identify a *baseline obfuscation algorithm* which is supposed to be the most widely adopted in mobile development. If developers indeed consider protecting their products, it is very likely that more than one obfuscation algorithm will be employed. In such cases, detecting the baseline obfuscation can help us identify the heavily obfuscated samples. Based on this insight, we developed an automated method to identify scrambled symbol names, since symbol renaming is considered by a large volume of previous research the most prevalent obfuscation method on mobile platforms [13, 24, 26]. In practice, symbol name scrambling imposes little execution cost while being highly effective in disturbing manual analysis.

Details of the detection algorithm are presented in Section 4. After running the algorithm for all crawled app instances, we obtained the likelihood of each app being obfuscated by symbol name scrambling. Based on the available man-labor, we examined the top 6600 most likely obfuscated samples, of which 601 are conformed to be true positives by manual verification. These samples, which can be further grouped into 539 applications identified by a unique ID assigned by App Store, are taken as the data set for subsequent



(a) Symbol renaming (b) Exotic string encoding (c) Decompilation disruption (d) Control flow flattening

Figure 1: Illustration of obfuscation techniques considered in the study



†The 6600 cut off is based on the maximum labor available for manual verification

Figure 2: Workflow for sampling obfuscated iOS apps

study. This sampling process is illustrated by Figure 2. We again emphasize that *these 601 samples should not be regarded as all the obfuscated apps among the* 1, 145, 582 *crawled instances*. We set the cut off at 6600 to bound the manual work within a manageable amount.

3.3 Per-App Inspection

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

3.3.1 Detecting Obfuscation. To detect the presence of anti-decompilation obfuscation techniques, we use IDA Pro [2], a commercial integrated reverse engineering environment that has been widely regarded as the de facto industry standard for analyzing binary code. IDA Pro can automatically dissect a binary executable into functions and translate the assembly code of each function to a high-level representation similar to C. We consider that a binary is protected by anti-decompilation techniques if IDA Pro reports too many failures. All results were manually validated.

To identify flattened control flows, we developed a binary analysis framework to disassemble app binaries and construct the control flow graph (CFG) of each function in a binary. If a CFG is flattened, most of its basic blocks will be included by a single loop, which can be captured by a standard loop detection algorithm [30]. Also, the "diameter" of the loop, which is defined as maximum length of the shortest path from the loop header to other basic blocks, should be of the logarithmic order of the total number of all basic blocks in the loop. Based on these two characteristics, we can find functions with flattened control flows.

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

3.3.2 Identifying Obfuscated Third-Party Libraries. As introduced in Section 2.2.2, we need additional manual effort to identify third-party libraries in the examined iOS apps if the library code contains any obfuscation by themselves. We decide if an obfuscated code region belongs to some third-party library by observing whether there are similar code patterns appearing in multiple samples developed by different vendors. Typical signatures of code patterns include control flow graphs, special algorithms, and uncommon data structures. Once a library is detected, we try to identify its origin through public information searching, with clues such as names of library APIs and special string literals, e.g., strings used for logging and generating crash reports. Some libraries do not provide even the most subtle information that can help reveal their identities. In such cases, we extracted the semantic signatures of obfuscated code, e.g., control flow patterns and unique data structures, and check if they appear in different apps.

3.4 Cross-Validation

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

way, we established a highly accurate and cross-validated protocols for the manual analysis on obfuscated iOS apps.

4 DETECTING SYMBOL OBFUSCATION

In practice, obfuscation tends to replace human-made symbols with randomly generated gibberish which can be detected by natural language processing (NLP) techniques. Previous research discovered that human-written source code is "natural" in the sense that it can be described by statistical language models [21]. Based on this insight, "unnatural" symbol names are possibly obfuscated.

