Methods for Improving the Quality of Software Obfuscation for Android Applications

Methoden zur Verbesserung der Qualität von Softwareverschleierung für Android-Applikationen

Der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Yan Zhuang
aus Henan, VR China

Als Dissertation genehmigt von
der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Tag der mündlichen Prüfung:	26. September 2017
Vorsitzende des Promotionsorgans:	Prof. Dr.-Ing. Reinhard Lerch
Gutachter:	Prof. Dr.-Ing. Felix C. Freiling
	Prof. Dr. Jingqiang Lin

Dedicated to my dear parents.

Abstract

Obfuscation technique provides the semantically identical but syntactically distinguished transformation, so that to obscure the source code to hide the critical information while preserving the functionality. In that way software authors are able to prevail the resources e.g. computing power, time, toolset, detection algorithms, or experience etc., the revere engineer could afford. Because the Android bytecode is practically easier to decompile, and therefore to reverse engineer, than native machine code, obfuscation is a prominent criteria for Android software copyright protection. However, due to the limited computing resources of the mobile platform, different degree of obfuscation will lead to different level of performance penalty, which might not be tolerable for the end-user.

In this thesis, we optimize the Android obfuscation transformation process that brings in as much "difficulty" as possible meanwhile constrains the performance loss to a tolerable level. We implement software complexity metrics to automatically and quantitatively evaluate the "difficulty" of the obfuscation results. We firstly investigate the properties of the 7 obfuscation methods from the obfuscation engine Pandora. We evaluate their obfuscation effect with 9 different software complexity metrics, when iteratively apply each obfuscation method multiple times to more than 240 Android applications. We show from the result pictures that the obfuscation methods can exhibit two types of properties: monotonicity or idempotency. For most of the monotonicity obfuscation methods, their variants of the complexity values are constant and stable, which are the foundation for our statistic based algorithm to optimize the complexity results.

To reach the desired complexity, we then design and implement our obfuscation framework which can select the optimum obfuscation techniques for the target complexity and apply them to Android applications, while measure the performance cost. The optimization process in our framework is controlled by the Obfuscation Management Layer, which implements the Naïve Bayesian Classifier algorithm to select the obfuscation techniques.

Our obfuscation framework can transform the result APKs to arbitrary target complexity. Meanwhile, the unpredicted performance loss will be caused by the obfuscation. We compare the discrepancies of CPU cycles of original APKs to their obfuscated versions by dynamic testing, and define them as the performance penalty generated by obfuscation. We evaluate the performance penalty of different obfuscation methods. We find out that some of the obfuscation methods consume significantly more performance at the same time have minimum impact on complexities.

Finally, we statistically measure performance losses of different methods, and calculate them as a special metric in the Naïve Bayesian Classifier algorithm. We therefore can optimize the performance cost of the obfuscation to target a tolerable level. Meanwhile, to maximize the code coverage, we develop an automatic testing tool which are used to generate the testing cases.

Zusammenfassung

Obfuskierungstechniken liefern eine semantisch identische aber syntaktisch unterschiedliche Transformation, in welcher der Qüllcode versteckt wird um kritischen Informationen unter Beibehaltung der Funktionalität zu verbergen. Auf diese Weise können Software-Entwickler mehr Ressourcen von Reverse-Ingenieuren einfordern, z.B. Rechenleistung, Zeit, Toolset, Erkennungsalgorithmen oder Erfahrung etc. Da der Android-Bytecode praktisch relativ einfach zu dekompilieren und damit zu Reverse Engineeringen ist, im Vergleich zu nativen Maschinencode, bietet Obfuskierung ist eine herausragende Möglichkeit zur Umsetzung von Urheberrechtsschutz bei Android-Software. Aufgrund der begrenzten Rechenressourcen mobiler Plattformen wird jedoch ein unterschiedlicher Grad an Obfuskierung zu einem unterschiedlichen Leistungsniveau führen, was möglicherweise für den Endbenutzer nicht tolerierbar ist.

In dieser Arbeit optimieren wir den Android-Obfuskierung-Transformationsprozess, der die „Schwierigkeit“ erhöht, während er den Leistungsverlust auf einem tolerierbaren Niveau hält. Wir implementieren Softwarekomplexitätsmetriken, um die „Schwierigkeit“ der Obfuskationsergebnisse automatisch und quantitativ zu bewerten. Zunächst untersuchen wir die Eigenschaften der 7 Obfuskationsmethoden der Obfuskierungsengine Pandora. Wir bewerten ihre Obfuskierungswirkung mit 9 verschiedenen Softwarekomplexitätsmetriken, jeweils als iterative Anwendung der Obfuskierungsmethode mehrfach auf über 240 Android-Anwendungen. Aus den Ergebnisbildern leiten wir ab, dass die Obfuskierungsmethoden zwei Arten von Eigenschaften aufweisen können: Monotonie oder Idostose. Für die meisten der Monotonie-Obfuskationsmethoden sind die Varianten der Komplexitätswerte konstant und stabil, was die Grundlage für unseren Statistik-basierten Algorithmus zur Optimierung der Komplexitätsergebnisse bildet.

Um die gewünschte Komplexität zu erreichen, konzipieren und implementieren wir dann unser Obfuskierungsframework, das die optimalen Obfuskierungstechniken für die Zielkomplexität auswählen und auf Android-Anwendungen anwenden kann, während die Performanzkosten gemessen werden. Der Optimierungsprozess in unserem Framework wird durch die Obfuskierungsverwaltungsschicht (Obfuscation Management Layer) gesteürt, welche den Naïve Bayesian Classifier implementiert, um die Obfuskationstechniken auszuwählen.

Unser Obfuskierungsframework kann die Ergebnis-APKs in eine beliebige Zielkomplexität transformieren. Dabei wird ein unvorhergesehener Leistungsverlust durch die Obfuskierung verursacht. Wir vergleichen die Diskrepanzen von CPU-Zyklen der ursprünglichen APKs mit ihren obfuskierten Versionen durch dynamische Tests und definieren sie als Performanzstrafe, welche durch die Obfuskierung ensteht. Wir werten die Performanzstrafe für verschiedenen Obfuskierungsmethoden aus. Unser Ergebnis zeigt, dass einige der Obfuskierungsmethoden deutlich mehr Leistung kosten und zur gleichen Zeit nur minimale Auswirkungen auf die Komplexität haben.

Schließlich analysieren wir die Leistungsverluste verschiedener Methoden statistisch und berechnen sie als spezielle Metrik der Naïve-Bayessche Klassifikation. Wir können daher den Leistungsverbrauch der Obfuskierung optimieren, um ein tolerierbares Niveau zu erreichen. Um die Code-Abdeckung zu maximieren, entwickeln wir gleichzeitig ein automatisches Test-Tool, dass zur Generierung von Testfälle verwendet werden kann.

Acknowledgments

This thesis would not have been accomplished without support from others. First and foremost, I would like to thank my doctoral advisor Felix Freiling for giving me the opportunity to work with him at his Security Research Group at the Department of Computer Science at the Friedrich-Alexander University Erlangen-Nürnberg. My academic progress is founded on his continues guidance and support. I would also like to thank Jingqiang Lin, from the Chinese Academy of Sciences, for agreeing to be my second reviewer to this thesis, and for the stimulating discussions we had in the 2014 DAPRO workshop. I also thank my colleagues at the Security Research Group, for a cheerful and friendly working atmosphere.

I would like to extend my thank to Mykolai Protsenko, the author of *Pandora* obfuscation framework which plays an important role in this research, for his theoretical and technical support.

In addition, I wish to thank (in alphabetical order): Michael Gruhn for not only his proof-reading the thesis, recommending the Grammarly grammar checker and, but also, illuminating input of the ideas and theory; Christian Moch for his helpful discussions, his assistance at the preliminary stage of this research, especially on the database constructions, and also, proof-reading the thesis; Philipp Morgner for his proof-reading publications this thesis is based on, and corrections to the German language translations; Tilo Müller for his advising and collaborating on publications this thesis is based on, and his helpful support on other projects; Ralph Palutke for proof-reading parts of the thesis, interesting discussions and sharing ideas.

Last but not least, I would like to thank my wife Yaling Li, for her years of spiritual support for my ideals, and unconditional trust on my capability and potential, even when my confidence and morale hit rock bottom when I faced frustrating setbacks. I also thank my parents, for giving me the strength to move forward and for the courage they brought me during the darkness before dawn.

Mum, Dad, Ling, and Felix, I couldn't have gone this far without you.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Software obfuscation, the process of transforming a program into semantically identical but syntactically distinguished, has become an essential part of software protections against malicious reverse engineering. Because obfuscation provides an "unintelligible" functionality preserving technique [41], which aims at making the target code more complex so as to prevail the resources, such as computing power, time, toolset, or experience, that the reverse engineer could afford. However, there have always been controversies in the software obfuscation research area. Though some of the research indicate that, in theory, the universal obfuscation is infeasible [10]. They showed that when the execution path contains f_s, which belongs to the set of *unobfuscatable* functions $\{f_s\}$, the secret asset s could be easily extracted. Meanwhile, an efficient reverse engineer could not acquire any information of s with non-negligible advantage given merely black box access. Garg et al. [41] implements the functional encryption scheme to demonstrate a general purpose indistinguishable obfuscator for the polynomial-size circuits. Yet the effectiveness of hiding the secret asset s in practical usage is unknown.

The widespread of the malware samples which try to evade signature based detection with different appearances, and the prevalence of commercial software which thwarts plagiarism by their obscurity, illustrate that the obfuscation technique is nevertheless popular in practical use.

Still, it is yet to be explored, to what extent modern obfuscation techniques can defeat human or automatic analysis in the arm race between the developer and the reverse engineer [80], that is, how much "difficulty" can obfuscation bring into the software. The most direct methodology of evaluating this "difficulty" is to test the human's analyzing performance in a statistical or abstract way. The "difficulty" of reverse engineering lies in how to reconstruct enough source code information to modify or reuse the program. Dalla Preda et al. [72] therefore provided the abstract interpretation method to model the obfuscation (hiding information) and the reverse engineering (acquiring information), to compare the potency of different obfuscations. Ceccato et al. [23] conducted the statistics on the acquired information accuracy and the time expenses of different groups of subjects (human).

Human based evaluation is intuitive and demonstrative, however lacking of efficient or quantitative accuracy. To automatically and quantitatively measure the "difficulty" level, we implement a collection of software complexity metrics to evaluate the obfuscation results. Each of the complexity metrics represents one aspect of "difficulty". The metrics types and the complexity values can be chosen according to the given security requirements [22]. For instance, metric *Dep-Degree* describes the number of variables

needed to be considered by the reverse engineer [17], the increase of which would make source code intuitively more confusing. The metric *Lack of Cohesion* provides the cohesiveness that represents the complexity inside the class. The higher the value, the more perplex is the internal structure of the Object-Oriented design. Another advantage of using complexity metrics to evaluate the obfuscation result is that the simplicity and the briefness will make the measurement more efficient. As a result, quantitative research on a large scale of obfuscation effectiveness will be more feasible compared to human based analysis.

Besides the complexity, the performance of the obfuscation result also concerns the software authors, because a well protected but extremely slow-running software would worsen the user experience significantly. However, the tolerances to the performance penalty of the obfuscation are not unanimous for all scenarios. On one hand, some software can only permit a slight performance loss. For example, the fluency of all the mobile games is so vital for the player that the obfuscation degree (the complexities) must be as high as possible together with a strict control of performance loss. On the other hand, some software do not require much fluency but their safety is the priority, e.g. the online backing APK of *SparKasse*. This thesis aims at improving the protection of mobile software by obfuscation, at the same time optimizing the performance of the obfuscation result.

1.1 Contributions

Properties of Obfuscation Methods on the Complexity Metrics In Chapter 5, we give an overview of the properties of the obfuscation methods regarding to different complexity metrics. To investigate the properties, we iteratively obfuscate the target set of APKs for multiple times with the same transformation methods, and then evaluate the mean and deviation of their increased complexity values of different metrics and their similarities to the original APKs. According to our empirical observation, there are two types of properties: idempotency and monotonicity. However, the same obfuscation method has different properties regarding to different metrics. We show that for the monotonous obfuscation methods, the variants of the complexity values are constant and stable, which is the desired property for our further optimization of the obfuscation. Our research lay the foundation for using statistic based algorithm to optimize and obfuscate the APKs to reach the target complexity.

Complexity Metrics Optimized Obfuscation In Chapter 6, we present the algorithms that can optimize the obfuscation process of the APK to satisfy the target complexity metrics. We apply the different monotonous obfuscation transformations on the APK learning set. For each of the transformation method, we find that each sample set of result complexity variation ratios exhibits a significant statistical feature in the distribution histogram, the mean values, and the standard deviations. We assume that the result distributions affiliate to the Gaussian distributions. We show that the Naïve

Bayes algorithm can classify the conditional probability of the Gaussian distribution, select the transformation method and apply it to the APK, as a result maximizing the probability of reaching the target complexity value. Furthermore, we present another algorithm, the mean value based simple search, which is able to select the transformation method whose mean complexity variation has the closest Euclidean distance to the target complexity value. To compare the performance of the two algorithms, we use our obfuscation framework to implement one of the algorithms respectively to transform 110 APKs. Our evaluation shows that the Naïve Bayes classifier outperforms the mean value based simple search significantly, and majority of the APKs can achieve the target complexity values. This work serves as the base of our further performance research on the obfuscation methods.

Performance Penalty of Obfuscation Methods and Complexity Metrics Our obfuscation framework can transform the result APKs to arbitrary target complexity. Meanwhile, the unpredicted performance loss will be caused by the obfuscation. In chapter 6, the scale of performance loss of the transformed APK is evaluated for the practical use of the performance optimized obfuscation: We compare the discrepancy of CPU cycles of the original APKs to their transformed versions. Our framework either targets different metrics and complexity values, or only transforms the APKs with different obfuscation methods regardless of the complexity. In targeting different metrics and the complexity values, we find that the performance losses constantly and stably increase with the target complexity values, regardless to the different obfuscation methods are used.

Moreover, for the same target complexity values, the *Dep-Degree* is the most performance saving metric and the C*yclomatic Complexity* is the most performance expensive. For the different obfuscation methods, we show that the Strings Encryption, the Move Methods, and the Compose Locals are the top three performance expensive methods. We do the statistics for more than 100 APKs as the learning set to evaluate their performance regarding to different obfuscation methods. This research serves as the foundation to use performance as an independent metric during the transformation process for the performance optimized obfuscation.

Performance Optimized Obfuscation In chapter 6, we use performance as an independent metric during the obfuscation process to confine the performance loss of the result APKs to a certain level. According to the evaluation of performance losses of all the metrics, we find that some of the performance saving obfuscation methods have a relatively large complexity values increase. Those methods can be more frequently used in the obfuscation process for the performance saving goal while target complexity increase are the same. Therefore, we do the statistics of the performance losses of different methods the same way as the other complexity metrics, assuming them to affiliate to the Gaussian distribution, which could be calculated by the Naïve Bayesian Classifier algorithm. We evaluate the performance optimized result APKs and find that their average

performance expenses significantly drop.

1.2 Related Works

Obfuscation in Theory Many of the obfuscation techniques firstly appeared in malware samples [80]. To evade signature based anti-virus detection [85], the polymorphic engine had been widely implemented in the malware to generate the "syntactically different but semantically identical" [80] versions in the early 1990's. Since then, the polymorphic techniques make those malware more complex and difficult to detect [64].

Since the innovation work of Collberg et al. [29], a taxonomy of the obfuscation techniques, considerable research in this code obfuscation area have been proposed in the literature. Not only arguing that automatic obfuscation is the most practical method to protect the software from plagiarism, Collberg et al. also classify the different obfuscation transformations and roughly evaluate their quality from three aspects: the potency - to what extent the human reverse engineer can be confused; the resilience - how hard it is for the automatic de-obfuscator to recover the hiding content; and the cost - how much performance overhead is added. In this way, these attributes can be adjusted for a certain obfuscation goal. For instance, the user can choose a transformation with high potency, strong resilience, and costly in performance, or a transformation with medium potency but without performance expense.

Theory Evaluation In theory, it is "impossible" to construct an obfuscator that is universally applicable to all programs [10]. However, Dalla Preda et al. [71] designed a formal framework to theoretically analyze and certify the effectiveness of different obfuscations by the abstract of semantic interpretations, which is the amount of obscurity the transformation added to the program. In this framework, the attacking activity A represents the abstraction of the semantic information, which is the amount of information that can be observed from the program, while the semantic preserving transformation T is trying to protect the concrete properties of the program. Through the comparison between the degree of abstraction of the activity A and the properties preserved by T, it is possible to measure if T is effective against the attack from A, and help researchers understand the potency and resilience of the transformation T. In this way, the potency and resilience of the different transformations can also be compared with each other, as well as the ability of different attackers regarding to a certain obfuscation.

In the research [73], Dalla Preda et al. showed that the abstract interpretation-based methodology in their framework can be used to significantly mitigate the effect of opaque predicate obfuscation. However, Giacobazzi et al. [43] showed that by using some of the code obfuscations to transform the program to a state of semantically incompleteness, the abstract interpretation-based analysis in the framework will not be precise. As a result, the program can be protected from the attack of abstract interpretation method.

Obfuscation Automatic Assessment Several literature have proposed signature based detections to roughly evaluate the effectiveness of the obfuscation. Rastogi et al. [77] developed *DroidChameleon* framework to obfuscate the malware samples and evaluate the minimum obfuscation requirements to pass the anti-virus detection. Huang et al. [50] obfuscated the Android applications and measured their similarity with the cleartext version.

However, the above works do not exactly measure the effectiveness quantitatively. Zheng et al. [99] built the *ADAM* system to automatically transform the mobile malware samples into different obfuscated versions. And then the detection rates from the different anti-virus software are measured. According to the results, the obfuscation methods which cause the complexity changes will significantly reduce the detection rates. For instance, the modification of the control flow graphs decreases the average detection rates from the 93.78% to 76.67%. The encryption of the strings decreases from 93.78% to 50.95%.

Complexity Assessment Software complexity metrics can be used to evaluate the effectiveness of automatic obfuscation [28], by quantitatively illustrate to what degree the program structure has been changed, or how many more elements should be considered in order to understand the program. For example, by adding more condition switches with opaque predicates transformation, the metric Cyclomatic Complexity [95] will be increased. By encrypting the strings to prevent the critical information disclosure, there will be more computing variables added to the program because of the encryption algorithms. The metric value of the data dependencies [17] will be increased.

Some of the Android anti-virus or piracy detection literature are not directly related to the software complexity. Nevertheless, the change of the complexity metrics by obfuscation can reasonably lower detection rates of the malware or plagiarism.

To find the repackaged APKs in the third-party Android market, Zhou et al. [100] implemented the *DroidMOSS* system to measure the similarities by extracting the opcode sequence as the fingerprint. However, if DepDegree has changed by obfuscation, the opcode sequence based fingerprint will be extracted differently. Thus, the repackaging detection rate will be lower. Crussell et al. [31] presented the *DNADroid* system to compare the Java methods dependency graphs between candidate APKs to spot plagiarism. But if the metric *Lack of Coherence* and the metric *Coupling between Objects* of the APKs are changed by the *Move Methods* obfuscation or the *Extract Methods* obfuscation, it will be more difficult to find a pair of matched dependency graphs. The same effect could be applicable to the work of Hanna et al. [46]. They proposed Amazon EC2 cloud based *Juxtapp* system, which hashed the features extracted by the *k-gram*, to find vulnerable code sections, detect malware and piracy.

Intuitively, if we apply obfuscation to the research objects of the above literature to change their complexities, e.g. obfuscate the APKs in the Android market in the research of Zhou et al. [100], the accuracy of the detection results will be clearly affected.

Therefore, the complexity metrics can be recognized as the index of the effectiveness of the obfuscation, though the relation between the obfuscation effectiveness and the complexity changes still need to be researched.

Obfuscation Effective Human Assessment In a series of papers, Ceccato et al. [24, 25, 23] pioneered research in using experiments with humans to measure the strength of different obfuscation methods. In those papers, the authors use statistical methods to calculate and compare the performance of participant subjects in reverse engineering experiments. They use these results to compare different obfuscation methods and their protection quality.

Ceccato et al. [24] compare the human reverse engineering performance on cleartext (unobfuscated) source codes and on code obfuscated by identifier renaming and opaque predicates. They distinguish between a comprehension task (code understanding) and a change task (code modification). Unpaired analysis is used to calculate the performance discrepancy between groups treated with those different source codes. The conclusion is that in both cases, the cleartext group outperforms the obfuscated code group, but for the change task the performance difference is not statistically significant.

Ceccato et al. [25] repeated their experiments, this time using only one obfuscation method (identifier renaming). In contrast to their earlier work, the authors used paired analysis to test if different tasks make a difference in the performance. In these experiments, most test results are statistically significant. Additionally, the factors of ability, experience, test environment and participant's learning curves are also included and related to the performance results. Therefore, the authors not only illustrate that source code attacks can be mitigated by obfuscation, but a quantified measurement is provided to elaborate on how much strongly the obfuscation techniques could protect the source codes.

Ceccato et al. [23] summarize and extend the above results. The authors compare identifier renaming and opaque predicates using human experiments and come to the conclusion that identifier renaming, though much simpler, is more effective against human reverse engineering than opaque predicates.

Performance Assessment The performance of the transformed APKs are vital for the user experience. There are different methodologies to research on the performance of the result APKs and the validity. Majority of them focus on accurately modeling or measuring the energy consumption of the APKs. To encompass the lower OS level power behaviors, e.g. file open and close, GPS module, or drivers, Pathak et al. [68] proposed a system-call-based power modeling scheme for various low level power optimizations programmed in the device drivers, which significantly improved the accuracy of the energy drain profiling. To avoid the energy-greedy APIs and discover their usage patterns, Linares et al. [94] used a hardware monitor to profile the energy consumption for different methods calls while executing the APKs. However, the methods energy

consumption curves measured by the hardware have to be approximately matched with the execution time slot of the methods. This can be a threat to the accuracy of the assessment. Pathak et al. [67] presented a case study to scale the internal energy dissipation of the applications e.g. AngryBirds, Browser, Facebook, and etc.. Not only did they find that up to 75% of energy is spent in third-party advertisement modules, they also spotted the "energy bugs" in the source code.