4.1 An NLP-Based Detection Model

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

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

We use cross-entropy to capture the naturalness of an identifier. Intuitively, lower $H_{\mathcal{M}}(s)$ means s is more natural within $\mathcal{M}$. In particular, we adopt the n-gram language model that assumes the word sequences suit an $(n-1)$-order Markov process. Historically, n-gram has been utilized in various software engineering applications, including automated code completion [21] and bug detection [41]. Within an n-gram model, the definition of cross-entropy can be further formulated as

$$H_{n\text{-gram}}(s) = -\frac{1}{k}\sum_{i=1}^{k}\log_2 P(x_i|x_{i-(n-1)},\cdots,x_{i-1}) \tag{2}$$

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

$$L(I) = \min_{s\in S_I} H_{n\text{-gram}}(s) \tag{3}$$

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

4.2 Implementation

Considering that identifiers are usually not too lengthy, we can efficiently compute $L(I)$ in equation (3) using the Viterbi algorithm with the complexity of $O(nl^2)$, where n is length of the identifier and l is the length of the longest possible word in the language [38]. In fact, the worst cases can often be avoided, since most normal symbol names are already naturally segmented by programmers with underscores or the camel case scheme. We first compute the cross-entropy of an identifier by assuming the symbol is naturally segmented. If the entropy computed this way is already low enough, we can skip the relatively expensive Viterbi segmentation.

Our n-gram corpus contains two parts, i.e., the natural language corpus and the software source code corpus. Most identifiers in the crawled apps are named in English, but there are also many written in Chinese pinyin or even a mixture of English and Chinese. For English, we use a portion of the Google web trillion word corpus introduced by Franz and Brants [18] and derived by Norvig [32]. For Chinese, we employ the Lancaster Corpus of Mandarin Chinese (LCMC) [6]. As for the source code part, we crawled all identifiers appearing in iOS official APIs, which are all naturally segmented. Each identifier is then turned into a word sequence, thus forming a n-gram corpus.

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

$$p(s) = \begin{cases} \dfrac{p_{\text{EN}}(s)+p_{\text{CN}}(s)+p_{\text{code}}(s)}{3} & p_{\text{EN}}(s)+p_{\text{CN}}(s)+p_{\text{code}}(s) > 0 \\ 20^{-(|s|-1)}\cdot 2^{-(H+1)} & p_{\text{EN}}(s)+p_{\text{CN}}(s)+p_{\text{code}}(s) = 0 \end{cases} \tag{4}$$

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

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

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

5 FINDINGS

In this section we present 8 findings of our empirical study, grouped by their relevance to research questions raised in Section 1.

5.1 (RQ1) Characteristics of Obfuscated Apps

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

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

As previously mentioned, we paid special attention to third-party libraries when inspecting the obfuscated apps. The examination shows that these libraries indeed make a major source of obfuscation. In total, we captured 35 third-party libraries. The major functionality of each library, inferred by analyzing their code and retrieving publicly available information on the Web, is presented in Table 1.

Table 1: Obfuscated Libraries Grouped by Functionality

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

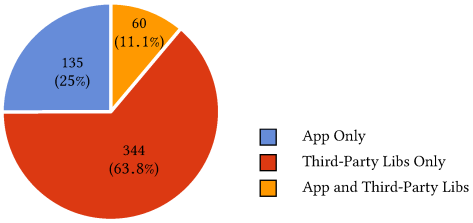
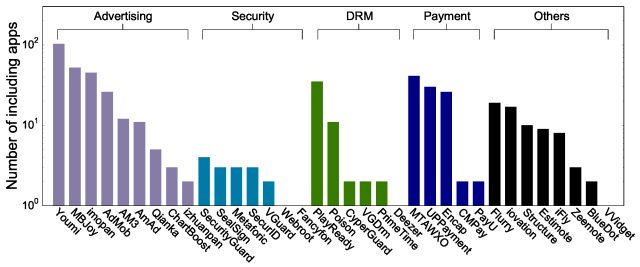
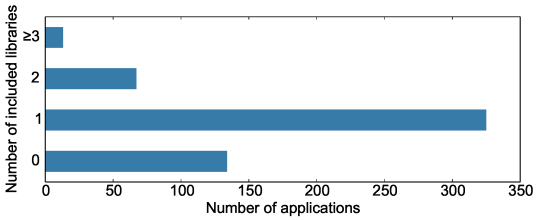