However, literature on performance assessed by the CPU cycles of the mobile devices are limited, which can more directly describe the performance penalty of the obfuscation.

1.3 Publications

In the paper "*An Empirical Evaluation of Software Obfuscation Techniques applied to Android APKs*" [40], authored together with Felix C. Freiling and Mykola Protsenko, we investigated the problem of creating complex software obfuscation for mobile applications. This paper, used in Chapter 5, elaborates obfuscation methods behaviors on the different complexity metrics, and is the basis for Chapter 6, which applies obfuscation in practice to reach a target complexity values.

Based on the properties of the obfuscation methods in our previous paper, we have developed a framework to optimize the target APKs to a certain complexity values by using different kinds of optimization algorithms. The research paper "*Approximating Optimal Software Obfuscation for Android Applications*" [101] was created under the guidance of Felix C. Freiling. The framework and the Naïve Bayesian Classifier were accomplished by the author of this thesis. The Pandora obfuscator and the SSM for compleixty assessment were from the work of Mykola Protsenko.

Chapter 6 explores the performance loss of the obfuscation results in regards to different complexity metrics or obfuscation methods. Finally, we can use the same framework as described in the previous paper to optimize the performance of the obfuscation results. It is based on our submitted yet under reviewing research paper "*The Performance Cost of Software Obfuscation for Android Applications*", which was also done under the guidance of Felix C. Freiling. In this research, all the environmental setup and the programming which includes the framework, the Naïve Bayesian Classifier, and the black box testing tool, were accomplished by the author of this thesis.

The paper "*An(other) Exercise in Measuring the Strength of Source Code Obfuscation*" [102] was coauthored with Mykola Protsenko, Tilo Müller, and Felix C. Freiling. We used this experiment to test the effect of obfuscation from the human perspective. The experiment software was developed by Mariano Ceccato. The experiment process design and the result statistics were accomplished by the author of this thesis. However, because this research topic is not related to our main research purpose, we do not include it in this thesis.

1.4 Outline

This thesis is organized as follows: Chapter 2 provides the basic concepts and technical background necessary to implement our research. In Chapter 3, we present a survey on the development of PC malware and mobile malware. Chapter 4 focuses on introducing the components and their working scheme in our obfuscation framework. Chapter 5 evaluates the attributes of the obfuscation techniques on different software complexity metrics. In Chapter 6, we optimize the obfuscation process to produce the transformation result with target complexity and a limit of performance loss. Finally, we conclude our work in Chapter 7, and propose future work.

Chapter 2

The Background of Android and Obfuscation

In this chapter, we will illustrate the basic concepts and techniques this thesis is built on. Our explanations provide the reader with the specialized technical knowledge necessary to understand our work. This is not intended to be a complete description of the Android operating system, but a compact primer on the concepts utilized within this thesis.

2.1 Android Background

2.1.1 Android Applications

Android applications are mainly developed with the Java language and their distributions are in the form of *Android Packages* (the APK files). APK files are signed *zip* files which compress the Android bytecode *DEX* file together with its Manifest file, resources, third-party libraries, and data. The Figure 2.1 shows the build process and the toolchain that converts the Java source code project to the ready install *Android Package*.

Though the compilation process of Android application is very different with the Java application, it starts in the same way:

The Java source code **.java* is firstly compiled into the standard Oracle JVM Java bytecode file **.class*, a stack based intermediate representation (IR), by the *javac* compiler. In this step, the interface to the Resources *R.java*, the Java libraries, and the Manifest file are also compiled together with the source code.

And then the *dx* tool translate all the **.class* into one bytecode *DEX* file *classes.dex*. The *classes.dex* is the 3 address register based bytecode IR, which can be assembled and executed by the *ART*, or interpreted by the *Dalvik* virtual machine of Android. At the same time, the third-party libraries, e.g. the social network or the advertisement interfaces of Android, are integrated into the *classes.dex* file. In this thesis, the obfuscation transformation we focus on is at this IR level.

To form the Android Package file **.apk*, the *Android Asset Packaging Tool* (*aapt*) is implemented to combine the *DEX* file *classes.dex* with the compiled resources which are indexed by *R.java*. However, this *Android Package* file cannot run, because it is not signed yet. It can be signed with the debug or release keystore for different purposes by the *jarsigner* tool, which is the part of *Oracles Java Development Kit*.

As the last step, the result **.apk* file needs to be processed by the *zipalign* tool, which makes sure the byte boundaries are perfectly aligned in its compressed part. After all the above steps, the Android package is ready for the distribution.

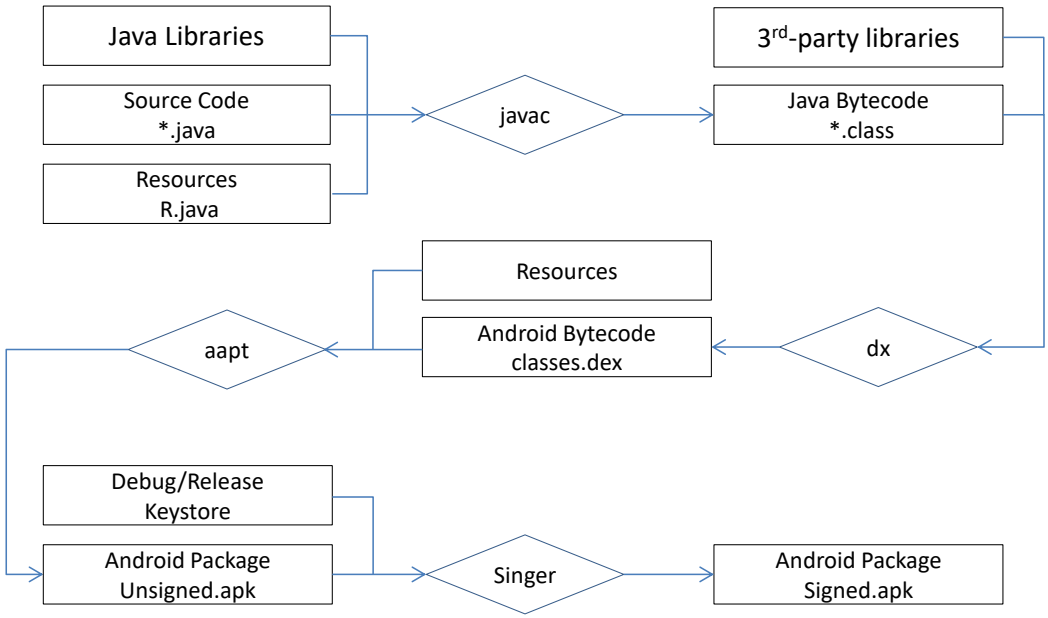

Figure 2.1: Android application build process

2.1.2 Android Software Stack

Android is a Linux based open source software stack led by Google and Open Handset Alliance (OHA) for a wide array of devices. The Android software stack includes the customized Linux kernel, hardware interface, the native libraries middleware with the Android Runtime, the Java framework, and built-in applications. All those components are shown in Figure 2.5.

Linux Kernel The Linux kernel is the foundation of the Android operating system, which implements fundamental Android system level functionalities and instructions. For example, the fork of the new application process from the *Zygote*, the thread scheduling in this new process, the memory management, and the garbage collection all rely on the function of the Linux kernel.

Besides Linux kernel functions, the key security features of Android platform are also inherited from the Linux user-based protection mechanism to setup the kernel-level Application Sandbox, the filesystem permissions, the encryption, and etc., in order to identify and isolate the different application processes with their files and memory resources. The security features include: the user-based permissions model, the process isolation, the Binder IPC.

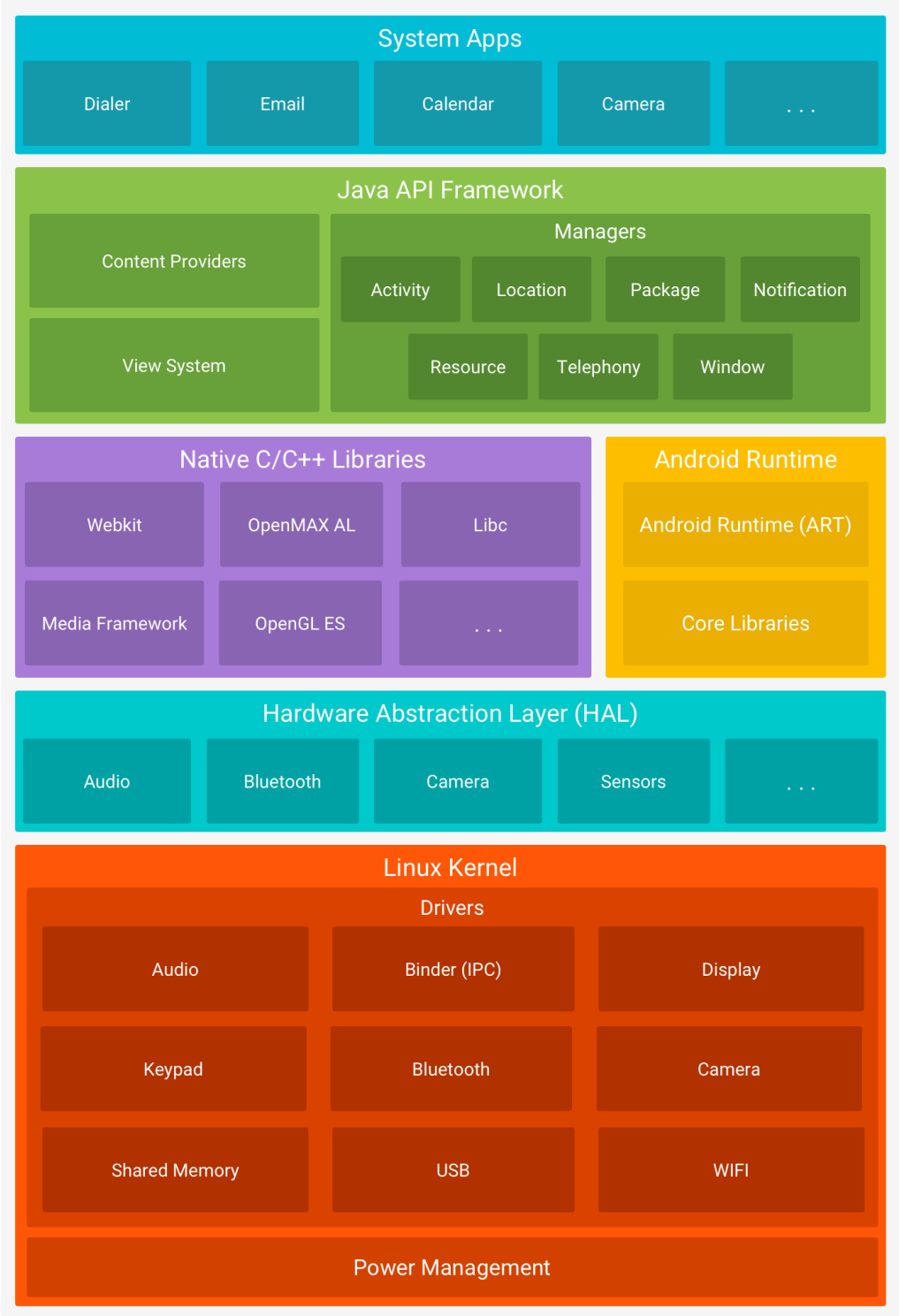

Figure 2.2: Android Software Stack

Hardware Abstraction Layer (HAL) The Hardware Abstraction Layer is composed by a collection of libraries that is specifically implemented as the standard interfaces for the all hardware modules. The high level Java framework in the Android software stack can directly access the hardware functions, e.g. camera or bluetooth module, according to the interfaces.

Android Runtime Each Android application has its own Android Runtime (ART) virtual machine instances in their own process. ART is designed for the execution of the DEX files in low-memory devices. Different from Dalvik VM, which executes the DEX file by interpreting, it implements ahead-of-time (AOT) compilations to firstly translate and optimize the DEX into the native code by LLVM [1], and then execute the native in CPU. Not only running the application, ART also provides very important debugging support, for example, the sampling profiler for the performance of the Java methods, the diagnostic of the exceptions, crash reporting etc..

Native C/C++ Libraries The native libraries support the core components and services in Android system which are built from the native code and relies on the native libraries, such as ART and HAL. This native libraries platform can be directly accessed from the Android NDK.

Java API Framework The *Java API framework* provides the entire functions set of Android OS with building block components for the application developer. To implement those functions, we can simply import the core, system module, or the services in the form of *android.** packages in the application source code. The components of this framework are running as the background service processes in the Android OS, and providing their functions when called by the applications. The processes include but are not limited to the following:

- *Activity Manager*, the service that manages the lifecycles of the applications, which are *onCreate()*, *onStart()*, *onStop()*, *onDestory()* etc..
- *Content Provider*, the service that shares data between applications. For instance, the *Contacts* can share its own data with other software.
- *View System*, used to build user interface in the app.
- *Telephony Manager*, it manages phone calls and provides the service of device telephony information, such as the cell location and the unique device identifier (IMEI).

System Apps The *System Apps* are the Android applications built on the top the *Java API framework* and designed for the interaction between the application users and devices. The Android system offers a collection of basic task applications for the

end user, e.g. Telephone, SMS, Google Map, and Browser. The user can also install third-party applications from Google Play for various purposes, e.g. Zhihu, BiliBili, or WeChat.

2.2 Android System Startup

The initialization of the Android OS starts from the very bottom of the software stack in Figure 2.5 and proceeds to the top, in other words, from the booting up of the Linux kernel, to the *zygote* process including Android Runtime and Native Libraries, and finally to the services in the Java API Framework. Finally, when the Android applications are successfully started with the support of those framework services, the Android startup procedure is finished.

This section provides a dynamic formation of the Android system, focusing on the important processes of each stack layer. We show the derived relationships between the processes, as well as the functions of each process at different stages, most notably the *zygote* and the *ActivityManagerService*. In addition, the dynamic profiling part of our obfuscation framework assumes an understanding of the Android OS working principle and the processes mentioned in this section.

2.2.1 Startup in Linux Kernel Level: The *init* Process

Figure 2.3 shows the procedure of the starting up of the Android OS, which begins with the first process *init* from the Linux kernel level. Similar with the Linux system, the *init* is the direct or indirect predecessor of all the other processes in Android. One of its direct child processes is the *Zygote*, the most important process in Android which forks all the application processes and system service. To fork the *zygote*, the following command line in the startup script *system/core/rootdir/init.rc* is interpreted and executed by the *init* process in the first stage of the Figure 2.3.

```
1 service zygote /system/bin/app_process -Xzygote /system/bin
     --zygote --start-system-server socket zygote stream 666
```

From this script, the service *app_process* is forked and initialized from the *init* process (Stage 1). Its name is then changed into *zygote*. The parameter *socket* means the *zygote* process is allocated with the *Unix Domain Socket* type resource under the directory */dev/socket/* with the name "*zygote*" (at the end of Stage 2). The *SystemServer* process is started at the same time (Stage 3).

2.2.2 Startup in Android Runtime and Native Level: The *zygote* Process

zygote is a daemon and its only mission is to fork the processes of the applications. Because of the Linux user based security mechanism, each of the application has its

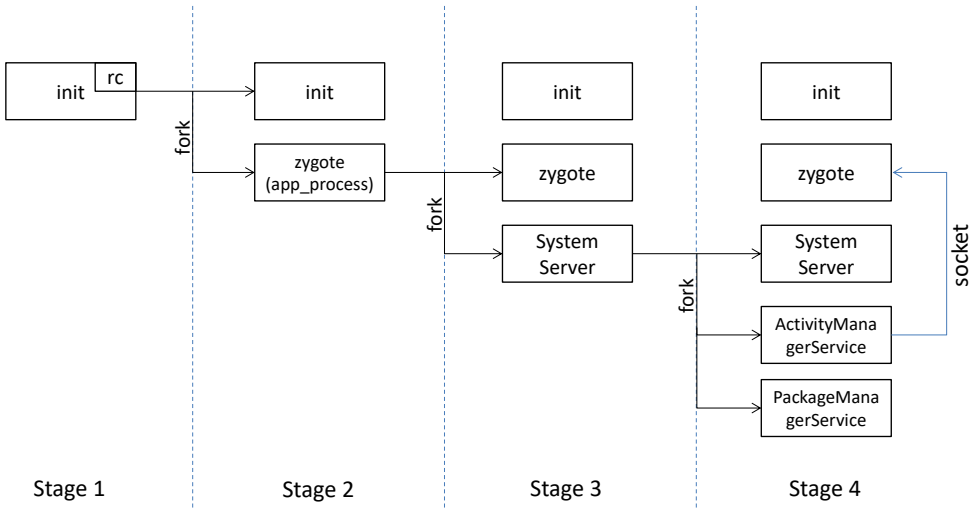

Figure 2.3: Android Starting Up Processes

independent process in the Android system, and their *DEX* bytecode is interpreted and executed by its own instance of the *Dalvik/ART* virtual machine. The performance of the Android system might be suffered, if the virtual machine has to be initialized in each process. To avoid the time consuming application starting process, the application processes will be forked from the *zygote* which already contains a running virtual machine instance [57] with all the necessary classes and libraries preloaded (the runtime), so that all the process context will be directly inherited from the *zygote*.

Android Runtime The *Android Runtime* (not the *ART*) is the APIs class of the runtime library in Android, which is the environment in the software stack supporting the running of the applications in the Android system. It implements the execution environments and fundamental behaviors of the Android programming languages (Java and C++). The *Android Runtime* includes *Dalvik* or the *ART* virtual machine, the Android Java class library (e.g. *java.lang*, *java.util*, *java.net*), the *Java Native Interface* (JNI), and the C library (libc).

Reading from the source code of the Android system, we elaborate its starting procedure as following: After the instance of *Android Runtime* is started (the *AndroidRuntime.start()* function), the instance of virtual machine is then created (*startVM()*) and all the necessary threads inside the virtual machine are initialized, such as garbage collection and profiling which will be used later in our obfuscation framework. Meanwhile, the interface to access native threads is registered (*startReg()*) and attached to the virtual machine (*attachVM()*).

At this point, the preparation of runtime environment is finished. It is the most time consuming part in the starting of the *zygote* process. The virtual machine (VM) and the threads inside are initialized. The native library could access the VM and the threads by the interface object *JavaVM()* and *JNIENV()*. The native threads are registered in VM. For example, the native profiling thread is initialized and registered in the *ART* VM at this step, and then can be called to run at the beginning when the application process is forked from the *zygote*.

2.2.3 Startup in Java API Framework Level: The *SystemServer* Process

The *SystemServer* is forked by *zygote* in the third stage of the Figure 2.3. It creates the *ServerThread* which derives the *UI thread* and the *WindowManager thread*. The derived threads are running in their loops (Android Looper mechanism), to cooperate with each other and subsequently create the service processes, which are important for maintaining the functionality of the Android system.

There will be a serial of services created from this stage, e.g. *Activity Manager*, *Package Manager*, *Account Manager*, *System Content Providers*, *Battery Service*, *Sensor Service*, *Window Manager* etc.. The most important among them are the *Acitivity Manager* and *Window Manager*. When the initialization of those services is finished, the *systemReady()* function in the services is called by the *SystemServer*. It will launch the user

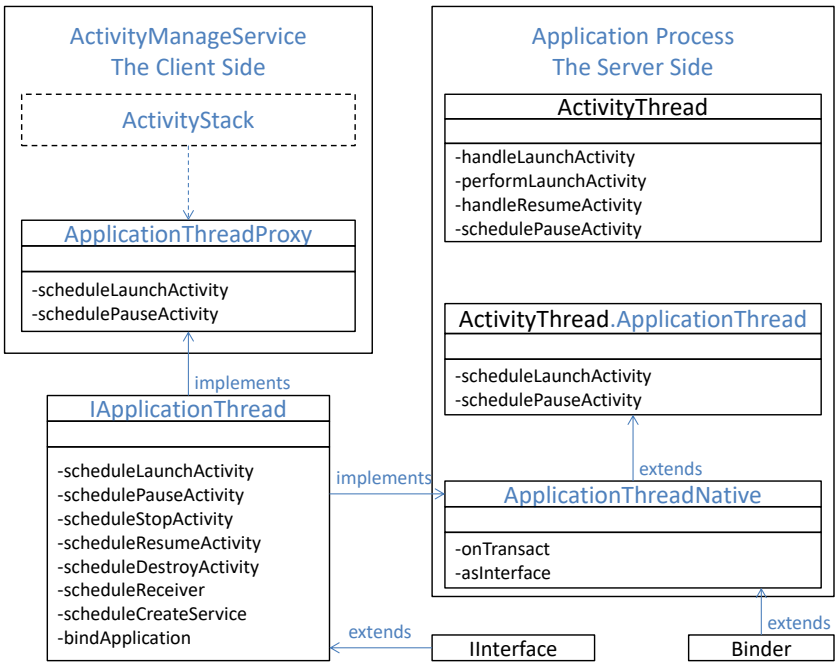

Figure 2.4: Class Diagram of the *ActivityManageService* managing the Process

interface of the system and the "HOME" of Android, or other designated startup applications, e.g. the television program in the Android TV. Meanwhile, in this stage, the *zygote* uses *runSelectLoopMode()* method to start an infinite loop to monitor its socket for the request of forking a new application process.

ActivityManageService In the last step of Figure 2.3, a serial of system services are created by the *SystemServer* (Stage 4). The start up stages of the Android system are then finished. After that the Android system is ready to launch any end-user application with the *Intent*.

When received an *Intent* to start an new activity, the *ActivityManageService* acquires the activity information from the parsing of *AndroidManifest.xml* by the *PackageManager*.

After that, the *ActivityManageService* communicates to the *zygote* by socket (which is created from previous stage) to fork a new activity process. In this new process, the instance of *ActivityThread* is loaded to manage the main thread, which assimilates to the main entrance of the Java program. The instance of *ApplicationThread*, which is the inner class of *ActivityThread*, is initialized to dispatch and execute different activities or broadcasts according the requirement of the *ActivityManageService*.

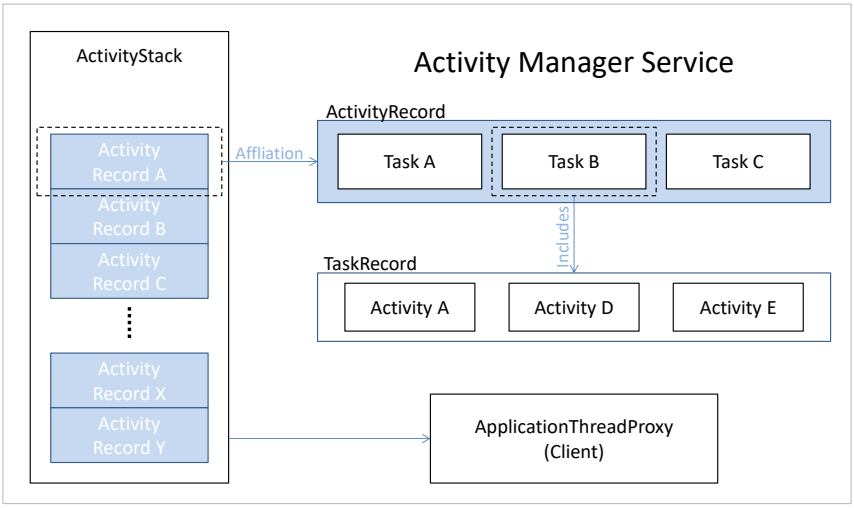

Figure 2.5: The Activity Stack in the *ActivityManageService*

The Figure 2.4 shows the class diagram of the *ActivityManageService* which communicates and schedules the activity in this process. With the interface *IApplicationThread*, the *ActivityManageService* uses the instance of *ApplicationThreadProxy* as a client, to communicate with the instance of *ActivityThread* as the server. Therefore, the threads of different activities in this process can be scheduled to launch, pause, stop, resume, or destroy.

In the default case, all of the activities and services, which belong to the same application, are running in the same process. However, maintained by the *ActivityManageService*, the *ActivityStack* (in the dashed rectangle in the Figure 2.4) can use the *ApplicationThreadProxy* to schedule and assign activities and services from different applications to execute. In this case, they will be executed in the main thread of this process.

Figure 2.5 shows the data structures of the *ActivityStack* maintained by the *ActivityManageService*. The *ActivityStack* is used to record the history of activities which have been invoked. When the activity in it is destroyed, the stack pops it out. If a new activity shows on the screen, it will be pushed onto this stack.

An application can include multiple activities. However, during the running of an activity, other activities from different applications are needed sometimes. For instance, if we are surfing the contact list application and try to send out a message, the button of *Message* needs to be clicked from the *view* of contacts. Then the activity of "Writing a Message" from another application is shown on the screen (The underlying work is done by *Intent*). The collection of the activity in the contacts and the activity in the messenger are defined as the *Task*. So the multiple activities, which serve the same

mission but do not come from the same application, are pushed onto the stack within one data structure named *TaskRecord*.

As shown in the Figure 2.5, in this stack, the single activity is stored as the *ActivityRecord*. Each of an *ActivityRecord* maps into one or more *TaskRecords*. It means this activity can be used in multiple *Tasks*. Each of the *TaskRecord* is a stack of the activities which serves the same mission. Therefore, the *ActivityManageService* uses these sets data structures to control the schedules of the processes and threads in the runtime with the client *ApplicationThreadProxy*.

2.3 Obfuscation

The unified definition is given by Collberg [29], to comprehensively describe the obfuscation. It serves as the foundation of our further development of the obfuscation concept and process.

Definition 1. *We consider the function* $\Omega : \mathbb{P} \to \mathbb{P}$ *as the transformation of a program. If observed testing result of* P *and* $P_\sigma = \Omega(P)$ *are consistent, the* Ω *could be defined as the obfuscation transformation. More precisely:*

- *If* P *can not terminate, or exit because of an error, the* P_Ω *may or may not exit.*
- *otherwise,* P_Ω *will exit and have the same output with* P*.*

According to the above definition, all sorts of transformations which preserve the original semantics can be recognized as obfuscation methods. Thus, the optimization and the compilation of the program can also be considered as obfuscation. In certain cases this consideration is correct, i.e. the optimization and the compilation could make the program harder to understand than its source code form. This is because during the software's development process, the source code is cleartexted and developer-oriented, which makes it easier for authors to cooperate and manage the project. However, the optimization and the compilation are machine-oriented. All the tags, comments, and the regular variables' and methods' names are useless for the CPU. To shrink the size of the programs, those elements are usually removed. Therefore, the optimization or compilation a program can also be recognized as obfuscation.

However, the effectiveness of the obfuscation transformation as defined above does not necessarily thwart reverse engineering. For example, after the optimization or compilation, the string "Login Failed" is still in the same place of the program. Revealing of the location might make it possible for the attacker to bypass authorization. At the same time, the difficulty to recognize the control flow, the call graph, or the inheritance of the classes are not changed. More insight information of the program can be easily extracted by an experienced reverse engineer. Thus, the obfuscation effectiveness is not guaranteed.

An effective obfuscation requires the obscurity of the possible attributes of the program which might be potentially exploited by the reverse engineer, e.g. the location of "Login Failed" in the program. Furthermore, the degree of effectiveness depends on the raised "difficulty" level brought by the obscurity for these potential attributes. The attributes in the program shall be obscured with a certain "difficulty" level are listed as following:

- The whole or the particular code section of the program, e.g. algorithms or functions, code in methods, expiration date, encryption keys, login information such as the location of "Login Failed" strings etc..
- Object-Oriented Design, e.g. structures of methods and classes, inheritance relation of classes, location of algorithms and functional code section in programs, call graphs in the programs etc..
- Data or Control Flow, e.g. values assignments to particular variables, dependencies among instructions, code execution paths, obscured predicates etc..
- Identifier protections, e.g. renaming variables, renaming classes and methods.

To define the obfuscation effect to certain attributes, the following definition is given [29].

Definition 2. *We consider Ω to be the process of obfuscation transformation, P^σ is the program with the target attributes σ on which the obfuscation is trying to apply, the result program is $P^\sigma_\Omega = \Omega(P^\sigma)$. The analysis framework A can compute the σ out of the program before $(\sigma = A(P^\sigma))$ and after $(\sigma_\Omega = A(P^\sigma_\Omega))$ the transformation, iff:*

$\sigma_\Omega \not\approx \sigma$, and to compute $A(P^\sigma_\Omega)$ consumes more resources than $A(P^\sigma)$, the obfuscation is effective.

$\sigma_\Omega \approx \sigma$, and to compute the $A(P^\sigma_\Omega)$ consumes approximately equal resources to $A(P^\sigma)$, the obfuscation is uneffactive.

$\sigma_\Omega \approx \sigma$, and to compute the $A(P^\sigma_\Omega)$ consumes less resources to $A(P^\sigma)$, the obfuscation is defective.

The Definition 2 confines effective obfuscation to be when more resources (e.g. time, experiences, brainpower, computing power, or the toolset) are needed for reverse engineering to certain attributes, i.e. effective obfuscation will raise the "difficulty" level to understand or modify the program. Therefore, to prove the effectiveness, the "difficulty" of reverse engineering before and after the transformation are computed and compared, by comparing the change of the reversed σ or the change in resources spent on reverse engineering.

However, as we have mentioned in Chapter 1, human based comparison of obfuscation effect such as the research of Ceccato et al. [23] is intuitive and demonstrative, yet lacking of efficient or quantitative accuracy. Thus, to create the automatic obfuscation framework, we measure the complexities of the program before and after transformation to make the "difficulty" of reverse engineering quantitatively comparable. More precisely,

the obfuscation effect of attribute σ in the program which we are trying to protect is evaluated by its different sorts of complexity. The following section will introduce the definition of potency of obfuscation to certain attributes measured by corresponding complexity metrics.

2.3.1 Software Complexity Metric

The software complexity metrics are normally used in the process of software development, to evaluate the design and the structure, meanwhile to estimate the progress of development, and to measure the quality of the project. In our research, we use the them to evaluate the potency (effect) of obfuscation in the Definition 2. Because software complexity metrics can also reveal how much the structure is deteriorated by the obfuscation methods, and how many more factors (the added switches, the redundant parameters etc.) need to be put into consideration for reverse engineering, so that can reflect the obfuscation potency, the "difficulty".

Intuitively, the higher the metric values are, the more complex the program will be. For instance, if more conditional statements are added to the source, we need to concern about more switches during the reverse engineering stages. If the data structures are transformed to a form with more variables calculations (increase of the DepDegree), we need to concern more parameters in order to deobfuscate.

The potent obfuscation [29] is defined as follows:

Definition 3. *Let Ω is a obfuscation transformation, P^{δ} is a program with the target attribute δ, $P^{\sigma}_{\Omega} = \Omega(P^{\sigma})$ is obfuscated version of P^{σ}. $A = \{A_1, A_2, ..., A_n\}$ is the toolset of analyzer.*

For the P^{σ}, if $\exists A_i \in A$, A_i is an effective obfuscation, while for the the other $\forall A_j \in A$, according to A_j, Ω is not a defect obfuscation, then the Ω can be defined as potent.

In our research, the Definition 3 shows that if obfuscation increases at least one complexity metric, and will not decrease the complexity values of the other metrics, this obfuscation can be recognized as *potent*.

To show the internal complexity of the program P, the following metrics can be evaluated: The number of operators and operands; the number of predicates (Cyclomatic Complexity); the nesting depth of the conditional statements; the number of basic blocks; the number of formal parameters and global data structure (the Fan-In and Fan-Out Complexity); the static data structure complexity (e.g. DepDegree); the Object-Oriented Design complexity, e.g. number of methods, depth of the inheritance tree (DIT), coupling between classes (CBO), and reponse set of the methods etc..

Furthermore, too much complexity will add extra performance overhead to the program, i.e. in the program, the same code functions but more parameters will surely need more calculations, thus more computing powers from CPU.

Therefore, to make sure our obfuscation is practical, we will consider the performance loss in our obfuscation framework.

2.3.2 Obfuscation Performance

The performance loss from the obfuscation is an important factor to be considered. We define the performance influenced by obfuscation as follows:

Definition 4. *Let the I is a dynamic testing case which denotes the fully and single operation routine testing of the program P^t, $t(t = I(P))$ is the executing time of P under this testing. After the potent obfuscation Ω, $t_\Omega(t_\Omega = I(P_\Omega))$ is the executing time of obfuscated program P_Ω under the same testing case I. Consider δ, which is $t_\Omega - t$:*

If $t_\Omega >> t$, $\delta >> 0$, Ω are the performance loss transformation.

If $t_\Omega \approx t$, $\delta \approx 0$, the Ω are the performance free transformation.

If $t_\Omega << t$, $\delta << 0$, the Ω are performance optimized transformation.

Majority of obfuscation methods will cause performance loss. It is because after transformation, those obfuscations will make the software structure more complex, since they will add more redundant data structure.

While some obfuscations are performance free transformations, they only rename the variables, methods, and the classes of the target programs. The data, control flows, and the object structures remain the same. In certain cases the performance free transformation can protect the software as effectively as the performance costly transformation [23].

As we mentioned after Definition 1, the transformation from the optimizer or the compiler could also be recognized as obfuscation in certain cases, e.g. after optimization and compilation, the C language is difficult to understand when reversed into the source code.

For the user experience, the performance optimized cases are more preferable. However, it does not apply in all cases. For instance, optimized and compiled Java program probably makes no significant difference in "difficulty" for the reverse engineer.

Therefore, performance optimized obfuscation, unfortunately, only exists in certain cases. And the choices of performance loss free obfuscation are limited. But since different users have difference tolerance level, there can be a certain flexibility in the allowance for performance loss.

There is no unique standard of how much performance burden can be tolerated. It depends on the software types and the user requirement. Some programs rely very much on its real time response. The obfuscation omitting performance concern will significantly affect user experience.

Some programs run on a supercomputer or a computing grid, which both incur more costs as the computing time increases. If the runtime environment remains the same,

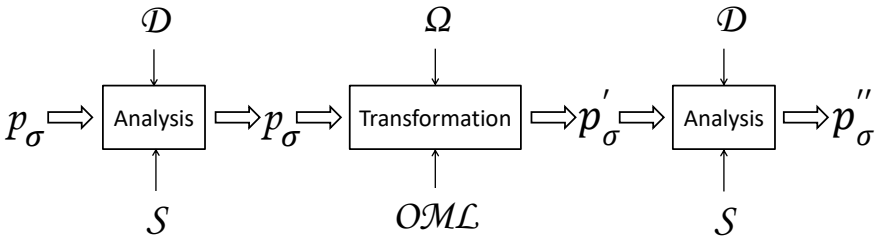

Figure 2.6: The Process of Obfuscation

obfuscation will lead to more complex control flows and data structures, which means more time is needed to executes the programs. It would cause time, computing power, furthermore economic losses for the users.

Other programs have very strict security requirements. Without the protection of obfuscation, it is possible to cause critical privacy information leaks or economic losses. Meanwhile, their response or execution time during operations is not the key criterion to evaluate the software. And the launching frequency is relatively low compared to the other software. Therefore, a certain extent of decline in user experience caused by performance loss is conceivably tolerable in this case.

Different cases have different requirements for obfuscation. Clear requirements will help balance the complexity metrics change and performance loss, and can optimize obfuscation results for both of the program users and the authors.

Different obfuscation transformation can cause ultimately different degree of performance loss, even when the complexity metrics changes are similar. Thus, research on the relation among obfuscation transformations, complexity metrics, and performance loss are necessary. This is the motivation of this dissertation.

2.3.3 Obfuscation Process

According to the above definitions in this section, we have developed our own concept of obfuscation process as shown in Figure 2.6 to blueprint our performance and complexity optimized obfuscation framework. In this Figure, the program p with its target attributes σ is transformed by the selected obfuscation transformation. After the comparison in both dynamic testing (performance) and static analysis (complexity) before and after obfuscation, a potent and performance optimized obfuscation result p'' is generated.

The S represents the static analysis and optimization tool frameworks, which provides the intermediate representations (IR) to the target programming languages and to represent the abstraction structure of the program. With this IR, the program can be altered or optimized with the data flow, the control flow, and the architecture of the whole program. Meanwhile, to evaluate the effect of the transformation, the complexity analysis of the program is also implemented by the static analysis tool.

Besides static analysis, dynamic analysis is also indispensable for this obfuscation process. The D are the possible testing cases set for the program to check if the obfuscation result remains the same functions, and if its performance fulfills our expectation. Because the goal of our research requires to know if the performance losses of the program after the obfuscation are tolerable. Thus, dynamic analysis, which is a black box test with multiple designated input, is implemented to this program before and after obfuscation to measure the differences of the execution time to evaluate the changes of performance.

The Ω represents the set of semantics-preserving transformations which protecting our target attribute σ, according to our Definition 2. During the transformation process, the *Obfuscation Management Layer* (OML) will select the obfuscation methods from Ω and subsequently apply them to the target program.

The OML is the module that decides which obfuscation methods should be used on a particular part of program for the desired "difficulty" level, which are the complexity increase in our research according to Definition 3. It manages the transformation process and can affect the overall result. We define the module OML as the *Obfuscation Management Layer*. The conceptual detail and realization would be introduced in Chapter 4.

According to the Figure 2.6, to obfuscate the target program, firstly static or dynamic analysis is applied to the program to construct the IR out of the program and evaluate the complexity and the performance. The *Obfuscation Management Layer* OML therefore chooses the suitable obfuscation methods from the set Ω, to transform the IR to a semantically identical but syntactically distinguished status. And then the result performance and the complexity after the transformation are measured again. Ideally, the result of the obfuscation should satisfy our desired "difficulty" level, and the performance loss is tolerable.

Our framework, as introduced in the Chapter 4, is built to realize this blueprint.

Chapter 3

The Evolution of Mobile Malware Parallel to PC Malware

In the 1993 American comedy "Groundhog Day", a reporter finds himself caught in a daily time loop while reporting about an annual event. The movie has become an idiom for deja vu type incidents in which history is repeated — deterministically, but in a way that in hindsight could have been foreseen. Looking into the area of computer security, many of us feel like being in a time loop when viewing the evolution of malicious software for mobile devices. In this section, we demonstrate the milestones in the malware history in both PC and mobile platform. We shows that obfuscation technique - firstly applied in the malware to avoid a fixed fingerprint presented to anti-virus software - is one of the most eventful milestones in both of the time lines.

3.1 Introduction

Malware is a program that compromises computers and performs malicious tasks. For different purposes, different infection methods and non uniformed operating systems and platforms, there is no common infrastructure for the malware. With prosperity of computer science and development of technology, the evolution of malware has never stopped, and even accelerated. Increasingly complex structures and more stealthy infiltration techniques make it one of the most serious IT security problem in the world. Worse still, it moves form the traditional PC platform to the mobile platform, making our smartphones vulnerable for security issues and economic losses.

Normally, malware uses deception, exploit tools or concealment methods to disguise its original offensive intent to infect as many victim devices as it can. After the initial infection, it typically attempts to either sabotage and steal sensitive user data from the compromised computers or to undermine the security of the whole system.

Awareness towards detrimental traditional PC malware has been largely heightened among the public. Although the concept and development of malware is not a total surprise, for most modern smartphone users, the advent of malware for such devices comes as a seemingly new and unexpected threat.

Yet the threat of malware posed on mobile, or smartphone, appears to be well-known in the research community[15, 86]. Therefore, we try to make the public as aware of the danger of mobile malware as those of traditional PC platforms.

From the first and brief observation, there are many parallels between classical desktop malware and malware for mobile devices. But in what sense does mobile malware follow

similar evolution steps as classical desktop malware? In this chapter, we present the results of our study and show that both desktop malware and mobile malware exhibit strong parallels in their development. Every step taken in the desktop malware evolution can be found in the mobile malware evolution with similar chronological order, albeit at a dramatically increased speed. Is this a time loop like in the "Groundhog day" movie? What kinds of parallels or differences are there in both of their developments?

In this chapter, we try to answer those questions by observing parallel events in malware evolution, bearing in mind that the evolution in mobile malware developed in a much more intense manner. Moreover, we try to speculate on the next step of the evolution of mobile malware, not only what will possibly happen but also when.

3.2 Classification of Malware Related Events

Before we discuss the parallels between the desktop and the mobile malware world, we need to clarify methodological issues and ask the question: What is a parallel event in both worlds? So in this section, we briefly describe a taxonomy of malware-related events that allows us to define and characterize events according to different attributes. We then later use this taxonomy to relate events from the Desktop world to the "equivalent" ones in the smartphone world.

3.2.1 What is Malware?

Briefly speaking, malware is software with malicious functionality. In this general definition, the characterization of malware could not yet be quite clearly depicted, since the definition of malware depends heavily on user's expectation and experience. Meanwhile, there is no real technical/formal way to define malware. However, this does not mean that the definition of malware is purely subjective. Some forms of software are obviously malicious, such as virus, worm, while some are not, like backdoor or rootkit. At the same time, there is a large grey area. In this chapter, we try to focus on examples that are clearly malicious. We also focus on malware that is associated with economically motivated cybercrime, i.e., "mass malware" that targets many people, as opposed to malware for targeted attacks like Stuxnet and Duqu [16].

3.2.2 Attributes of Malware

We use three main attributes of malware to characterize and compare events in both areas: (1) propagation mechanism, (2) malicious payload, and (3) communication mechanism. Interestingly, these also reflect the evolution of malware in the desktop world.

Propagation Mechanism Historically, the first specific characteristic within the evolution of malware was the way in which it propagated. In general, the type of malware we are concerned with in this chapter wishes to infect as many computers as possible. Many different propagation mechanisms have been created. They can be broadly classified as autonomous and non-autonomous:

- Autonomously spreading malware needs a vulnerability on the target system, e.g. a weak password or a buffer overflow vulnerability. The attacker then exploits the target system to execute malicious code and finally install a copy of the malware onto the system [75]. For example, in the early days of the desktop device, standalone malware, like worms, try to infect as many computers as they can. They were able to spread rapidly through well-known but unpatched vulnerabilities in operating systems and web servers. Famous examples are the Morris Worm and Slammer which will be discussed below.
- In the non-autonomous propagation method, malware needs user interaction to spread. Thus, the spreading mechanism usually contains some aspect of social engineering. Non-autonomous spreading mechanisms became popular with the increased resilience of operating systems against network-based exploits. Today, emails, SMS, MMS and online social networks are indispensable for this propagation method to work.

Malicious Functionality After infecting a system, the primary goal of malware is to execute malicious functionalities on the infected system. This is often termed as *payload*. Historically, the first kind of malicious payload to the best of our knowledge implemented simple backdoor functionality. Later, diallers provided by Trojan horses that created revenue by calling premium-rate numbers became popular. Today, multi-purpose malware with configurable functionalities in the form of remote-controlled bots is state-of-the-art.

Communication With the advent of remote-controlled bots, communication with malware and between malware become increasingly important for coordinating malicious activities, but at the same time, such communication methods become an important gateway of detection and mitigation. While the Morris worm, for example, only transferred back the infection information in a very simple way, bots later used standard protocols like IRC for remote control. To prevent easy detection and mitigation, peer-to-peer structure evolved as well as proxy-based communication. In another evolution step, encryption was added.