Figure 3: Origins of obfuscation in 539 obfuscated apps



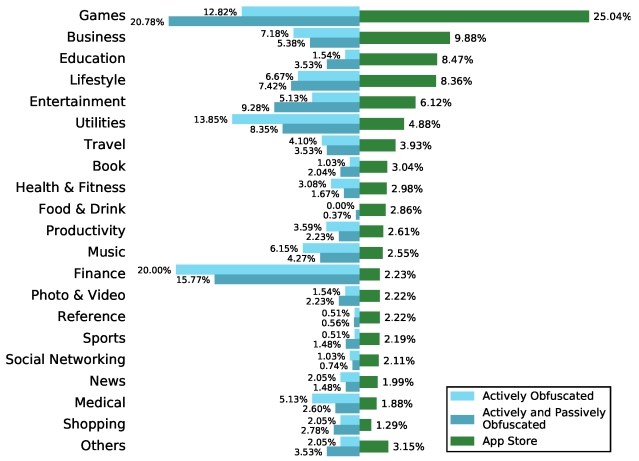
(a) Number of apps including each third-party library



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

Figure 4: Popularity of obfuscated third-party libraries

Figure 3 shows the breakdown of the origins of obfuscated code in the samples. Among the 539 apps employing obfuscation, 404 (75%) of them include at least one obfuscated third-party library. In particular, for 344 (63.8%) apps, the obfuscation is solely introduced by libraries. The popularity of these libraries can be further demonstrated in two aspects. Figure 4a shows for each library the number of including apps and Figure 4b shows the distribution of apps including obfuscated third-party libraries regarding the number of libraries.



Data for App Store from Statista [1]

Figure 5: Distributions of apps regarding their categories

Figure 3 indicates that the occurrences of obfuscation are mainly caused by the practice of depending on third-party libraries rather than app developers actively considering software protection. Based on the observation, we believe that it is important to consider the impact of third-party libraries for empirical software engineering research whenever app obfuscation is involved. To distinguish different sources of obfuscation, we henceforth call an app is *actively obfuscated* if its obfuscation is *not* entirely contributed by third-party libraries; otherwise it is called *passively obfuscated*.

The most notable kind of third-party libraries is for advertising purposes with both metrics being the highest in Table 1. Our preliminary analysis on some of these libraries shows that the obfuscated parts are used for communicating with the back-end ad servers. It is known that mobile advertising has been bothered by reverse engineering, through which a malicious party instruments advertising libraries to forge fake advertisement display or user clicks and tricks ad providers into paying in vain [25]. For ad providers, obfuscating their libraries is a reasonable response to such malicious attempts.

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

We found that in contrast to the distribution of all apps in App Store regarding their categories, the distribution of obfuscated apps has a vastly different pattern. This pattern varies further when the impact of third-party libraries is considered. Figure 5 shows the differences between these distributions, leading to the following key observations:

- The proportions of obfuscated apps in certain categories are exceptionally high compared to the shares of all apps in these categories across App Store, no matter whether passive obfuscation is taken into account. These categories are Finance (20.00%/15.77% vs. 2.23%), Utilities (13.85%/8.35% vs. 4.88%), Music (6.15%/4.27% vs. 2.55%), and Medical (5.13%/2.60% vs. 1.88%). According to our investigation, most of the obfuscated Music apps provide streaming services for copyrighted

musical contents. The inspected Utilities apps are mainly toolkit software providing assistance to daily activities, the majority of which regularly record user data that may be closely tied to personal privacy or enterprise secrets.
- For some other categories, the situation is flipped, namely the proportions of apps carrying obfuscated code are significantly lower than the store-wide ratios. Categories of such include Education (1.54%/3.53% vs. 8.47%), Book (1.03%/2.04% vs. 3.04%), Food & Drink (0.00%/0.37% vs. 2.86%), and Reference (0.51%/0.56% vs. 2.22%).
- The distributions of obfuscated apps in the Games, Finance, and Utilities categories are heavily influenced by obfuscated third-party libraries. Apps in the Games category are easily passively tainted by libraries. The Finance and Utilities apps, on the other hand, have a relatively higher rate for being actively obfuscated.

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

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

5.2 (RQ2) Obfuscation Patterns

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

- How many and what kinds of obfuscation techniques are found in the code;
- In what scopes the obfuscation algorithms are applied to the code, i.e., at the function level, class level, or module[1] level;
- Whether multiple obfuscation methods are applied to the same code region to achieve a synergy effect, which we call *synergistic obfuscation*.

We performed this pattern analysis on actively obfuscated apps and obfuscated third-party libraries separately. The results are presented in Table 2 and Table 3, respectively.