Methodology In the following, we present several malware-related events — first for the desktop malware world and then for the mobile malware world. These events can be characterized by a single attribute or a combination of them. Moreover, only the

historically first known occurrence to the best of our knowledge will be presented. Also, we only inspect events that can be seen in both the desktop and the mobile field.

Basing our findings on publicly available literature, we try to select as many such events as possible. Ideally these were peer-reviewed publications, but for some events we needed to refer to non-reviewed information sources. In such cases we checked whether the information was consistent with other information sources such as blogs or whitepapers.

3.3 Desktop Malware Timeline

We now discuss the evolution of desktop malware with the help of the timeline in Fig. 3.1, which depicts relevant events.

The first example of malware that was found self-replicating in the wild was called *Creeper* that appeared in 1971 and is generally considered as a milestone in computing history [19][9][30]. According to our modern definition, Creeper can be considered the first worm, although it seemed to be only a proof-of-concept without malicious payload, and it only worked on a closed network configured to allow the trusted peers to access each others memory and copy executable code. No modern propagation technique was used.

Ten years later, in the year 1981, the first large outbreak of a computer virus called *Elk Cloner* caused concern among the informed public [13]. Elk Cloner was a boot sector virus written for the Apple II computer whose source code is still available online [83]. When a computer booted with the infected floppy disk, the virus would copy itself into the memory and execute itself. Then the boot sector would be infected by the malicious code and keep infecting other floppies.

In his Turing Award lecture 1984, Ken Thompson put forward a concept that we today call a *backdoor* [91]. He showed how easy it is to establish such a backdoor and how hard it is to remove it.

The first documented occurrence of a Trojan horse, a non-autonomous propagation malware disguised as normal software, which tries to gain privileges in order to access operating systems to play malicious functions, appeared in 1986. It masqueraded as the shareware word processor PC-Write [63], a popular word processor at that time. When running, the Trojan would perform like a real word processor, but at the same time, it will delete all user files.

The first malware that received mainstream media attention was the Morris worm that was launched on November 2, 1988 [66]. In a specific sense, the Morris worm did not contain a real malicious payload, however, it contained several remote exploits of known application vulnerabilities, e.g. it could abuse a stack overflow vulnerability in $fingerd$ library, which is without boundary-checking at that time. It then started a remote shell procedure, a weak password guessing mechanism, and a local information collector to compromise local PC and find other potential vulnerable nodes. It also had a network

connection function to return its instance infection status to its parent node. And its flawed mechanism of self-replication could cause excessive load and denial of service on compromised machines.

The year 1990 documents the advent of the first polymorphic malware *Chameleon* [13]. Its propagation or infection procedure is normal for a virus, but what makes it so extraordinary is its malicious payload, the revolutionary polymorphic function. The virus, written in the assembler language, would search and affix itself to a `.com` files after one of the infected files had been launched. It has the ability to choose different decryption algorithms, thus acquiring a different appearance in every running instance.

The year 1999 marks a turning point in the development of desktop malware: the beginning of the botnet era. IRC botnets *Pretty Park* showed up in the middle of 1999 [38]. Pretty Park was propagated by a successful social engineering method. When executed by unsuspicious users, the worm would email itself to all the addresses in victims' contact list every 30 minutes. After the infection, malicious payloads of the bots would perform destructive behaviors, and collect sensitive information on host machines, e.g. productkey, system root path, network connection username and password etc. It will open a backdoor for further malicious code execution or secondary injection. Furthermore, Pretty Park would join an IRC channel to receive command and control messages to coordinate malicious behaviors.

Botnets enabled a new effectiveness of desktop cybercrime. At the beginning of 2001, *Dialer.Trojan*, which could dial premium-rate-numbers, was first discovered by Symantec [88].

In 2002 the first botnet *Slapper* was found protecting its command-and-control (C&C) communication using proxies [8]. Slapper evolved from worm *Apache Scalper*. While previous classes of worms infect a computer and transform it into a cyber-crime web node that unanimously executes the same predefined procedure of malicious behaviours, the Slapper could assign different tasks for different nodes. During the propagation, Slapper tried to exploit OpenSSL vulnerability of remote hosts.

It used P2P network binded with port 2002 UDP to protect its C&C Channel, and provided reliability by using checksums, sequence numbers, and acknowledgment packets [39].

It is believed that Botnet *ZeuS* firstly appeared in the year 2007, which was originated from benign software package [65]. It is highly dynamically configurable, user-friendly and competitively priced, making it one of the favorite malware in the underground communities. *ZeuS* could be categorized as Backdoor, initiating a keylogger and man-in-the-middle attack to steal banking credentials for financial gain [49]. The bot establishes its communication channel by encrypting packages with RC4 algorithm [18]. Significantly, the emerge of *ZeuS* is a point where the PC malware and mobile malware converge, because the mobile malware *ZitMo* found in the wild arose as a complement to the PC based *ZeuS*. The detail would be introduced in the following part.

3.4 Mobile Malware Timeline

We now identify several events relevant to the evolution of mobile malware. We depict them in Fig. 3.1 (bottom).

The worm *Cabir* for Symbian was seemingly a proof-of-concept malware that never really spread in the wild [20]. It was published by the hacker group 29A on June 14, 2004 and marked the beginning of a quick malware evolution. The Cabir worm was encapsulated in a file named *caribe.sis* and exploited a vulnerability of Bluetooth [36]. However, the whole exploit process was not automatic and needed the potential victim to click the yes button. Using Bluetooth, the compromised mobile phone would scan the vulnerable device within a distance, and send transferring request. After the agreement of the user, the *.sis* file had to be clicked again to proceed the installation. Then, the *caribe.app* would be auto-activated, and keep searching for nearby devices.

The virus *Dust* for Windows Mobile by the same group 29A followed in August 2004 [14] [45]. Dust could propagate by multiple media, e.g. email, Bluetooth, removable memory, or synchronizing with computers. As simple as the first Computer virus, it would affix itself to the end of infected files in root directory, and modified the entry pointer. As the concept of the virus is only to proof that windows mobile is vulnerable under ZERO-Day exploit[37][45], no malicious payload is involved in Dust.

The malware *Mosquit* in 2004 was believed to be the first ever Trojan for smartphones [76][44]. Like the first Trojan in the PC world, which masqueraded itself to be a useful software for propagation, Mosquit was disguised as an illegal copy of a smartphone game which was posted on website or P2P networks. And the Malicious Payload was designed to send messages to premium-rate-numbers. To disinfect, the user could just uninstall the game[81]. Meanwhile, Mosquit is also the first Dialer in mobile malware history.

In the same year 2004, the backdoor *Brador* for Windows mobile emerged. It could provide network access and receive malicious command from a malware controller [14][45]. Being a traditional backdoor in a new platform, Brador could not propagate by itself. The user could only download it as an attachment of email, or from computer. However, the malicious functionality should be paid more attention. It will duplicate itself as svchost.exe in the mobile startup directory and start each time along with mobile devices[90]. An email containing the victim information would be sent back to the backdoor author after Brador had fully accessed the mobile device. Hence, that device would be attached to the attacking list, and port 2989 would be opened for further remote commands. Brador could perform multiple malicious behaviors, including executing remote command, uploading and downloading, or stealing messages of the victims.

Shortly after, in March 2005, *ComWar* appeared. It was the first successful and practical worm with malicious payloads which is spread by MMS and Bluetooth with multiple variants [76, 56]. In the late version, ComWar even could attach its code to the end of files. Since then, MMS becomes increasingly popular as an infection method for mobile malware. The ComWar worm was transferred between mobile as a .sis file with random

file names. During the propagation phase, like Cabir, ComWar would search nearby Bluetooth devices, and attempt to send .sis file, and send MMS to potential victims in the contact list. Once the Bluetooth or MMS receiver opened it, the worm would be installed and executed. After full infection, ComWar would start another propagation procedure to spread to new vulnerable devices.

In August 2008, the mobile worm called *PMCryptic* appeared [89]. It is of significance to our malware historical research. PMCryptic could spread through memory cards and disguise as file folders. When executed by miss clicking, it would create file *system.exe* and replace existing files in all directories while retaining original file names, afterwards make a copy of itself into the operating system. However, what makes it the milestone is its malicious payload, the first polymorphic scheme of the mobile malware. Since then, the malware implementing obfuscation techniques is prevalent in the mobile world. Filled with random data, and changing code during each transfer, the code of each copy of the worm is totally different, leaving only the data section unchanged.

The botnet era for mobile malware started in 2009. *Ikee.B* is the first malware that had a full-fledged C&C logic to communicate with a botmaster [69]. This botnet propagation was initiated by remote attackers, who then detect for the weak password 'alpine' in SSH service left by jailbroken running iPhones. It would scan designated IP ranges with port 22 for SSH services with a binary named sshd in the malware, and login as root after a positive response. Once the IP is located, attacker would remote login to the device and upload Ikee.B source code. The *inst* script of the malware had to be remotely invoked to install the source code. The *inst* script would create ID, setup self-propagation and C&C components, collect and upload sensitive information, open HTTP session to remote Botnet server, and change the default password. As the Botnet in PC, Ikee.B used C&C Channel to update clients and assign specific malicious tasks. An HTTP GET request is crafted by script *Duh* and sent by script *syslog*, in order to keep contact with server and receive new commands and updates, e.g. C&C server could push a DNS poisoning script to clients, to perform phishing site attack.

The malware *Andbot* emerged in 2010. Although it was a proof-of-concept smartphone malware in the laboratory, it communicates with proxy-based solutions [97].

After the prevalence of *ZeuS*, the emergence of a mobile malware named *ZitMo* [59] in 2010, which means ZeuS-in-the-Mobile, marked the convergence of mobile and PC malware evolution.

The victim would be redirected to an online-banking website spoofed by *ZeuS*, and asked to input his cellphone number. The account name and password entered will be intercepted by the malware authors. Then the mobile phone would receive a message with a 'security certificate' link. If the link is clicked, *ZitMo* would be downloaded and installed, to get the mTAN of the online banking system. The botmaster gets everything he needs to make and authenticate a cash transaction. Since then, the malware in mobile world and in PC world have converged as a joint attacking vector.

3.5 Relating both Timelines

Everything that happened yesterday in the PC world and will happen again in the mobile world tomorrow, just like a Groundhog Day. In Fig. 3.1 we have connected related events in the PC and mobile malware evolution with dotted lines. There is an obvious relation between both timelines in every evolution step. They follow the same sequential pattern: from the first worm, proof-of-concept though, to the first successful self-propagation worms showed up in both fields with fully functional exploit components, to the first botnet with very complete Command and Control logic, every development step of PC malware and mobile malware is almost the same.

However, there is one point of divergence related to the emergence of the dialer. It emerged at the early stage of the mobile malware development in 2004, but at the later stage of PC malware development in 2001, drawing a relatively close time lapse. This is because the it is much more convenient for the attacker to make money from the mobile phone than a PC.

The appearance of ZeuS is a significant point throughout the history of malware. It is the first time a malware could transcend the PC world and combine it with the mobile. Since then, the development of mobile malware caught up with and started to keep pace with the PC malware.

From the both of the time lines we can learn that the obfuscation techniques emerged at relative early period of the malware evolution process, and simultaneously at the approximately same developing stages. After that, those techniques are prevalent among the malware in both of the platform.

3.6 Conclusions

The purpose of digging into the dark history is to vision a brighter and clearer future.

After all this research, inevitably we would come to this question: Will the two developments synchronize at a certain point in the near future? Will the previous threats in the PC world repeat itself in the mobile world?

The answer to the two questions is probably yes. Firstly, the quantity of mobile banking transactions is on the rise, and still has very large growth potential. Secondly, according to the survey of TREND [62], financial crimes have not yet become a mainstream, but on a way since ZitMo. Thirdly, the increasing computing power of the smartphone blurs the boundary between mobile and PC in our daily life. Sometimes, a smartphone could be a more handful tool for some activities, e.g. surfing the Internet, checking Facebook, sending and receiving email, or even online banking. The launch of Ubuntu Phone, which merges those two together, would exacerbate this trend.

Although the ZeuS botnet is already prevalent in the PC, there is no similar malware designed for the mobile. In the future, there could be a mobile botnet targeting bank accounts, just like ZeuS, but only installed on smartphones.

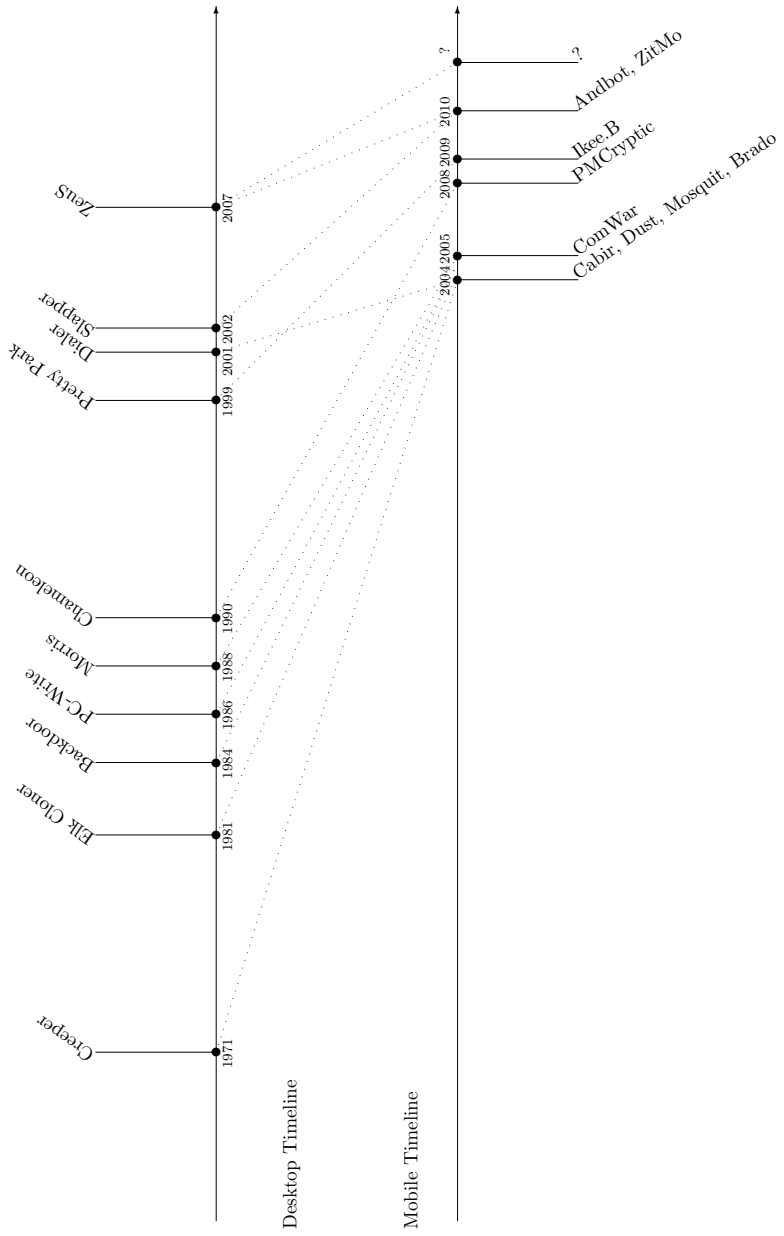

Figure 3.1: Malware Timeline

It works like a rootkit containing a keylogger hiding in a native library. It automatically detects online banking activities, e.g. bank websites, online banking applications, and tries to steal account numbers and passwords, transfers them back to dropzone, and intercepts mTAN.

The purpose of this section is to make a prediction about future threats. Then we have the chance to stop them before it is too late.

Chapter 4

The Obfuscation Framework

As introduced in the previous chapter, we approach practical obfuscation as a process to transform the target APK into a more complex form while not significantly slowing it down. We design a Python based framework not only to automize the obfuscation process, but also to carry out performance measurement and optimization simultaneously. In this chapter, we introduce the components and their working scheme in the framework, and more importantly the Obfuscation Management Layer which controls every work flow of the framework.

4.1 Introduction

The framework is designed to iteratively select and apply different obfuscation methods to the input APK programs, and then evaluate the complexity and similarity of the result APK programs. To implement different obfuscation strategies, it can integrate different algorithms into the Obfuscation Management Layer to guide the transformation process, and to compare the distinguished results. The algorithm will be introduced in the Chapter 6. The framework can be freely extended by adding new obfuscation methods, new software complexity metrics, and new algorithms, as we do in the following chapter.

The framework contains the following modules: the obfuscation engine Pandora [74], the complexity metric evaluator SSM, the similarity measurement toolset Androguard [34], the dynamic performance analysis tool Profiler [5], the GUI input generating and testing tool, the Android hardware device, the Preprocess and the Postprocess toolset, and the file system with the database.

The structure of the framework is depicted in Figure 4.1. To make the work flows of the framework easier to understand, we distinguish the connections between modules as data flows and control flows. In this Figure, the blue arrows represent data flows, e.g. the fetched APK files stored in the File System, the transfer of the log files from the Testing Tool to Profiler etc.. The black arrows represent control flows, such as the inquiry command from the Obfuscation Management Layer to the Database, or the copy paste command to the File System. Because the Obfuscation Management Layer, the Pandora, and the SSM cooperate more closely with each other than other modules, we integrate them as one central module in this figure. The Androguard is also integrated into this module. However, since the measured similarity values from the Androguard are only used as an index of the differences of the transformation results, and do not affect the obfuscation process, it is not explicitly represented in the figure. Its related working process will be introduced in Section 4.9.1.

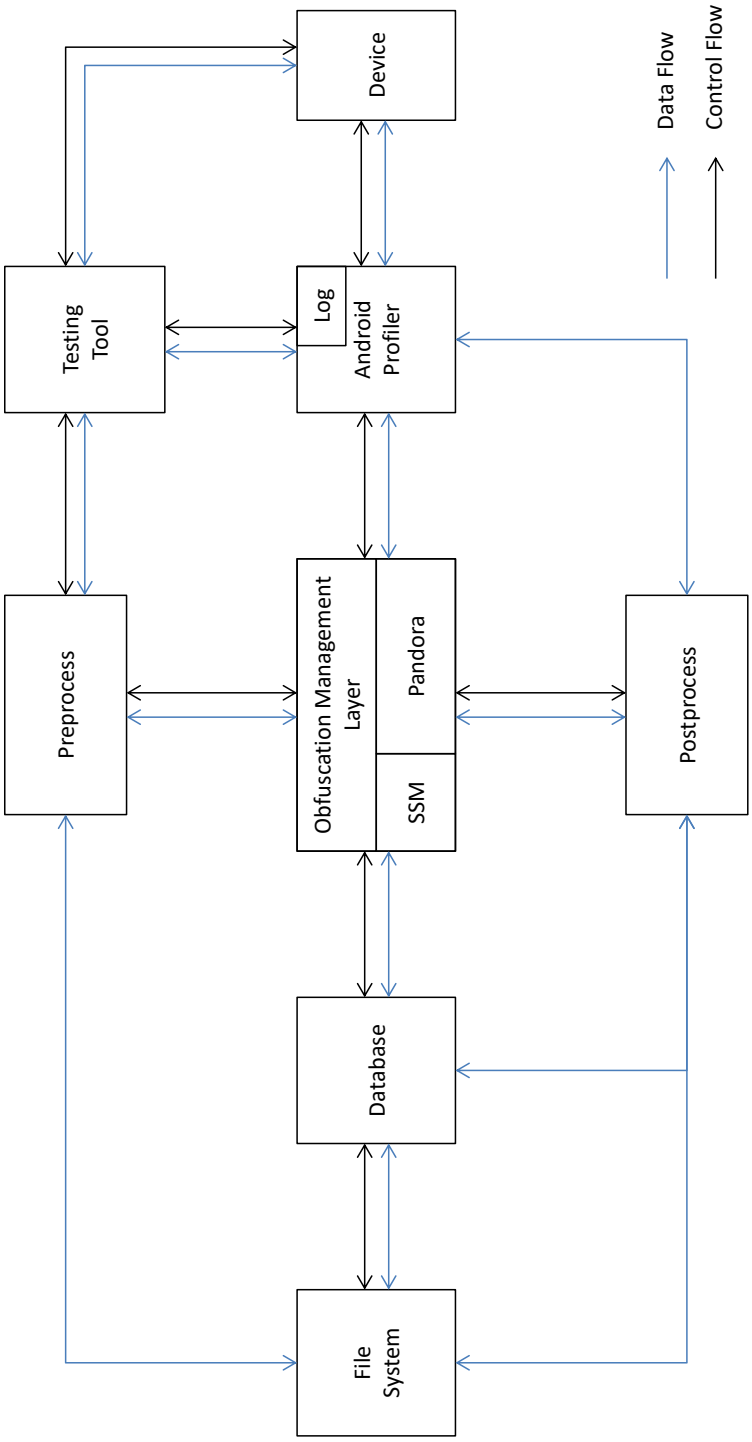

Figure 4.1: The Framework.

Now the working iteration of the framework will be briefly introduced.

In the beginning, the Obfuscation Management Layer (OML) will fetch target APK files according to the database information, and then push it into the Preprocess module. The Preprocess will unzip the APK into the Jar file and input it into the central module. At the same time, the APK is also input into the Testing Tool module to generate the testing cases log, if the APK's log file does not exist. The obfuscation starts when the central module receives the Jar file. According to the Database history, the OML selects a suitable obfuscation method and the obfuscation engine Pandora applies this method to this input Jar file. The SSM module measures the complexity metrics of the result Jar file, compares values before and after obfuscation. All the data are then recorded into the Database. The OML will judge if the result has reached our desired complexity, and decide whether next iteration should proceed. After that the result Jar file is transformed to *Dex* format and zipped into the APK by Postprocess module. The Profiler module implements performance test with the testing cases log for the two APKs to compare the performance penalty caused by obfuscation. The results are returned to the OML. All the files, logs, and data are stored into the File System and the Database.

4.2 Pandora

To make reverse engineering for Android application more difficult, Protsenko etl. [74] developed the Android obfuscation framework — Pandora (PANDORA: Applies Non-Deterministic Obfuscation Randomly to Android). It is a Soot [92, 93] based transformation system which operates on Android bytecode level, and aims at importing randomness as a factor to make transformation results as non-deterministic as possible.

Soot framework [92] includes four intermediate representations, the stack based low-level *Baf*, its Static Single Assignment form *Shimple*, the register based medium-level *Jimple* , and with its the aggregated version—*Grimp*. To transform the *Dalvik* bytecode, Pandora uses the *Jimple* representation. Because its strong typing and its limitation to 15 different 3-address based instructions, it is more convenient to analyze, optimize, and produce the result bytecode than the*Dalvik* bytecode which has untyped 218 instructions. The transformation of Pandora is implemented by the two extension modules of the Soot framework, the *BodyTransformer* and the *SceneTransformer* during its compile process. In this process, all the Object-Oriented elements are represented by the 15 Jimple instructions, and transformed locally and globally with the Jimple Transformation Pack (*jtp*) and Whole-Jimple Transformation Pack (*wjtp*).

For the sake of simplicity, we call "obfuscation transformation" as "transformtion" in this thesis.

In the following part, we demonstrate the theories and the implementations of these transformation methods that constitute the *BodyTransformer* and the *SceneTransformer* extension modules. Pandora has 11 obfuscation methods, including 6 data and control flows obfuscation (the Java method level obfuscation), 5 Object-Oriented obfuscations(the Java class level obfuscation), and an identifier renaming obfuscation. It can

apply an arbitrary subset of these obfuscation methods to the Android applications. Because the identifier renaming has no effect on the complexity metric, and its relation to the "difficulty" level against human revere engineering has already been elaborated in the work of Ceccato etl. [24, 25, 23], we emphasize the first two categories of transformations.

4.2.1 Data and Control Flow Transformation

This type of transformations is developed to obscure local and global variables, and therefore the operations on the basic data types and objects.

String Encryption This transformation provides two kinds of polyalphabetic string encryption algorithms, which replace each string with a serial of different characters. A diversity of substitution methods is provided to obscure the strings which are revealing key information, e.g. "Login Failed". This transformation method is flexible to integrate more encryption algorithms to enhance diversity. The decryption algorithms are additionally embedded into the source code of the target application. More effort for the reverse engineer is needed to revert embedded algorithms to the original strings.

- *Vigenère Encryption*: This algorithm improves *Caesar Cypher* [87]. It encrypts the target strings by determining the symbols in the strings with periodical shifting. It uses another string as the key for decryption.
- *Dynamic Key Encryption*: This algorithm is the upgrade of *Vigenère Encryption*. The shifting is defined by a series of randomly generated key k_i by the *Linear Congruential Method* [55], with the given initial key k_0.

Integer Encoding Besides the unencrypted Strings within the Java program, the integers in this program can also imply critical information that can assist reverse engineering. Therefore Pandora provides two algorithms, Split Modulo and Linear Encoding, to obscure the integers by adding extra *homomorphic* operations. In this way the decoding to the original values on the run is not necessary, so that the performance loss can be saved. The details of the two algorithms for the *homomorphic* encoding are described as follow:

Split Modulo

With any given integer, the $x \in I$ can be decomposed into the quotient and the remainder. With natural number m, exist a and b and that the following equations are accepted:

$$a = x/m, \quad b = x \ mod \ m$$

where the "/" denotes the integer division.

Then any two integer numbers $x_1, x_2 \in I$ can do the operations of addition, subtraction, multiplication, and comparison without using their original values [58]. We take addition for example. m values are randomly and locally (within this Java method) identical, the quotient and the remainder of x_1, x_2 are respectively (a_1, b_1) and (a_2, b_2) their addition result is:

$$a = a_1 + a_2 + (b_1 + b_2)/m,\ b = (b_1 + b_2) mod\ m$$

Then the addition result of x_1, x_2 can be restored dynamically only when necessary. Therefore all the integer and their calculations are protected through out the whole computing process.

Linear Encoding

The same as *Split Modulo*, *Linear Encoding* encodes the integer variables in the whole data flow, and decodes dynamically in JVM/Dalvik/Art. However, instead of encoding into the quotient and the remainder, it encodes the target variables x into z with linear transformation as the equation:

$$z = \alpha x + \beta, \alpha, \beta \in N$$

where the Natural Number α, β are randomly assigned. Thus the transformed integer variables can be computed without decoding and the therefor protected.

Array Index Shift Array is an important data structure in Java. This transformation maps all the elements in the array to a different sequence. It makes their index looks totally random, therefore obscuring the loop of the array. The application of this transformation is only confined to local arrays. Because if the whole array passes to another Java method as one parameter, the effect of obfuscation is limited.

Locals Composition This transformation composes the local variables, e.g. integers, chars, or strings, into containing objects. It is used to obscure the correlation among those independent and unrelated values, so that they appear to be semantically closely related. To compose the locals, a set of same type variables are chosen and compressed into the equal type of map or array, i.e. local integers to integer array, local strings to string array (or map). All those compressed variables are accessed by the indexes of the map or the array.

- *Compose to Array*: The length of the array will be $[0, n + 1]$, n is the number of compressed elements (local variables). The index $n + 1$ stores the number "n". The index of each element is randomly assigned.
- *Compose to Map*: The same type of the local variables and their generated indexes will be stored into one *java.util.HashMap* object. The type of the indexes (the key of the hash), which includes the integer, the char, or the string, is randomly selected. For each of the *HashMap*, only one type of the key is used.

4.2.2 Object-Oriented Design Transformation

The following obfuscation methods, which composite the other half of the Pandora system, are constituted by the *SceneTransformer* extension modules. The goal of the Object-Oriented Design obfuscation is to move, modify, replace, and integrate the elements within the Java objects, such as fields, methods, and classes, so that the appearance of those objects will be changed.

Drop Modifiers Drop modifier is an essential transformation for the total obfuscation process. It alters the access restriction for the classes and all their elements by replacing all the modifiers of the fields, methods, classes with *public* and removing *final*. All the other OOD obfuscation methods which modify the structures of the objects therefore can be applicable.

Extract Method Extract Method refactors the Java methods by outlining the whole body of the given methods. It generates a new Java method with the same arguments and the same method body, while replacing the whole body of the given method with a statement to invoke the new method and pass the arguments.

Extract Method is not only a obfuscation method, but also a preliminary preparation for further transformation. It is because in Java or Android, the signature of a set of particular method cannot be changed, e.g. the Java entry point *main* method, the Android *onClick* methods and the *onCreate* methods in the *MainActivity* class. To apply OOD obfuscation on those methods (with Merge Methods for example), method body outlining is necessary.

Encapsulate Fields Encapsulate Field creates a *getter* method to define the value of the target field and uses a *getter* method to extract this field. Therefore the target field can be encapsulated to avoid direct access, and allow modification and further obfuscation on that transformed field, i.e. the *getter* method. Consequently, Encapsulate Field will change software complexity of the structure and the appearance of the call graph in the target Object, and thus confuse similarity algorithms.

Move Fields Move Fields transfer the fields from one class to another. For the static field, the references to this field need to be updated. While for the instance field, since the base object (moved-out class) instance for this field is required, the moved-in class must create a new reference corresponding to the moved-out class, to initialize it before the moved field is used. Furthermore, the based object instance will be constructed only once and remain unchanged afterwords, until the life cycle of the moved-out class is finished.

Move Methods Similar to the Move Fields, the static methods are replaced by the invoking statements while the instance methods need the base objects. However, the synchronous life cycles of the moved-in and moved-out class instances are not necessary, because the reference to the moved-in class is available for the call sites.

As introduced before, some of the Java methods, e.g. *onClick*, *onCreate* which are called from outside of the program, or overloading of the abstract methods from interface or superclass, are not applicable for Move Methods.

Merge Methods The Merge Methods is to combine two methods in the same class which have the identical return types and modifiers. After merging, the bodies of two previous methods can be interleaved in the newly merged method, while the execution of different bodies are chosen by the *if...else...* condition. The parameters lists of the previous methods are attached together as one input list of the new method, along with the extra bool variable which controls the *if* condition.

4.3 Androsim

In order to spot and scale the identical, similar, and distinct methods between two Android *dex* files and thus the over all difference ratio, we select *Androsim* as our measurement tool. This tool is described in the publication authored by Desnos etl. [70], as built for measuring and evaluating the distance of bytecode similarity between two APKs to prevent plagiarism.

Androsim consists of similarity comparison algorithm *Normalized Compression Distance* (NCD) [27], that computes the distance between two series of strings which are compressed by snappy [3], and a module to translate the raw binary of *dex* file into a representation signature string, which is the input of that NCD algorithm.

In order to measure bytecode similarity between two *dex* files, the method signatures from each of the *Dex* file are generated by that translation module. This signature is a concise form of representation for the raw binary file, and is created based on Silvio Cesare grammar [26]. To generate this signature, the identical methods between the two *dex* are firstly excluded by comparing the checksum of the instruction sequence among all those methods. Secondly, the features of control flow graph, i.e. *BasicBlock*, *If*, *Goto*, *Return* etc., along with the API, and the Exceptions are represented sequentially by *chars* which are mapped from the dictionary of Silvio Cesare grammar [26]. Those generated char serials (strings) are the input of the following NCD algorithm.

The *Normalized Compression Distance* is defined as $d_{NCD}(A, B)$. A, B are two input strings. With the compressor C (snappy), the length of the compressed string are $L(C(A))$ and $L(C(B))$ (abbreviate as L_A and L_B). $A|B$ are the concatenation (join together and get rid of A's common string) of A and B:

$$d_{NCD}(A,B) = \frac{L_{A|B} - min(L_A, L_B)}{max(L_A, L_B)}$$

With this equation, the similarity of two methods can be computed. Thus the overall similarity of the two *dex* file.

4.4 SSM

For software vendors, quality of their products plays a key role for their market status. However, quality is not an index that can be directly assessed and intuitively exemplified, and it is hard to control during the development process. To improve and control the quality of program design and development, the software complexity metrics are used to model and quantify the quality. [53][103]

The software complexity metrics are not only used during software (re)engineering to measure and evaluate the quality of the programs, also they could indicate productivity [53], estimate and project reliability [54], improve maintainability [52], predict performance [32], and meanwhile reflect the difficulty of understanding the program.

In our research, we integrate *SSM* into our framework to measure the different software complexity metrics, for the static analysis of the obfuscation results.

With the input Jar file, the *SSM* works as follows: To measure the metrics, the *SSM* firstly profiles the relation of the classes and methods in the target Jar file with a set of Object-Oriented Design metrics, and then identifies the data or control flow within each Java method. Meanwhile, it generates the analysis graphs, e.g. the class inheritance diagram, the method call graphs, the control flow graphs etc.. After that it evaluates those output graphs and generates the results data. These data are later used to evaluate and characterize the obfuscation methods and then joined with the performance measurement to optimize the obfuscation results.

Same with *Pandora*, the SSM module is also developed based on the Soot framework, using *Jimple* as the intermediate representation (IR). *SSM* evaluates software complexity metrics in both the Object-Oriented Design aspect (Java class level) and the Control and Data Flow aspect (Java method level).

4.4.1 Method Level Metrics

Cyclomatic Complexity Control Flow Graph (CFG) is the abstract form of operation logic that represents the specific module (method) in software with an explicit entry node (*call*) and exit node (*return*). Each possible way of execution in this software module corresponds to a traverse path from entry node to exit node in the CFG. The different way of execution depends on the values of the predicates in condition switch statements like *if*, *while*, etc.. To fully test or understand this module in the software,

all the possible paths in this CFG need to be covered by giving different *bool* values to the predicates. Therefore, if there exist more possible execution paths in the CFG, it requires more efforts to totally test or understand this software module. Thus it would be more complex for reverse engineering.

To quantify this complexity, the total number of possible paths need to be calculated. In *Graphs Theory*, the CFG could be recognized as a strongly connected graph with a virtual directed edge added which points from exit node to entry node. The total possible paths of this modified CFG, which are recognized as the *fundamental cycles* in the *Graphs Theory*, are computed as $(e + 1) - n + 1$.

So, this Cyclomatic number formula, by simply adding one virtual edge, defines the metric *Cyclomatic Complexity* [95].

Dependency Degree DepDegree is the number of the directed edges in the dependency graph that models the data dependency relation among the operations (native instructions) in the method [17]. Therefore the dependency graph is the data-flow graph in the operations level. It is consisted of the nodes which represent every native code instruction in the method, and the directed edges which represent data flows or dependency relation between those nodes (instructions). The dependency relation means that, in data flow graph, the variable in node A is defined by the assignment or function call operations in node B. Thus, there is a directed edge pointing from node A to B.

For the whole program, DepDegree is the index of how much information has to be dealt with and how many different data states have to be processed during reverse engineering. For every single node, it represents the amount of the predecessor operations that need to be considered, which decide the current state of operation node. If there are more possible states for each node in the data-flow graph, it would be more complex to comprehend or optimize the program [17].

Lines of Codes The Lines of Code (abbreviated as LOC) is a simple widely-used metric for measuring the quantity of lines of text in the source code and the size of the program [98].

LOC was traditionally associated with programming errors in the predictive models. Early research [12] shows that bug density of the Fortran modules, which has less than 200 LOCs, is inversely correlated with the quantity of LOC. Another research done in the same period [82] also shows that, when the LOC is above 500, the bigger modules have a higher bug density regarding to Pascal, PLS and assemble language.

However, the LOC metric cannot indicate the real complexity of each code line [98]. Meanwhile, it loses the information of overall structure complexity of the programs (e.g. branches, loops). Therefore, LOC metric is usually combined with other metrics to provide more comprehensive measurements to the programs.

4.4.2 Class Level Metrics

- *Weighted Methods per Class* (WMC): the number of the methods defined in a class.
- *Depth of Inheritance Tree* (DIT): the number of classes along the path from the given class to the root in the inheritance tree.
- *Number of Children* (NOC): the number of direct subclasses.
- *Coupling Between the Object classes* (CBO): the number of coupled classes. Here, two classes are considered coupled, if one of them uses methods or instance variables of the other one.
- *Response Set for a Class* (RFC): the number of declared methods plus the number of methods the declared ones call.
- *Lack of Cohesion in Methods* (LCOM): Given a class C declaring methods $M_1, M_2, ..., M_n$ and a set of instance variables used by the method M_j denoted as I_j, define $P = \{(I_i, I_j)|I_i \bigcap I_j = \varnothing\}$ and $Q = \{(I_i, I_j)|I_i \bigcap I_j \neq \varnothing\}$. Then $LCOM = max\{|P| - |Q|, 0\}$. Intuitively, if the cohesiveness is low, the original class should be split into two or more classes.

4.5 Preprocess Module

For the preparation of the optimized obfuscation, the Preprocess module is used to fetch and process different files or data from the File System according to the requirement of the Obfuscation Management Layer. Before the obfuscation starts, firstly a new APK from File System is input into this module. It then unzips the new APK to extract the *Dex* file, and translates into the *Jar* file with the tool *dex2jar* [2]. If this APK has been transformed before, the *Jar* file will already exist. According to the records in the Database, the *Jar* file will be fetched directly from the File System. Those unzipping and translation steps will be bypassed. Finally, the *Jar* and the APK are all sent to the central module (includes the Pandora, the SSM, the Obfucation Management Layer, and the Androsim) to continue the transformation process. When the performance optimization parts are integrated in the framework, the Preprocess module will meanwhile start the Testing Tool module. It will then forward the APK into this module, wait for the result logs, and store it back to the File System which will be fetched at the disposal of the Obfuscation Management Layer.

4.6 Postprocess Module

When the obfuscation transformation is finished, the *Jar* files, the SSM and Profiler log files are input into this module. For the *Jar* file, it firstly will be optimized by the *Jasmin* [61], then translated into the *Dex* file, and finally zipped into APK. This APK file will be sent back to the Obfuscation Management Layer for comparing of the

APKs' similarity before and after obfuscation by *Androsim*. For the SSM log files, it will compute the sum up values of complexity metrics of this *Jar*. The complexity results are also sent back to OML to measure the differences. For the Android Profiler log files, it calculates the total *CPU* cycles of those applications from the logs. After that, the APK, the *Jar* files and the log files are stored into the File System, and all of the results data from the computations are updated into the database.

4.7 Automatic Black Box Testing

To improve the code coverage in this performance analysis, we develop a black box testing tool which can be widely adapted to all Android applications, automatically generating GUI behaviors on the run, and exploring as many APK functions as possible.

During the development process of this Android black box testing tool, there still exist several problems we need to pay attention to. Firstly, our testing tool is trying to use the greedy searching method to invoke all the functions in the APKs, which can cause infinite loops in the tests. For instance, button A in Android *view* M lead to *view* N, yet button B in *view* N vice versa. Secondly, the re-invoking of the functions will redundantly execute particular sections of codes. That can cause the recalculation of the CPU cycles during performance profiling.

To solve these problems, the *view* in the activity needs to be identified and a suitable traversing algorithm for the GUI tree has to be implemented. We use the *view* hash values to recognize every Android GUI *view*. And then we record the *view* status changes when our tool clicks each of the *clickable* UI elements (e.g. buttons, slides bars, and etc.), i.e., we can record in the log files which *view* this button belongs to and what is the next *view* when this button is clicked. We implement "Deep-Priority Search with One Step Back" algorithm in the testing tool, which can avoid *view* looping with the help of the hash values. When all the *clickable* UI elements have been traversed, the algorithm will terminate the testing.

4.7.1 The Implemented Tools

Because our black box testing tool can only acquire information of current *view* on the screen during runtime, the context of predecessor and successor *views* have to be established from the log file as the program runs, i.e. we record the changes of the views in the log file when a button is clicked, and then a directive traversing graph of the current APK GUI is constructed.

The "Deep-Priority Search with One Step Back" algorithm is iteratively launched to determine the strategy of selecting the next GUI element to construct this graph. In each of the iteration, firstly the connection is established from the host machine (the Testing Tool module to be exactly) to the Android testing device. Then the data stream of the current *view* on the screen is received by the Testing Tool module in the host

machine and interpreted into the tree data structure. With the help of the tree data structure and the traversing history log file, the algorithm is applied to select and invoke the next element to traverse, and record into the log.

In the above procedures, the connection, the data stream interpretation, and the invoking of the active or clickable elements are crucial. We implement the following tools for those functions.

1) *view server*: A service running on the Android device. Android GUI the *view* is inherently a tree data structure that formulates the screen layout on Android devices. The *view server* dynamically acquires the data of the current *view* each time it updates, and transforms them into the data stream. It is connected by the *socket* or the *telnet* from a local host machine running the debugging software (*solointerface* in our case). It transfers the stream data of *view* that end with "Dump -1". After that the data transfer is suspended until a new *view* layout shows on the screen.

2) *solointerface*: Our testing tool uses Apk View Tracer tool [4] to connect to the server in android devices. When the APK is running, it retrieves information about the view in the current screen. All the elements in that view and with their details (e.g. name, id, clickable or not, position etc.), are edited into an *node* object sets, and then compressed into the tree data structure. The connection is established by telnet or socket. When the view is changed, the view list will also be updated. The nodes will be processed by the Tracer module listed in the following section, to produce a directive graph.

3) *ViewTracer*: This module is the GUI input generator in our testing tool. It processes and records all the information passed by the solointerface, features and recognizes the current view or keys waiting to be clicked, and then call the *monkeyrunner* to make a click or other action on the device screen.

4.7.2 The Algorithm in the Testing Tool

Before starting the algorithm, we need to create 3 data structures, for the storage of information during runtime.

1. *UI_elements*: To identify the *views*, we construct the data structure *UI_elements*. Firstly, it hashes all the UI elements in each view with their id, coordinates and classname. And then, we used the hash numbers of the views as index paired with a list, which is composed by all of the clickable UI elements within that *view*. Each UI element contains the following information: [id, text, isClicked, coordinates, classname, locations]. The Boolean flag "isClicked" is the variable that records the traversing status of each UI element. If traversed, the "isClicked" is valued *True*, otherwise *False*.
2. *UI_hash_list*: It contains the hash values of each different UI interface which has appeared on the screen of the Android device. It is the list of index from *UI_elements*

3. *key_list*: It sequentially records the clicked UI elements (buttons) which will be output to the result log file. It contains information of each key's location, text, button id, the affiliation UI, the next UI after click. During the performance profiling process, the APK would have the same UI elements invoking order with this *key_list*.

```
1
2  Terminate = True
3
4  While(Terminate)
5
6          Extractor(current UI elements data) from view server
7
8          update key_list of the next UI
9
10         if current UI is new
11
12                 UI_hash_list.attach(hash of current UI)
13
14                 UI_elements.attach(pair of hash and objects list)
15
16         if current UI is not fully traversed
17
18                 for elements in current UI
19
20                         if element is clickable and not clicked:
21
22                                 generate click event
23
24                                 update in the elements list
25
26                                 break for
27
28                 continue While
29
30         else:
31
32                 if any key leads to the other not fully traversed
                       UI:
33
34                         click this key
35
36                         update key_list
37
38                         continue while
39
40                 else:
41
42                         press back button
43
```

```
44                          update key_list
45
46                          if all UI have been traversed:
47
48                                  Terminate = False
```

Listing 4.1: Overall Algorithm of Black Box Testing Tool

The algorithm is shown in the List 4.1. Before the main *while* loop begins in the algorithm, the loop breaking condition *Terminate* is valued as *True*, to keep the loops running until the termination condition is satisfied, which is when all the clickable UI elements in all the *views* are traversed.

After the *while* loop begins, firstly the function *Extractor()* receives the stream data from the *view service* which can provide information of the current GUI on the testing Android device.

Unclickable or invisible elements are filtered out from the stream data, which will be restored into the tree data structure (in *solointerface*). The tree nodes are the clickable *UI_element* objects and also contain all the necessary information of those *UI_element*.