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

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

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

In theory, obfuscation can be manually conducted without the aid from automated tools [3]. Nevertheless, we believe this is not the case in mobile development. For actively obfuscated apps, the proportion of those employing module-level obfuscation is 55.90% (109 out of 195). For third-party libraries, the rate is even higher, reaching 71.43% (25 out of 35). Compared to function-level and class-level obfuscation, the workload of protecting one or more modules is significantly heavier, implying that *most mobile developers rely on automated tools for obfuscation.*

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

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

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

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

It is made clear by Table 2 and 3 that the popularity of the four obfuscation families vastly differs. The number of apps and libraries containing decompilation disruption and control flow flattening is remarkably smaller than the number of apps and libraries protected with scrambled symbol names and exotic string encoding. Due to our sampling methodology, symbol scrambling is naturally the most popular obfuscation technique across the data set. However, even without symbol scrambling considered, it is still true for the other three families of techniques that, the more costly it is to implement and deploy an obfuscation algorithm, the less widely it is adopted. To elaborate on this trend, we roughly discuss the difficulty of automating obfuscation each method and their impacts on run-time performance, in an increasing order.

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

Table 2: Numbers of Actively Obfuscated Apps Employing Different Obfuscation Patterns

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

Table 3: Numbers of Third-Party Libraries Employing Different Obfuscation Patterns

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

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

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

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

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

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

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

Finding A.2 shows that apps serving life-, money-, and privacy-critical purposes are more likely to be obfuscated. It is further suggested by Table 2 and Table 3 that the security strength of obfuscation applied to apps and libraries of these kinds is also notably stronger. In general, the Finance, Utilities, Games, and Music apps, if obfuscated, are more willing to employ expensive obfuscation techniques, i.e., decompilation disruption and control flow flattening. These apps also tend to employ more different families of obfuscation techniques. For example, over half (20 out of 39) of the actively obfuscated Finance apps contain plural kinds of obfuscation. Moreover, in many cases (18 out of 20), these different methods were applied to the same part of the code, achieving synergistic obfuscation. Also, the scope of obfuscation in these apps is often larger, mostly reaching module-level protection.

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

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

As aforementioned, our sample crawling was continuous and lasted for nine months. For apps getting version updates during the crawling period, we were able to analyze the temporal evolution of their obfuscation patterns. With these historical versions and some additional examinations, we confirmed that 27 of the 195

actively obfuscated apps were unobfuscated at the beginning of the crawling period. It is very likely that developers of these apps were newly attracted by the benefits of software protection and started to employ it as part of their software engineering routines. Note that 27 is a possibly untight lower bound because the recorded version histories may be incomplete because of the limited workload capacity of our crawler.

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

5.3 (RQ3) Impact of Distributor Code Review

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

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

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

The first case is a heavily obfuscated app developed by a reputed commercial iOS security service provider, which only published that single app in App Store. Judged from the simplicity of its functionality, this app is merely a minimal working example of iOS development, whereas it is protected by all four kinds of obfuscation techniques considered by our study. Only two among the 195 actively obfuscated apps are obfuscated in this pattern. We speculate that the security solution provider submitted this app to address the concerns that their obfuscation algorithms may cause distributor review alarms, to the detriment of the sales of their services. It is known that App Store have various constraints on submitted apps, some of which may not be clearly documented. For example, each slice of an executable file in iOS apps must not exceed 60 MB [5] if the app is to be compatible with older versions of iOS, limiting the use of code transformations that bloat binary size too much. These constraints intrigue obfuscator writers to test the boundaries of acceptable obfuscation techniques. This case suggests that *developing new mobile obfuscation algorithms has to take the app vetting process into account to be practical.*

In the second case, we found that a third-party advertising library contains code for calling private iOS APIs, which is strictly forbidden by Apple App Store security policies. To circumvent store reviews, the library writer uses the `dlopen` system call to avoid direct linkage to internal iOS frameworks providing private APIs. The library then uses exotic string encoding to obfuscate the string literals provided to `dlopen` as parameters. In this way, Apple's vetting analysis failed to detect this violation. By searching related information on the Internet, we learned that this library was once caught using private iOS APIs in 2015 [4], long before we started crawling samples from App Store. Shortly after the incident was reported, Apple announced that it had removed all apps contaminated by this library from App Store. Yet our findings show that either authors of the library managed to bypass the app review process for another time or Apple failed to detect all apps including this library. Whichever is the case, this finding serves as empirical evidence that *obfuscation is not only employed to repel malicious reverse engineering but also for infiltrating publisher inspection*, even though this practice is previously regarded as a signature of malware.