To identify the *view*, we hash the useful information (e.g. the ID, the coordinates, the classname) of all the tree nodes as an index. If this *view* is new, the index hash value will be attached to the *UI_hash_list*. The pair of hash value and tree nodes (the clickable elements in a *view*) list is attached to the *UI_elements*. Those objects in the list all tagged the "isClicked" with *False* as introduced above.

After that, the algorithm starts to search for the traversing route. Firstly, it checks whether all of the tree nodes (the clickable elements) in this *view* are clicked. If not, it starts the *for* loop to iterate on those elements to find the unclicked element. Then GUI input action is generated from the *ViewTracer* to click on the element. Then the *UI_elements* is updated by valuing the "isClicked" of the selected element as *True*. Also the *key_list* (the result log file) is updated by adding this new element information to the list end, which is location, text, button id, the current *view* and the new *view*.

Because after the element is clicked, a new *view* may emerge on the screen. If so, the new iteration of algorithm has to be started. Thus the *while* loop executes *continue* after the click. And the new *view* in *key_list* will be updated at the beginning of the new *while* loop.

In the case of a fully traversed *view*, the algorithm will search in the *key_list*, to find all the particular elements in the *key_list* which direct to a different *view*. In the other *view*, the algorithm tries to find out if there still exist elements not traversed yet, by cross referencing the information of "isClicked" in the *UI_elements* list. If they exist, one particular element in the current *view* will be clicked to go to the new *view*. And then the algorithm replicates the procedure of updating the *key_list*, and continues on the *while* loop.

If the current *view* has been fully traversed and the other *view* which directly connects

to this *view* is also totally traversed, the back button is clicked. After updating the *key_list*, the algorithm will search whether all the elements have been traversed in this APK from the *UI_elements*. If true, the loop condition *Terminate* will be valued *False* to end the *while* loop.

Extract() module is composed by *solointerface()* and *uiProcess()* methods. The *solointerface()* uses telnet to connect to the service *view server* in Android devices. Each connection will return *view* stream data. *solointerface()* processes the data and revert them into the *view* tree in local host. And then the *uiProcess()* is used to iterate those tree nodes.

4.8 Android Profiler and Device

We implement the *Android Profiler* to measure the CPU time with which the target APK has to spend on the dynamic testing of the script from the *Testing Tool*. It composed by the *ViewTracer* and the native profiling module of Android OS. The *ViewTracer* module has already introduced in the Section 4.7. The natice profiling module will be invoked by the *adb* command line and run as a single profiling thread after the initialization of the APK process. The command line to initiate the process of the target APK and the profiling thread is listed as follow.

```
1 adb shell am start -n <COMPONENT> --start-profiler <FILE>
```

The $-n < COMPONENT >$ specifies the activity component of the target APK with package name prefix to invoke an explicit intent, such as the *com.caynax.alarmclock/.Mainactivity*, and the $< FILE >$ is the location where profiling log file stores. When this command is received by the *adb daemon* in the Android OS, the arguments *am start ...* will be passed to the *ActivityManagerService* daemon, which we have already introduced in Section 2.2.1. The process of the target APK is created by forking the *zygote* process when the $start - n < COMPONENT >$ message is received by the *ActivityManagerService* and communicate to the *zygote socket* as described in Section 2.2.2, while the profiling thread is already initialized in the *ART* vm by the *–start-profiler* (Subsection 2.2.2). Therefore the APK process starts along with the CPU time is already recording.

We employ the Samsung Galaxy S II (i9100) which implements the customized Android OS *Cyanogen OS 5.0* as the *Device* module.

4.9 Work flows of the Framework

After the introduction of modules, we now explain the work flows of the framework. We have three types of obfuscations work flows. Firstly, a number of APKs are obfuscated with the same transformation methods multiple times and then the attributes of those obfuscation methods, which are measured by the complexity metrics, are observed. This

is the base of our further research on the complexity or performance optimization. This part requires a simplified version of the framework without Obfuscation Management Layer (OML), and will be introduced in Chapter 5.

Secondly, certain algorithms in OML select the suitable transformation methods to guide the obfuscation process, to optimize the target APKs into the desired result complexity metrics. The algorithms will be introduced and compared in Chapter 6. In this part all the dynamic analysis modules, e.g. the Testing Tool, are not initialized.

Thirdly, while the target APKs are obfuscated to the desired complexity metrics, the performance of the result APKs are also optimized by the algorithm, to minimize the performance penalty imposed by obfuscations. The details of the profiling process will be introduced in the Chapter 6. All the modules are functional in this part.

4.9.1 Software Complexity Optimizations

The overall obfuscation process of the our framework is depicted in Figure 4.2. This section is the second type of work flows in the framework. As shown in this graph, each of the block colored black represents a functional module as introduced. The central module controlled by OML dominates the whole optimized obfuscation process. It initializes all the framework modules when receiving the obfuscation starting command and the designated complexity metric values. We will elaborate one iteration of the working process of OML independently, and the other accessory modules in the framework as a whole entirety will be introduced in the following subsections.

Obfuscation Management Layer (OML) The combination of the Pandora, SSM, and the Obfuscation Management Layer is called the central module of the framework. Because the control and data flow exchanges among those three modules are more regular and synchronous, the three modules could basically be recognized as one integrated entirety for the predigest of our framework work flows. This integrated module works as follows:

Firstly, when the designated complexity metric values assigned by the user and the previous iteration obfuscation are received, OML accepts the Jar file from the Preprocess module. After that, it starts the SSM module to measure a set of the complexity metrics of this Jar file. Meanwhile, OML sends a series of requests to the Database module. It then evaluates if the satisfaction of the target metrics should be expected in this iteration by any transformation method from Pandora.

We have two kinds of searching algorithms available to evaluate the result data returning from the Database in this case. Those two search algorithms, which will be thoroughly elaborated in Section 6.4, are based on either the mean values (the simple search) or statistics and probability models (the Naïve Bayesian Classifier).

On one hand, for the mean values based algorithm, OML will choose the transformation

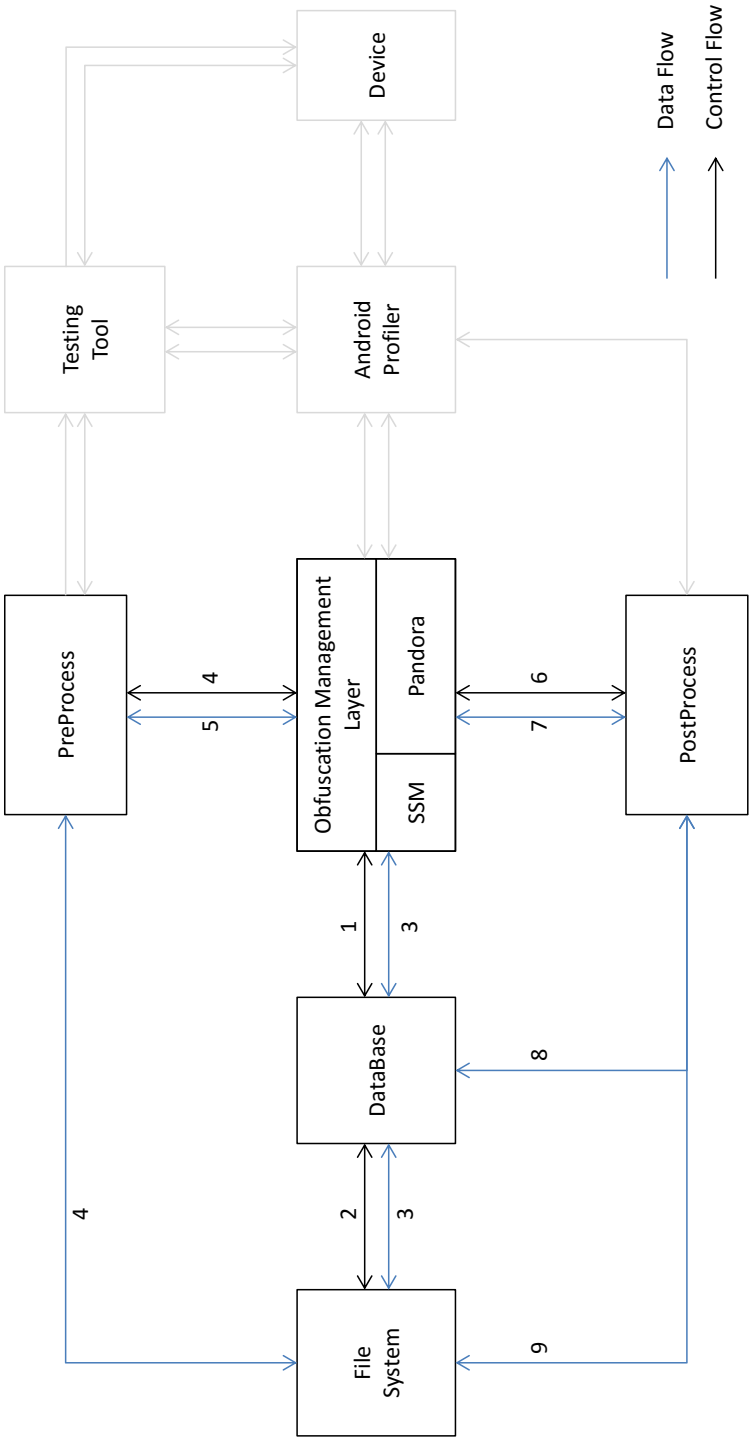

Figure 4.2: The Framework of Obfuscation with metric optimization.

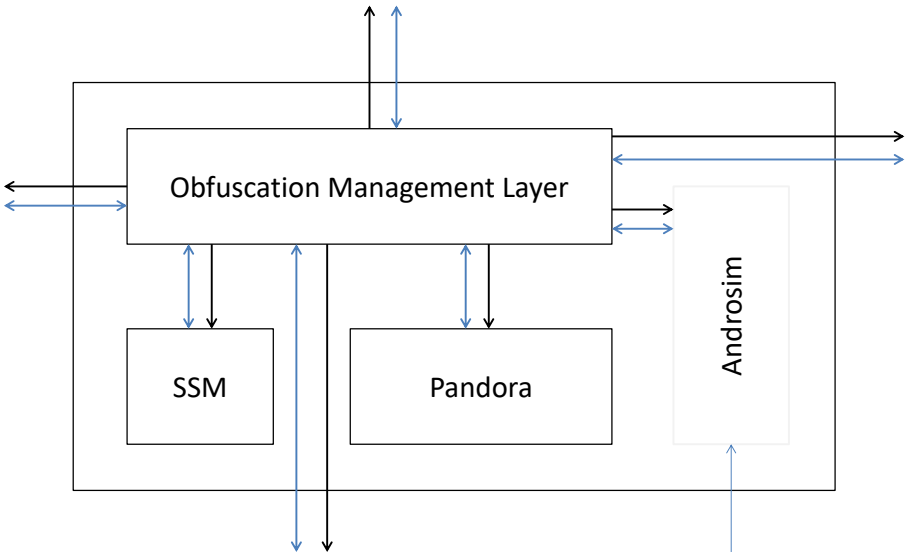

Figure 4.3: The Obfuscation Control Module.

method whose average metrics increase are the closest to the target values (in Euclidean distance).

On the other hand, for the statistics and probability models, OML will calculate the overall probability of reaching the target values, no matter what method it chooses. The details are shown in Section 6.4.3.

If this probability is low, OML will search the Database to calculate the metrics values which could possibly be reached at this obfuscation iteration. And then it substitutes the target metrics with the possible values. If the possibility is high, OML starts to calculate all the probability of the transformation methods for the target values, and then select the dominant method.

When the transformation method is determined, OML will forward the Jar file to the Pandora along with this method as an argument. Then the obfuscation process is started. The SSM module is used again directly after the transformation is applied. With the help of Postprocess module, the result metrics values are processed and compared to those before transformation (after Step 7 in Figure 4.2 by SSM).

Afterwards, OML starts to examine the results, to measure if the temporary metrics or the target metrics result are reached.

If the target metrics are satisfied, the total transformation process will be terminated after this iteration. If only the temporary values are reached, our framework will set

another iteration. That obfuscation result *Jar* file will be sent to Postprocess module for optimization, translated into *Dex* file, and then zipped into the APK afterwords. The procedures will be introduced in the next Section 4.9.1.

The new APK file is pushed back into OML to calculate the similarity ratio with the unobfuscated version of the APK file in the Androsim module. The Androsim in this central module is used to measure the APK file before and after the obfuscation, to observe if any differences have been made to the source codes during this transformation iteration. The *Normalized Compression Distance* algorithm used by Androsim is not a unique and accurate measurement of the differences between two strings, as introduced in Section 4.3. The result depends on the compression methods implemented. Thus, we only use Androsim result here as a reference, to check the amount of Java methods in the *Dex* file that have been modified or remain the same. The result data of Androsim is still writing into the *obfuscated_apks* table in the Database module.

Workflow of the Framework In this Figure 4.2, the rectangles colored gray mean the modules which are not functional in this complexity optimization section. The blue arrows are the routes to pass the data, e.g. required database items, APKs, Jars, log files etc., while the black arrows mean control flow which contain instructions to command and coordinate the other modules perform particular tasks.

To initialize the framework, the target complexity metric values, e.g. DepDegree and the Cyclomatic Complexity, which are set to increase 30% and 50% respectively ($Dep0.30 \wedge Cyc0.50$), are assigned to the target set of APKs and input into the command line interface of our framework.

When this initial instruction is received, the central module, which is under the control of OML, interprets this command as database inquire messages and then sends them to the Database module (Step 1 in Figure 4.2). The Database module searches in its table for the information of the APK in the target APKs set, i.e. name, stored directory etc..

According to this information, instructions are passed from OML to the File System (Step 2) to search for the input APK and data for the Preprocess module. In the following steps, our framework will start to transform this APK to the desired complexity metric values. When the above searching is finished, all the information, e.g. the previous SSM log files, is returned to OML (Step 3) for further support of transformation.

From the information, the framework would know if this APK had been transformed before. If so, the *Jar* file of the target APK is already stored in the File System and the metrics are measured. Then the *Jar* file will skip the Preprocess module, directly go to OML, and skip the SSM module before transformation as well. If not, the APK is pushed to Preprocess module (Step 4). In this module, the *Dex* file is extracted from the APK and translated into the *Jar* file. In this section, performance analysis is not involved. Thus, the Testing Tool module (colored gray) will only be initialized by this Preprocess module in the following section.

When the *Jar* and the APK are both input into OML (Step 5), the obfuscation process

inside the central module starts as elaborated in Section 4.9.1. During this process, OML will select the dominant obfuscation method , start transforming, and then compare the complexity metric values. If the transformation fails, OML will select another transformation method and replicate the above process.

To be able to process the transformed *Jar* file and the data logs from OML, the Postprocess module is then implemented (Step 6). It does not only optimize the *Jar* file with *Jasmin* [61], transforms to *Dex*, and zips into APK (when received from Step 7), it also extracts and processes the log files generated by the SSM module or the Android Profiler module implemented in Section 4.9.2 (colored gray).

To measure the NCD similarity ratio made by obfuscation (as introduced in Section 4.3), the Postprocess module will send the obfuscated APK back to central module for the Androsim comparison. The similarity data are only used in Chapter 5, to show different properties of the transformations.

For the received SSM logs(Step 7), it does not only calculate the total values of every metric in the target *Jar* file, it also calculates the metrics values for each class.

To evaluate the transformation result, the total metrics values are sent back to OML (Step 7), and meanwhile database is updated (Step 8). This updating process includes: The result information of the APK is stored into the *obfuscated_apks* table, which contains the directory, transformation method, its original version, with its similarity and all the other data from the *Androsim*; The complexity metric information of the APK is stored into the *method_metrics* and the *class_metrics* tables.

The transformed *Jar* file, the APK file, and the log files are then stored into its directory (Step 9) in the File System.

The above procedures are one iteration of the total obfuscation transformation process of an APK. The iteration continues until the target complexity metrics are satisfied in this APK. Afterwards, our framework moves to the next target in the APKs set.

4.9.2 Performance Optimization

In the context of software protection, we study how to automatically and practically obfuscate a given APK to a given target level of "difficulty" while not substantially "slowing down" the program. To be able to quantify the "difficulty", we measure it with the software complexity metrics in our research, and implement our obfuscation framework with the two optimization algorithms to reach the given target level. The working process is introduced in previous section.

According to Section 2.3.1, simply optimizing the software complexity metrics is not sufficient for such obfuscation. Performance penalty also needs to be defined, evaluated, and then optimized together with the other complexity metrics. In the following section, we introduce the working process of our performance evaluation and optimization.

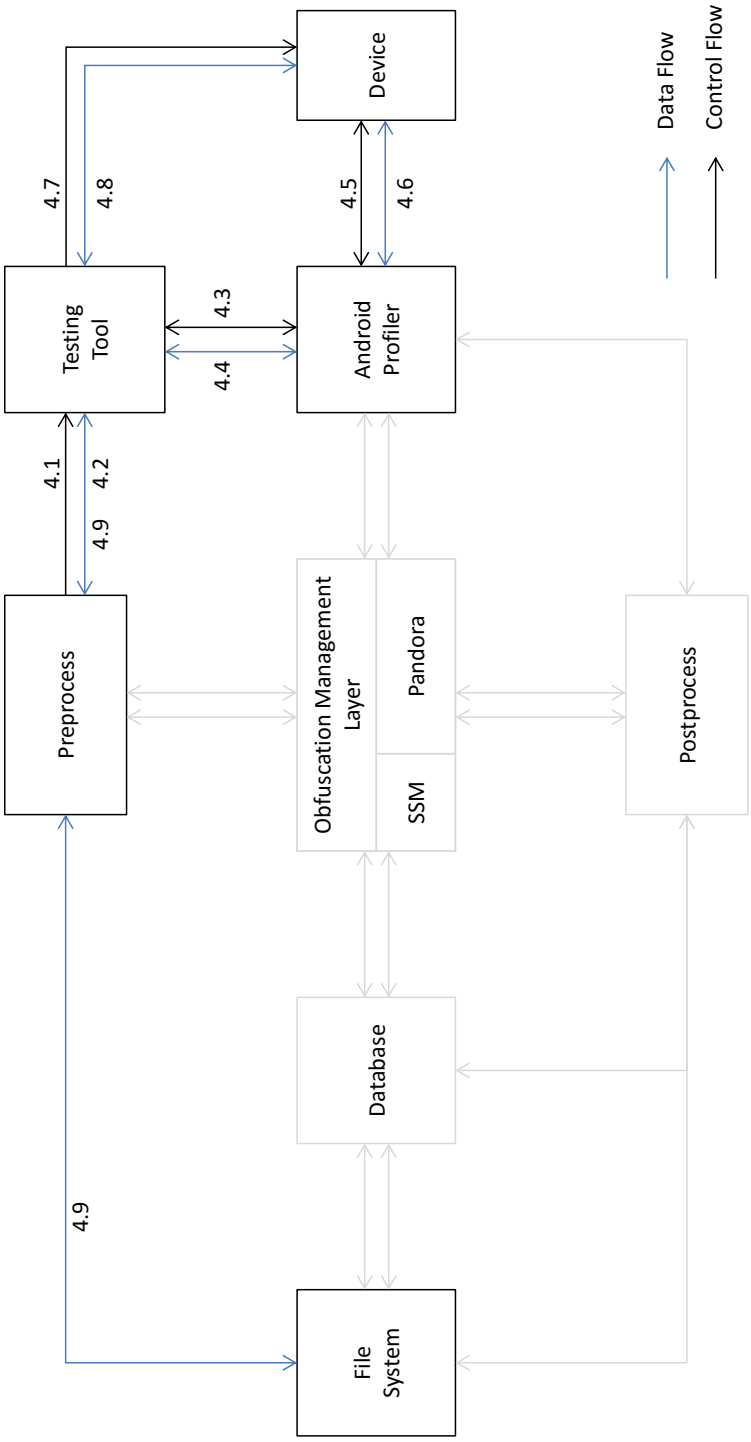

Figure 4.4: Implementing the Android UI Testing Tool.

Implementing the Android UI Testing Tool To evaluate the performance, the Android Profiler module initiates and launches the *MainActivity* of the target APK in the main thread of a *ART* virtual machine, and uses the profiling threads to log the CPU cycles of every Java method pushed on to or popped from the threads stacks. Launching the *MainActivity* alone however only covers a limited part of APKs' codes.

To have maximum code coverage, the Testing Tool module is implemented. Figure 4.4 shows the work flow of the Testing Tool module in the framework: When the Preprocess module receives the information of the target APK (Step 4) , hereafter called "new APK", the Testing Tool module will be started (Step 4.1) immediately after.

After receiving the new APK (Step 4.2), the Testing Tool module tries to install the new APK through the Android Profiler module (Step 4.4). In the process of Android *view* traversing, the Android Profiler does not record any performance log file. It only works as an initializing and launching module of the Android Device (Samsung i9100): Firstly, it tries to connect to *adb* (*Android Debug Bridge*) daemon of this device (Step 4.5). If successful, the new APK will then be installed on the Device (Step 4.6). After that the socket or telnet connection between the *Solointerface* server in the Testing Tool and the *ActivityManageService* (AMS) in the Device is established (Step 4.7). Then Testing Tool will receive the current *view* layout information (Step 4.8) after the APK is started with the command from it (Step 4.7).

The *view* traversing iterations as described in Section 4.7 starts from here. The data stream of the current *view* is transmitted by the *telnet* protocol until terminate mark "Dump -1", and reverted into a *view* tree data structure. After that, it selects one node, i.e. the clickable *view* element, from the tree data structure with the "Deep-Priority Search with One Steps Back" algorithm, clicks and logs this node. And then a new iteration starts.

The Testing Tool will judge the condition of termination during the loop, which is when all the clickable elements have been traversed. After termination, the APK file is uninstalled from the Device. This process produces a log file, which chronologically lists all the elements that have invoked actions and is stored into the File System (Step 4.9).

By clicking the *view* elements from the log file, the profiling process would have the same code coverage with the testing process.

Measuring the Performance with Profiler Figure 4.5 shows the performance measurement process for the target APK by our Android Profiler module. During obfuscation, performance is used as the standalone metric to be optimized by the algorithm. So the work flows of the framework are the same with Section 4.9.1 at the beginning (Step 1-7 in Figure 4.2). After the *Jar* file is translated and repackaged into the APK from the Postprocess module (Step 7), the performance measurement of the target APK starts.

To be able to start the profiling process, the Obfuscation Management Layer will command the Android Profiler module (Step 7.1) to initialize the Android Device (Step 7.2). In this initialization process, the Android Profiler firstly uses *adb shell* to confirm that

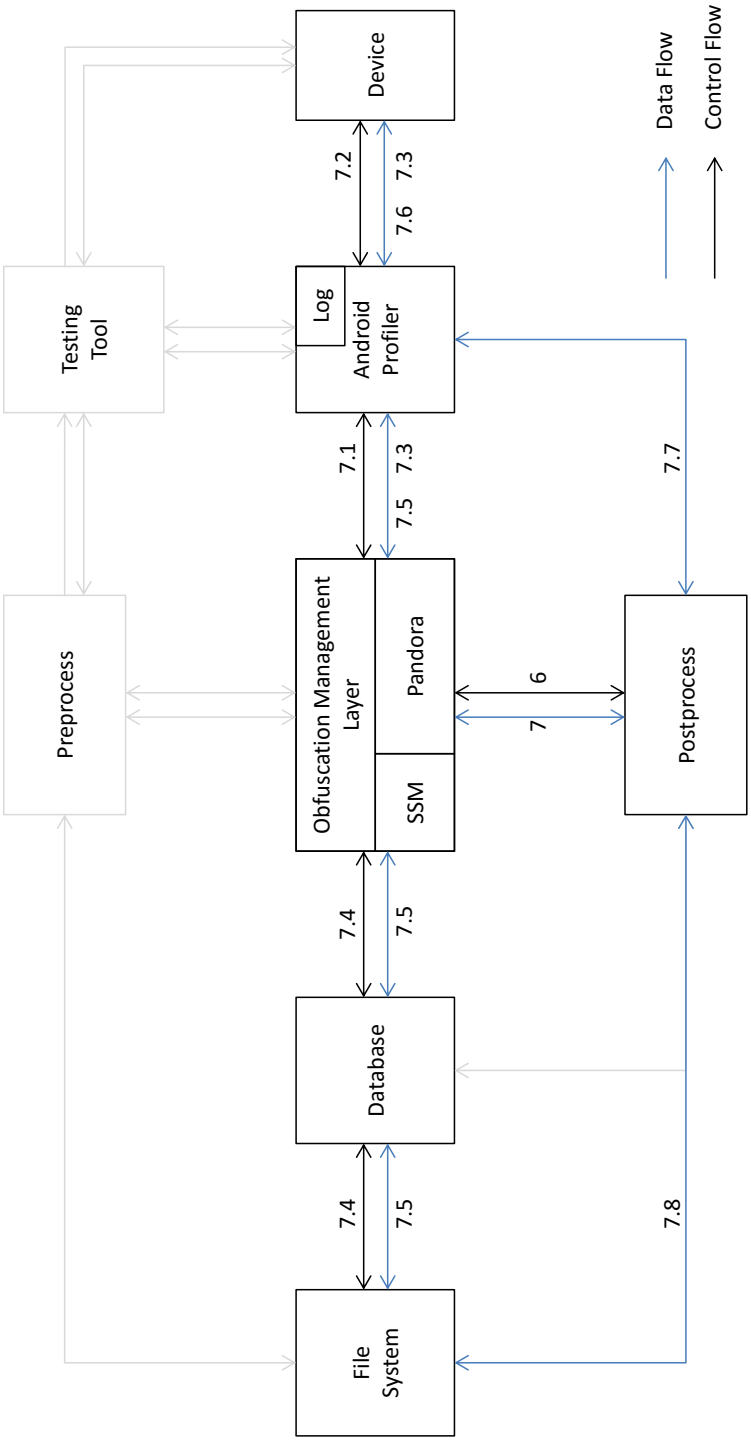

Figure 4.5: Implementing the Android Profiler.

the Device is connected at the port 5554. And then it sets the running environments in this Device. The environments include: The working mode of the CPU is set to *PERFORMANCE*; The operating frequency of the CPU is set to 1.2 GH; Permission of the *root* access from the *adb*.

When all those parameters are set, the obfuscated APK file will be fetched from the Obfuscation Management Layer by the Profiler (Step 7.3) and installed on the Device. If the APK is found already installed in the Device, it will be uninstalled first.

Because we already have the log from the Testing Tool, before the profiling, the log is fetched from the File System (Step 7.5), and used by the *ViewTracer* to generate the GUI input for the Device. The *ViewTracer* is a plugin of the Android Profiler module. To measure the performance of the target APK, the *activity manager* of the *adb shell* uses *–start–profiler* as an input argument from the command line (Step 7.6) of *adb* client. Thus, immediately after the starting of the APK's process, the profiler will be started as a single thread at the same time with the *MainActivity* thread.

After that, the Android Profiler module chronologically clicks the *view* elements one by one according the log file with a 3-second interval until the traverse is terminated. While the profiling is finished, the APK is stopped. The profiling log is extracted from the Device and processed by the Postprocess module (Step 7.7), and it will be compared to that of the unobfuscated APK with their sum up values of the CPU cycles.

Because the profiling could be affected by many runtime factors, the number of CPU cycles could vary in different runs of the same APK. Therefore, we repeat the profiling process 5 times with each of the APK and use their average values.

If the performance loss exceeds the limit set previously, the result will be abandoned. Another possible obfuscation method is selected and performed, and the above steps will be repeated.

Chapter 5

Evaluation on the Metric of Obfuscations

In this chapter, we investigate the problem of creating complex software obfuscation for mobile applications. We construct complex software obfuscation from sequentially applying simple software obfuscation methods. We formally define what it means to apply a sequence of obfuscation methods to a program and identify desirable and undesirable structural properties. For example, we identify *idempotency* and *monotony* as desirable properties of obfuscation functions.

We empirically evaluate 7 obfuscation methods with respect to 9 software complexity metrics on a set of 240 Android Packages (APKs). Following our research question, we restrict our investigation to properties inherent to a single obfuscation function, i.e., we only investigate iterative applications of the same obfuscation methods to a given program.

Finally, we show that most obfuscation methods exhibit "stable" properties when used iteratively, i.e., they are idempotent or monotonous. However, a single obfuscation method usually exhibits different stable properties with respect to different complexity metric, i.e., it might be idempotent regarding one metric and monotonous regarding a second metric.

In our research, we implement the *Pandora* [74], an obfuscation framework that contains a representative selection of obfuscating transformations specifically for Android. Its working theory and those composed obfuscation methods have already introduced in Chapter 4.

5.1 Introduction

As we have introduced in the Definition 1 and the Definition 2 of Section 2.3, an effective obfuscation transforms a program in such a way that

- the original program semantics are preserved (maybe with a negligible delay in performance) and
- the resulting program is harder to understand than the original one.

The problem of theoretical and practical definitions is to capture what it means for a program to be "harder to understand". While there exist some obfuscation methods that are provably hard to reverse [96, 21], the general understanding is that strong obfuscation

for general programs is impossible (for reasonable definitions of "strong") [11]. Despite theoretical advances of the field [41, 42], we must therefore continue to approximate the strength of practical obfuscation methods *empirically*.

Most practical methods are designed with an "idea" in mind of why the resulting program is harder to understand, but for most techniques there is no empirical evaluation, especially in comparison to other obfuscation methods. Emirically, the hardness to understand a piece of code can only be checked by human experiment [23] or (as an approximation) by using specific software complexity/quality metrics. In this chapter, we focus on software obfuscation for the Android platform and empirically evaluate the obfuscation techniques of *Pandora* [74].

The research question we investigate in this chapter is the following: *Considering the basic obfuscation techniques of* Pandora*, does obfuscation improve if we apply the same obfuscation technique multiple times?* Since obfuscation methods are usually applied only once to a piece of code, this might appear as a strange and unusual question. However, our aim is not primarily to build better obfuscation techniques but rather to *understand* the behavior of existing techniques better. Rephrasing the question, we ask: What are the characteristics of software if obfuscation methods are reused?

To answer our questions empirically, we have rebuilt the Android software obfuscation framework in this Chapter, which is the simplified version of our framework introduced in Chapter 4. It allows us to automate the task of obfuscation and software complexity measurement. In designing this simplified framework, we formalized the problem of building complex obfuscation methods from simpler ones. This allowed us to identify a set of desirable properties which practical program transformations should satisfy and to classify the investigated obfuscation techniques in this respect.

According to Collberg and Nagra [28], to measure the strength of practical obfuscation techniques, a definition that allows comparison between the *potency* of two transformations is required. Preda and Giacobazzi [72] classify a transformation as potent when there exists a property that is not preserved by the transformation. However, some properties of a program must be preserved since the obfuscated program should compute the same functionality. In practice, the definition of Preda and Giacobazzi [72] only allows comparison of simple transformations in isolated environments. To compare the strength of two obfuscated real-world programs, their framework cannot be applied.

Seminal work of Collberg et al. [29] surveyed different obfuscation techniques and classified them mainly into three categories: data obfuscations, control obfuscations, and layout obfuscations. They also investigated the effect of single obfuscation steps on different software metrics and even proposed a Java obfuscation framework (named Kava [29, Sect. 3]) designed to systematically obfuscate a program, such that certain quality criteria are satisfied. We are, however, not aware of any empirical evaluation of the framework.

Outline of the Chapter

This chapter is structured as follows: We firstly define desirable properties of obfuscation functions in Sect. 5.2. After giving an overview of our obfuscation framework in Sect. 5.3 we provide the results and a discussion in Sect. 5.4. Sect. 5.5 concludes this chapter.

5.2 Obfuscation as a Function

In this section, we formally define what it means to apply a sequence of obfuscation methods to a program and identify desirable and undesirable structural properties.

(Complex) Obfuscation Methods. We consider a finite set of obfuscation methods $\Omega = \{\omega_1, \omega_2, \ldots\}$. Each method is defined as a program transformation for a particular domain. Let $\mathcal{P}$ denote the set of all programs from that domain (e.g., all programs written in C). Then formally, every ω_i is defined as a function

$$\omega_i : \mathcal{P} \mapsto \mathcal{P}$$

such that $\omega_i(p)$ computes the same functionality as p without being exponentially slower or larger than p.

A complex obfuscation method can be defined as applying first a specific simple obfuscation method ω_i and then another simple obfuscation method ω_j. From the perspective of obfuscation as a function this is the *composition* of functions, i.e., $\omega_j(\omega_i(p))$.

We now define what it means to apply a *sequence* of obfuscation methods from Ω to a program. We denote by Ω^+ the set of all finite sequences of elements of Ω (including the empty sequence). An example sequence is $\langle \omega_1, \omega_2, \omega_1 \rangle$. For two sequences $\alpha, \beta \in \Omega^+$ we denote by $\alpha \cdot \beta$ the concatenation of α and β and by $\alpha \sqsubset \beta$ that α is a strict (i.e., shorter) prefix of β.

We now define the general notion of obfuscator composition $\mathcal{O}$ as using individual methods from Ω to create new (and possibly better) variants of obfuscation methods.

Definition 5. Complex obfuscator composition *is a function* $\mathcal{O} : \mathcal{P} \times \Omega^+ \mapsto \mathcal{P}$ *that satisfies the following conditions for all* $p \in \mathcal{P}$*:*

1. $\mathcal{O}(p, \langle\rangle) = p$
2. *For all* $\alpha \in \Omega^+$*:* $\mathcal{O}(p, \alpha \cdot \omega) = \mathcal{O}(\omega(p), \alpha)$

As an example, let $\alpha = \langle \omega_1, \omega_2, \omega_1, \omega_3 \rangle$. Then we have:

$$\mathcal{O}(p, \alpha) = \omega_1(\omega_2(\omega_1(\omega_3(p))))$$

Properties. We now define a set of properties that complex obfuscation methods should satisfy. These properties sometimes refer to software metrics such as those defined in Sect. 6.2. We formalize them as a finite set of of functions $M = \{m_1, m_2, \ldots\}$. Each metric is a function that maps a program into a totally ordered metrical space like natural numbers. For example, the metrical result of the simple complexity metric *lines of code* is a set of natural numbers and its ordering relation is $\leqslant$.

The first property of idempotency refers to the effect of an individual obfuscation function and intuitively states that applying the function more than once does not improve the obfuscation result regarding a particular metric.

Definition 6. *An obfuscation function ω is* idempotent with respect to metric $m \in M$ *iff for all $p \in \mathcal{P}$ holds that $m(\omega(\omega(p))) = m(\omega(p))$.*

It should be noted that some obfuscation methods such as *drop modifiers* are idempotent for all metrics since they satisfy the stronger property of $\omega(\omega(p)) = \omega(p)$. We call such methods simply *idempotent*. An obviously non-idempotent method is *extract methods*, since it creates a new method with every invocation.

We now define an additional property of obfuscation functions: monotony. Intuitively, monotonous obfuscation functions continuously increase (or decrease) the value of the considered metrics. Obviously, all idempotent obfuscators are monotonous. For non-monotonous obfuscators, there are metrics which are unstable, i.e., which rise and later decrease or vice versa.

Definition 7. *An obfuscation function ω* is monotonous with respect to metric $m \in M$ *if the following holds: Let α and β be sequences of ω: If $\alpha \sqsubset \beta$ then $m(\mathcal{O}(p, \alpha)) \leqslant m(\mathcal{O}(p, \beta))$, where $\leqslant$ is the order on the metrical space of m.*

While idempotency is rather easy to understand, monotony is more complex since obfuscation effect on a particular metric is not always clear. In general, we believe that the potent obfuscation methods should be either idempotent or at least monotonous for most metrics considered. Idempotency is a desirable property because it makes an obfuscation method easy to apply and it facilitates control of its effects. Monotony is positive, because it corresponds to the expectation that obfuscation methods make analysis (increasingly) "difficult".

We will evaluate the obfuscation methods presented in Section 6.2 and show that most of them satisfy these desirable properties.

5.3 An Android Obfuscation Framework

As introduced in Chapter 4, We have designed and implemented a framework (Fig. 4.1) with which we can automatically apply the combination of complex obfuscation methods to Android APKs and evaluate them. In this chapter, we only implement a simplified

version of the whole framework. Its functional parts are only based on three modules which we use to apply and evaluate obfuscation methods: The *Pandora*, the SSM, and the Androsim.

We now explain how APKs are processed in our framework. The starting point is a set of APKs stored in the *File System* (left of Fig. 4.1). In *Preprocess* module, The APK is transformed into a *jar* file that is subsequently processed by *Pandora* in *Obfuscation Management Layer* (*OML*). The resulting *jar* file, which is an obfuscated version of the original *jar* file, goes through the *Postprocess* module (e.g., it is checked whether it contains valid *JVM* code using the *jasmin* tool [61]). To distinguish the two files in the file system, we append the name of the applied obfuscation method to the original filename of the *jar* file ($*\_\omega.jar$, where ω is an identifier of the obfuscation method record in our database). After that, the *jar* file is turned into a *dex* file and compressed into an *Android Package* which is again stored in the file system. In the meantime, we use *Androsim* and *SSM* to compare and compute different metrics on the original and transformed *jar* files and APKs (see Fig. 4.1). All the generated data are inserted into a *Database*.

The above process refers to one step of the complex obfuscation composition of Def. 5. To apply multiple obfuscation techniques in sequence, the framework will perform the same iteration with another obfuscation techniques to the same APK. The selection of the input files as well as the iteration and type of obfuscation techniques applied on the APKs are totally automated.

5.4 Results

We applied the obfuscation transformations described in Sect. 6.2 to 240 APKs which we randomly selected from a set of more than 1000 APKs which we downloaded from the open source Android application market *F-Droid* [35]. With more time we would have chosen more APKs for the computation of our results but we consider 240 a large enough set to give our results significance.

Idempotent Transformations. As introductory example for a clearly idempotent transformation, we show two metrics (CBO and LCOM) of *drop modifiers* in Fig. 5.1. In these (and the following) graphs, the horizontal axis denotes the number of transformation iterations and the vertical axis denotes the percentage of the original APK's complexity or similarity. The red and green lines correspond to the given complexity metric and the similarity measured with Androsim respectively. The values of similarity serve as a reference and indicate the overall change of the program structure caused by obfuscation.[1]

Transformations applied on the intraprocedural level, namely *Compose Locals* and *Array Index Shift*, are idempotent with respect to all OOD metrics as expected. *Move Fields*

[1] In the following figures, we have scaled down the graphs to improve the visual "overview" impression with multiple graphs on one page. The caption repeats the method and metric for readability.

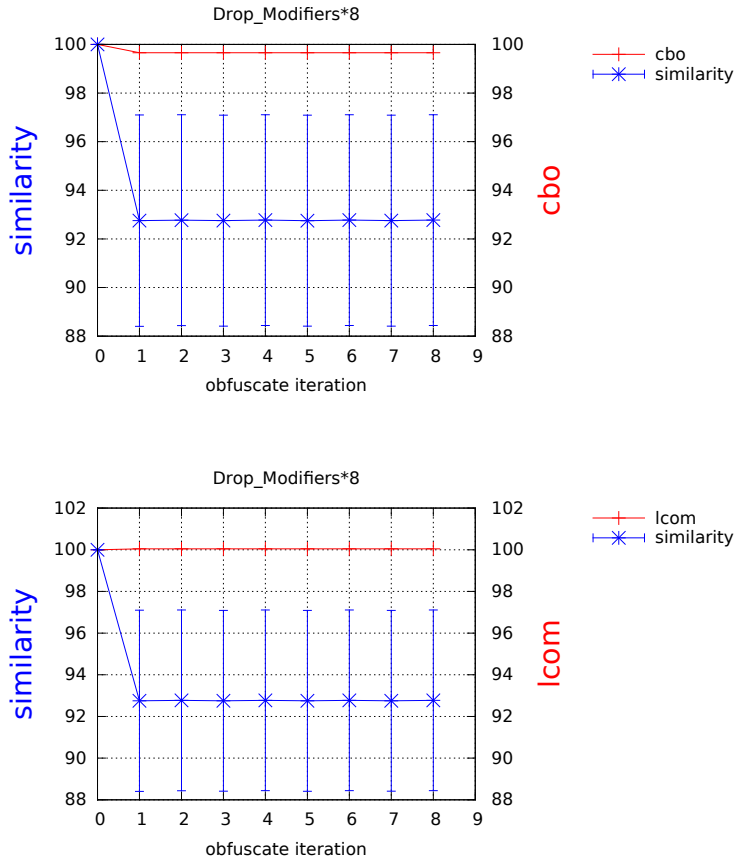

Figure 5.1: CBO and LCOM measurements of *drop modifiers*.

is idempotent for all metrics. *Move Methods* exhibits idempotency for Cyclomatic Complexity and WMC, since it neither changes the code of the methods nor their overall number.

Transformations applied on the intraprocedural level, namely *compose locals* and *array index shift*, are as expected idempotent with respect to all OOD metrics. This can be seen in Fig. 5.2.

Monotonous Transformations. Many of the evaluated obfuscation transformations were found to be monotonous, i.e., their continuous application keeps increasing some of the complexity metrics. In particular, the method-level obfuscations *compose locals* and *array index shift* are monotonous for the method-level metrics Cyclomatic Complexity, DepDegree and LOC (shown in Fig. 5.3). For *compose locals* the complexity growth was superlinear for all three metrics, whereas *array index shift* shows very slow linear increase reaching less than 101% of the original complexity after 8 transformations.

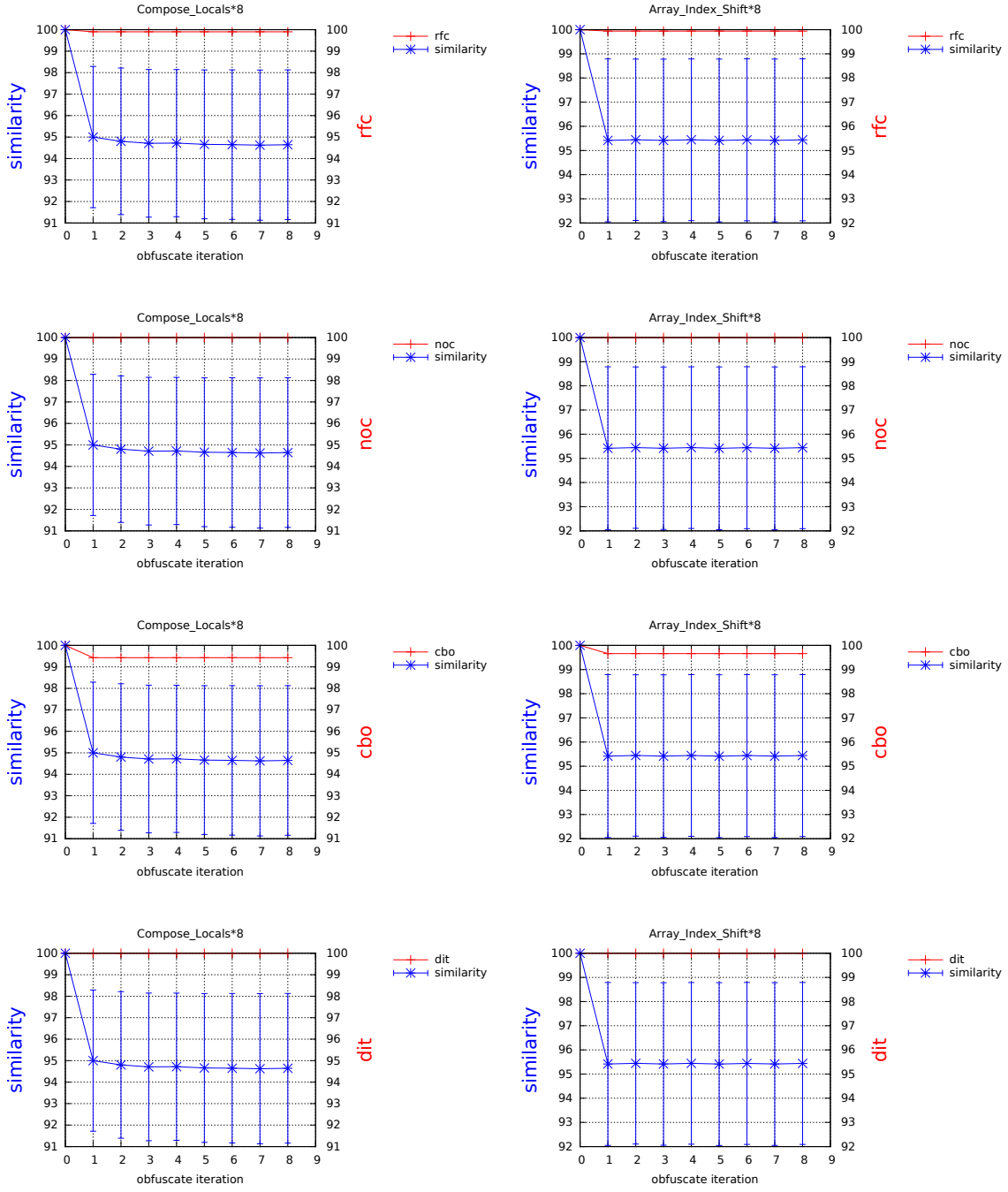

Figure 5.2: Selection of OOD metrics for *compose locals* (left) and *array index shift* (right).