By nature, ad providers are impelled to collect as much client data as possible for developing more effective ad distributing strategies, potentially placing themselves on the verge of infringing user privacy. Considering the large quantity of obfuscated third-party advertising libraries and their wide spread in the sample set, we are concerned by the possibility that abusing obfuscation to bypass publisher security policy enforcement is becoming a common practice for aggressive adware on the mobile. Mobile apps falling within a "gray area" that are controversially benign or malicious, aka. "grayware," has drawn attention from the security community [10].

5.4 (RQ4) Effectiveness of Obfuscation

We now present our findings regarding the effectiveness of real-world obfuscation for mobile apps. It should be emphasized that our goal is not to access the security strength of obfuscation techniques themselves like previous literature review did [37] but to investigate whether iOS developers are able to appropriately utilize these techniques and optimize the protection effects.

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

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

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

- Tokens assigned to apps for accessing third-party services. Some enterprise entities provide APIs for mobile apps to retrieve proprietary information or upload app usage data for analytics, usually at a price. Requests for accessing these services has to be sent with tokens issued by service providers to prove the identities of requesting clients. We found that some apps store these tokens as plaintext in variables whose names are not scrambled.
- In-app secrets. Apps may encrypt their private data such as execution logs and intermediate results before storing them on mobile devices. Some poorly obfuscated apps store encryption keys in plaintext as string literals.
- Information about back-end servers connected with the apps and the corresponding communication protocols. *In particular, we found* 4 *apps, which are the mobile clients of some financial institutions, leaking the URLs or IP addresses of their back-end testing infrastructures.* Surprisingly, accessing these infrastructures does not require any authentication. The communication protocols and even internal documentations are exposed to anyone knowing the URLs or IPs.

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

6 IMPLICATIONS OF THE RESULTS

Through this empirical study, we learned that third-party libraries play a significant role in iOS app obfuscation, which is consistent with the situation on Android [24]. Being a major source of obfuscated code, third-party libraries affect software attributes in various aspects without app developers being aware. We urge that future studies on iOS app repositories to take obfuscated third-party libraries into consideration and develop dedicated analysis techniques to handle them.

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

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

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

7 RELATED WORK

To the best of our knowledge, most historical work on mobile app analysis targets the Android platform. The Android Malware Genome project is among the earliest research efforts that perform large-scale analysis on mobile app repositories [47]. By working on over 1200 samples, the authors managed to present a systematic characterization on existing Android malware. According to this research, mobile malware authors by then had already started to apply obfuscation to bypass anti-virus analysis. Besides malware that harms users, mobile app repackaging that harms the interest of developers has also drawn attention. Various tools and systems have been developed to detect and analyze cloned mobile applications with both accuracy and scalability [15, 19, 39, 46]. Researchers have also worked on examining third-party libraries used by mobile developers. Tools like LibRadar [28] and LibD [24] were developed to detect third-party libraries in Android apps and classify them. Research by Chen at al. [16] detects libraries potentially harmful to user security and privacy for both Android and iOS.

Despite the progress in mobile app analysis, most studies of this kind either ignored or spent very limited effort in handling the presence and influence of software obfuscation. One of the few studies that systematically investigated the impact of obfuscation on mobile development is from Linares-Vásquez et al., who researched how obfuscation can affect the detection of Android code cloning [26]. Similar to our work, Linares-Vásquez et al. spent extensive manual work in identifying obfuscated code, but their analysis only covered 120 apps and did not consider obfuscation methods other than identifier scrambling. CodeMatch is a similar project that focuses on obfuscation-resilient Android library detection [20]. Xue et al. [44] proposed adaptive unpacking of Android apps to recover dex code, which can potentially enable obfuscation-resilient clone or library detection.

8 CONCLUSION

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

ACKNOWLEDGMENTS

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

REFERENCES

[1] Apple: most popular app store categories 2017 | Statistic. https://www.statista.com/statistics/270291/popular-categories-in-the-app-store/.

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

[3] The International Obfuscated C Code Contest. http://www.ioccc.org.

[4] iOS Apps Caught Using Private APIs. http://sourcedna.com/blog/20151018/ios-apps-using-private-apis.html.

[5] iTunes Connect Developer Guide. https://developer.apple.com/library/content/documentation/LanguagesUtilities/Conceptual/iTunesConnect_Guide/Chapters/About.html.

[6] The Lancaster Corpus of Mandarin Chinese. http://www.lancaster.ac.uk/fass/projects/corpus/LCMC/.

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

[8] Shrink Your Code and Resources | Android Studio - Android Developers. https://developer.android.com/studio/build/shrink-code.html.

[9] Smart Obfuscation for iOS Apps | PreEmptive Protection. https://www.preemptive.com/products/ppios.

[10] Benjamin Andow, Adwait Nadkarni, Blake Bassett, William Enck, and Tao Xie. 2016. A Study of Grayware on Google Play. In *Proceedings of the 2016 IEEE Workshop on Mobile Security Technologies (MoST '16)*.

[11] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS '14)*.

[12] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP '17)*. 633–651.

[13] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. 343–355.

[14] Hao Chen, Daojing He, Sencun Zhu, and Jingshun Yang. 2017. Toward Detecting Collusive Ranking Manipulation Attackers in Mobile App Markets. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (AsiaCCS '17)*. 58–70.

[15] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering (ICSE '14)*. 175–186.

[16] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P '16)*. 357–376.

[17] Zhaofeng Chen. iOS Masque Attack Weaponized: A Real World Look. https://www.fireeye.com/blog/threat-research/2015/08/ios_masque_attackwe.html.

[18] Alex Franz and Thorsten Brants. All Our N-gram are Belong to You. https://research.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html.

[19] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*. 431–444.

[20] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. CodeMatch: Obfuscation Won't Conceal Your Repackaged App. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '17)*. 638–648.

[21] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering (ICSE '12)*. 837–847.

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

[23] Tımea László and Ákos Kiss. 2009. Obfuscating C++ Programs via Control Flow Flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica* 30 (2009), 3–19.

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

[25] Wenhao Li, Haibo Li, Haibo Chen, and Yubin Xia. 2015. AdAttester: Secure Online Mobile Advertisement Attestation Using TrustZone. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*. 75–88.

[26] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2014. 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 '14)*.

[27] Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. 2017. Stochastic Optimization of Program Obfuscation. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. 221–231.

[28] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16 Companion)*. 653–656.

[29] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. 757–768.

[30] Steven S. Muchnick. 1997. *Advanced Compiler Design Implementation*. Morgan Kaufmann.

[31] Minh Ngoc Ngo and Hee Beng Kuan Tan. 2007. Detecting Large Number of Infeasible Paths Through Recognizing Their Patterns. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. 215–224.

[32] Peter Norvig. Natural Language Corpus Data: Beautiful Data. http://norvig.com/ngrams/.

[33] Damilola Orikogbo, Matthias Büchler, and Manuel Egele. 2016. CRiOS: Toward Large-Scale iOS Application Analysis. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '16)*. 33–42.

[34] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT '09)*.

[35] Andre Pawlowski, Moritz Contag, and Thorsten Holz. 2016. Probfuscation: An Obfuscation Approach using Probabilistic Control Flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 165–185.

[36] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques.. In *Proceedings of 23rd Network and Distributed System Security Symposium (NDSS '16)*.

[37] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1 (2016), 4:1–4:37.

[38] Toby Segaran and Jeff Hammerbacher. 2009. *Beautiful data: the stories behind elegant data solutions*. "O'Reilly Media, Inc.".

[39] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. 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 '15)*. 71–82.

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

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

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

[43] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2016. Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method. In *Proceedings of the 19th Information Security Conference (ISC '16)*. 323–342.

[44] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. 2017. Adaptive Unpacking of Android Apps. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. 358–369.

[45] Babak Yadegari, Brian Johannesmeyer, Benjamin Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P '15)*.

[46] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards Obfuscation-resilient Mobile Application Repackaging Detection. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks (WiSec '14)*. 25–36.

[47] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12)*. 95–109.