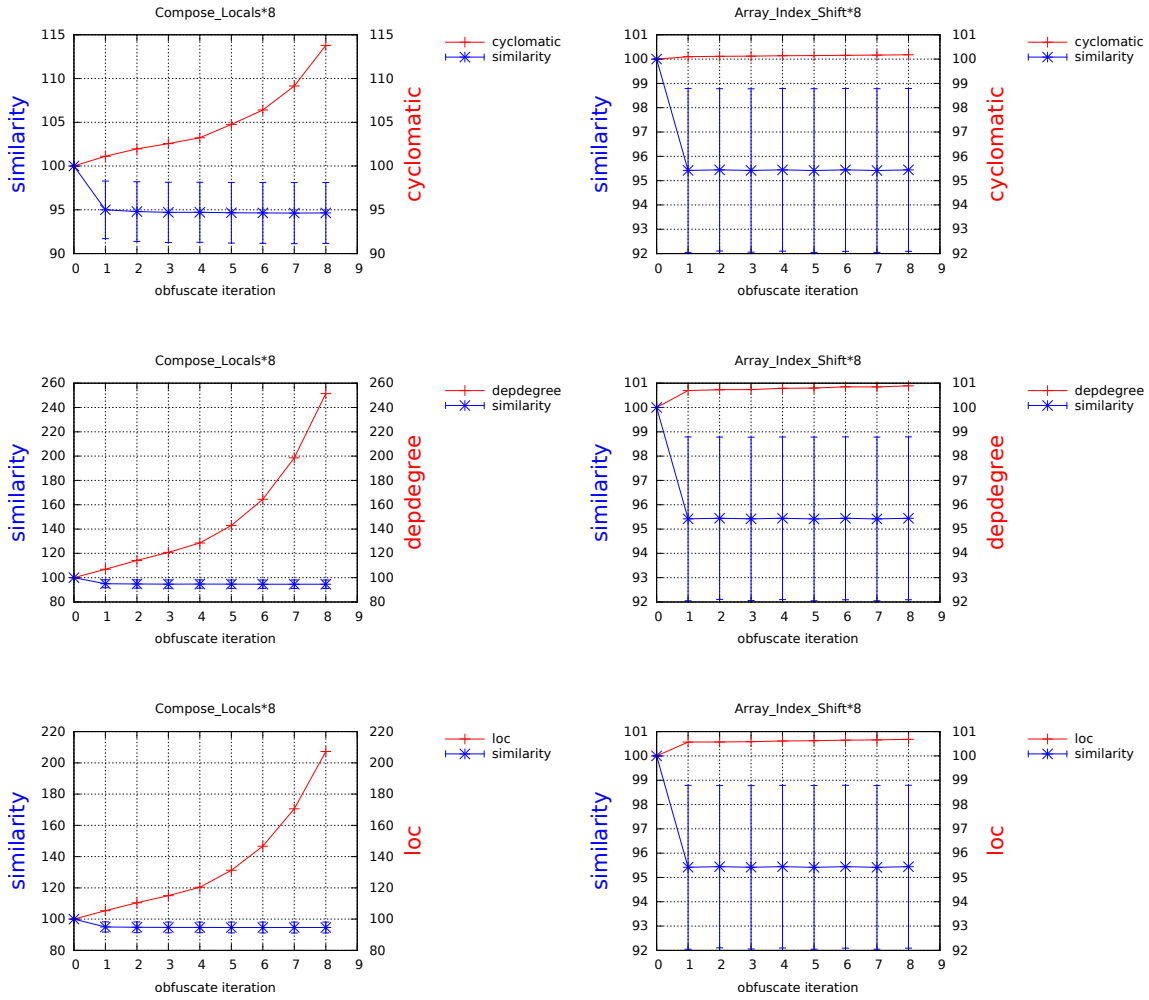

Figure 5.3: Cyclomatic Complexity, DepDegree and LOC metrics for *compose locals* (left) and *array index shift* (right).

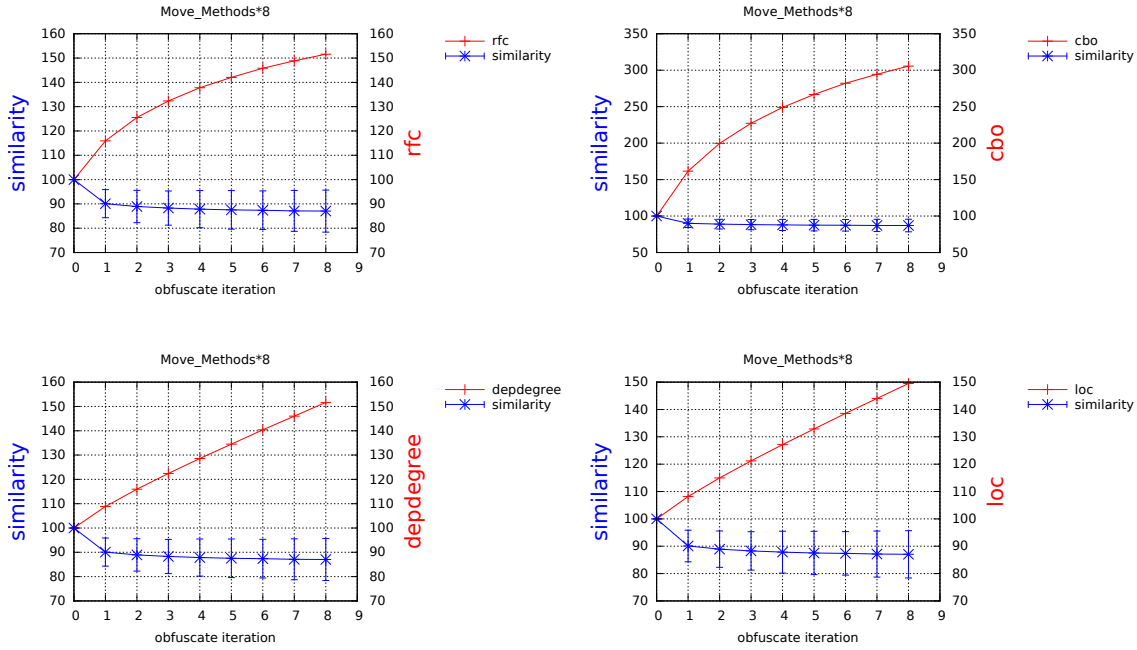

Figure 5.4: RFC, CBO, DepDegree and LOC metrics for *move methods*.

Move methods is monotonous for the OOD metrics RFC and CBO as well as DepDegree and LOC (in Fig. 5.4). As mentioned earlier, with respect to Cyclomatic Complexity, this transformation is idempotent. This is because moving non-static methods requires a reference object to the target class, which is stored in the class field and copied to the local variable before the method invocation. Since this does not add any branches, Cyclomatic Complexity stays unchanged. However, new instructions and variables are added, which increases the other two method-level metrics.

Merge methods is monotonous with respect to DepDegree and LOC: It adds new instructions and operations on variables. However, although it adds an additional branch per merge operation, the Cyclomatic Complexity (shown in Fig. 5.5) remains constant, since the number of circuits in the merged code equals to the sum of the circuits of the merged methods.

Unstable Transformations. Some of the obfuscation transformations did not fit the definitions of monotonicity or idempotency for certain metrics. These unstable transformations are particularly interesting. One example is *merge methods*. It exhibits monotonicity and idempotency for CBO and DepDegree, respectively, but is unstable in WMC, RFC, and LCOM (see Fig. 5.6, right). *Move methods* shows interesting unstable behavior with respect to the LCOM too: After decreasing within the first 3 obfuscation runs, LCOM increases again (see Fig. 5.6, left). The key to understanding this phenomenon lies in the randomness of the transformation and the nature of the metric: It should

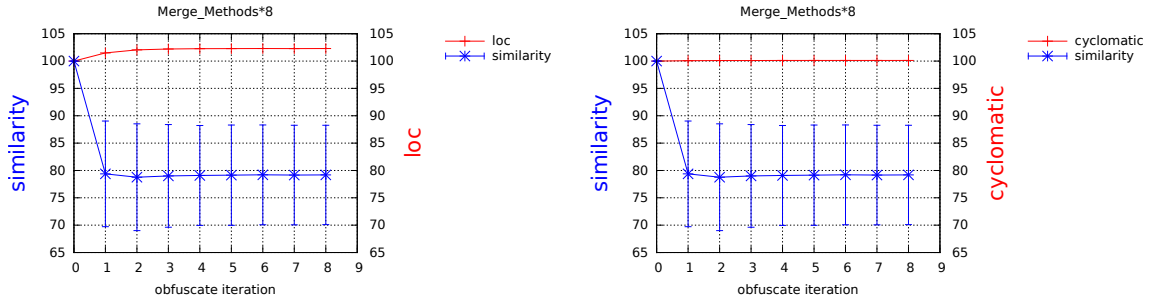

Figure 5.5: *Merge methods*: measurements for metrics LOC (left) and Cyclomatic Complexity (right).

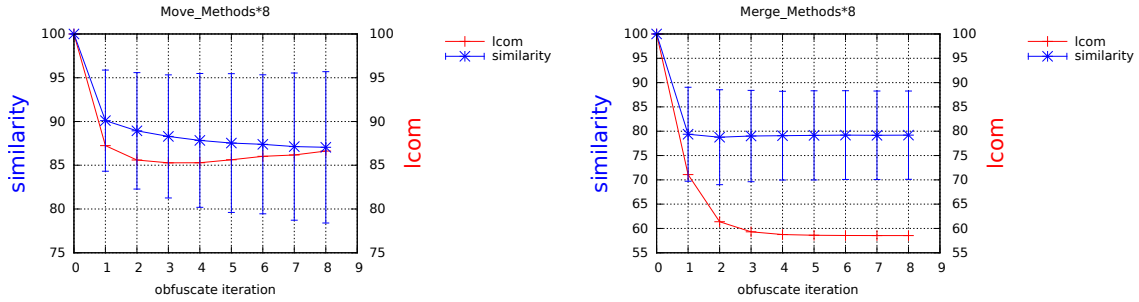

Figure 5.6: A comparison of LCOM for *move methods* (left) and *merge methods* (right).

be noted that LCOM increases with the number of class methods operating on different sets of instance variables and decreases with the number of those operating on the intersecting sets of instance variables. Since the movement process of *move methods* is randomized, a repeated application of the transformation can result in both more and less cohesive method layouts, therefore decreasing or increasing the metric.

5.5 Conclusions

In this chapter, we experimentally evaluated obfuscation methods when they are applied iteratively and we defined and revealed some structural properties of these methods regarding different software complexity metrics. While the results are interesting and show that most obfuscation methods we have used exhibit rather "stable" properties, the general picture is rather complex since a single obfuscation method usually exhibits different properties (i.e., monotonicity or idempotency) regarding different complexity metrics. Interestingly, a few obfuscation methods have unstable properties regarding

some of the metrics.

A more thorough understanding of the behavior of obfuscation methods is the basis for a more intelligent application of these methods in practice. In next chapter, following an idea of Collberg et al. [29], a detailed understanding of the effects of certain obfuscation methods on complexity metrics would allow us to transform programs in such a way that specific "target" complexity requirements are reached with a minimum number of obfuscation steps.

Chapter 6

Optimizing Obfuscation with the Complexity and Performance

In the context of software protection, we study how to automatically obfuscate a given program to a given target level of "difficulty" while not substantially "slowing down" the program. We measure difficulty with software complexity metrics, and evaluate performance with CPU cycles. We formalize the problem of optimal obfuscation and show that it differs from other informed search problems from the literature. Intuitively, this is because classical search algorithms assume that evaluating search candidates is cheap. In our case, constructing a search candidate (an obfuscated program) can take many minutes and so the number of generated programs should be kept to the minimum. Instead, the complexity values of candidates must be predicted based on a set of training measurements.

Within our framework for program obfuscation for Android APKs from Chapter 4 we empirically evaluate two different algorithms that search for an obfuscated version satisfying a conjunction of target complexity metric values. The first algorithm uses simple mean values to predict the complexity values and the second uses more refined probability calculations based on Bayes theorem. We show that the second algorithm outperforms the first.

We measure the performance overhead of the obfuscating transformations. By systematically evaluating obfuscation methods, software complexity metrics and performance overhead, for the first time we can give detailed insights into the inherent performance cost of obfuscation. For example, we show that some obfuscation methods (like String Encryption) are much more performance costly than others (like Compose Locals) to achieve the same metric value increase. We also found that especially for small metric value increases large performance benefits can be achieved if the right obfuscation techniques are used (i.e., those that are less costly in terms of performance).

6.1 Introduction

Software security of Android applications is especially susceptible (vulnerable) to malicious reverse engineer than the native code based software, because its Java bytecode is easier to decompile and to reconstruct the original Java source code. Therefore obfuscation is an essential criterion for the protection of Android applications. Meanwhile, the unpredicted performance loss will be caused by the obfuscation, which might seriously influence the user experience of the software. As introduced in the Definition 3

of Section 2.3.1 and the Definition 4 of Section 2.3.2, practical obfuscation techniques transform the original software into a potent form by preserving the original semantics while not substantially slowing it down. Therefore the obfuscation degree and the performance cost out of it require being optimized.

To optimize the obfuscation, we firstly need to define that, in practical obfuscation, what it means for a program to be "more difficult to understand" and being "not substantially slower" than the original.

6.1.1 Defining the Strength of Obfuscation Methods

Generally, an ideal definition of "difficulty" [10] not only refers to the consumption of time and resources by an attacker during reverse engineering (e.g., money, tools [6, 34], human labor), it also refers to the strengths of reversing methods, the knowledge and expertise of the attacker , which significantly affect the analysis processes. There is already some previous work that tries to measure the difficulty in a rather holistic way. For example, Cecatto et al. [23] perform experiments with human reverse engineers that have to understand the obfuscated code with the goal of comparing the effects of different obfuscations. An alternative way to define difficulty is to use software complexity metrics. While this approach is less holistic it is more adequate for automation.

Software complexity metrics are normally utilized for solving software engineering problems by evaluating and "measuring" software during the development process [33]. They are used as indicators for intricacy of structures, control and data flow, basically aiming at measuring how well or badly the program has been developed. When we obfuscate software, we assume that the structures of the code become more complex resulting in changes of at least some of the metrics.

6.1.2 Defining the Performance Cost of Obfuscation

To require that a program is "not substantially slower" than the original implies some notion of performance cost. There are many ways to define perfomance cost of software and all of them have different disadvantages [51]. Within the Android ecosystem we strive for a practical notion of performance that can be related to the user experience (i.e., the actual runtime) of an Android APK. This quest however is complicated by the fact that APKs usually are reactive and depend on user interaction to run. We therefore chose to use exclusive CPU cycles for all methods invoked by a "typical walkthrough" through the application. Performance increase is then measured by the increase in CPU cycles of the same walkthrough of the application before and after applying the obfuscation.

6.1.3 The Quest for Optimal Obfuscation

When difficulty and performance cost can be measured, as described above, these metrics can be used in an automated search for "optimal obfuscation", a problem posed by

Collberg et al. [29] in 1997, demanding a transformation algorithm that, given a program p and a set of complexity target values t as input, computes a variant p' that is semantically equivalent to p and has software complexity t.

The common approach to establish optimal obfuscation would be to systematically and sequentially apply a set of simple obfuscation techniques to the input program p until target t is reached. In previous Chapter 4, we have built such a obfuscation framework for Android APKs and have used it to study the properties of different obfuscation techniques in the Chapter 5. It can employ a total of 11 obfuscation methods (mainly from Pandora [74]) and 9 different software complexity metrics. We further extended this framework [101] to approximate optimal obfuscation. In all of these undertakings, however, performance cost was not taken into account.

Outline of the Chapter

This chapter is organized as follows: We give some background on the obfuscation framework, software complexity metrics and obfuscating transformations in Section 6.2. Then we formalize the problem of optimal software obfuscation in Section 6.3. In Section 6.4 we describe two algorithms to approximate optimal obfuscation with a brief comparison. In Section 6.5 we describe the problem of measuring performance overhead of Android applications and our approach to solve it. In Section 6.6 we use these two algorithms to compare the inherent performance costs of different obfuscation techniques and software complexity metrics. We conclude in Section 6.7 with a discussion of the results and ideas for future work.

6.2 Background

In this section, we briefly review our obfuscation framework, the obfuscation techniques and the software quality metrics it employs.

6.2.1 Known Dependencies between Obfuscation Methods and Complexity Metrics

When the same obfuscation technique is applied to a program multiple times in sequence, different effects on the values of software complexity metrics can be observed. For example, some obfuscation methods only take effect once, i.e., applying them multiple times has the same effect as applying them once. Such methods are called *idempotent* regarding that particular metric in Chapter 5. For example, most of the obfuscation methods above (except Move Methods) are idempotent regarding the metric Coupling Between the Object classes (CBO). Many obfuscation methods, however, are *monotonous* meaning that the metric increases with the number of times that the method is applied.

In general, idempotency and monotony are desirable properties of obfuscation methods because they enable predictability of metric changes. However, idempotent methods are much less helpful than monotonous ones when complexity metrics are sought to increase to large values. In the Section 6.4, we use all the metrics to evaluate all of the transformation methods, and train the algorithm with the evaluation results, and then use the results to make the framework generate optimized obfuscation results. Since hardly any of the object-oriented metrics have monotonous obfuscation methods, in the Section 6.6, we concentrate on the three method level complexity metrics DepDegree, LOC, and Cyclomatic. Furthermore, we restrict our attention to the 7 obfuscation methods that are monotonous regarding DepDegree, LOC and Cyclomatic, namely String Encryption, Integer Encoding, Compose Locals, Extract Methods, Move Methods, Encapsulate Fields, and Merge Methods.

6.2.2 Performance Cost

While trying to make software more complex, obfuscation also aims at not substantially slowing it down. Performance in comparison to the original program is therefore a central quality metric of obfuscated programs but it is difficult to define. We briefly review the general possibilities to measure performance here. We later show how we measured performance specifically of Android APKs in our experiments.

There are various ways to define the term *performance* in computer science. It can refer to the "speed" of a program or its resource efficiency [7], but it can also apply to a computer system as a whole or merely the software itself. For the complete system, the three most commonly used performance metrics are response time, throughput, and utilization [51] which can be measured by commonly available tools. For measuring the performance of (Android) software, software monitors [51] (also known as *profilers* [84]) are used. A profiler is able to identify the most frequently executed code fragments of the program and record the time (usually the CPU time) spent on each of examined parts of the code. There are different profilers which work on different levels all the way down to the machine instruction level [79]. For our measurements we use the *Android profiler* [5] that aims at measuring CPU time on the (Java) method level.

Profilers can be categorized into two classes based on their inner working mechanisms: they can be either event-driven or sampling-based. On the one hand, an *event-driven* profiler monitors specific events occurring in a system [60], e.g., the push or pop of the software stacks for example. Event-driven profiling is also often named *tracing*. Such profilers run by recording the time stamps when the control flow enters or exists each method. They extract the execution path of the classes and methods in a program which can also be used to determine code coverage [79]. Event-driven profilers capture all the time stamp records when methods execute no matter how small are they. Consequently, the performance overhead is relative high.

On the other hand, *sampling-based* profilers [51] is activated periodically after a user-defined time interval. When the profiler runs, the current state on the top of the stack

is recorded. It does not record every event occurring in the system, so overall there will be less computation overhead in sampling based profiling.

In our work, we try to measure software performance *after* obfuscation and ideally relate the performance of the new version to the old version. However, if *sampling-based* profilers are applied, "small" methods will be neglected, though the codes inside those methods may have been changed by obfuscation. Therefore, performance change before and after obfuscation transformation cannot be thoroughly measured by sampling.

6.3 Formalizing Optimal Obfuscation

We now define the problem of finding an "optimally obfuscated" program. AS defined in Chapter 5, we consider a finite set of obfuscation methods $\Omega = \{\omega_1, \omega_2, \ldots\}$. Each method is defined as a program transformation for a particular domain. Let $\mathcal{P}$ denote the set of all programs from that domain (e.g., all Andoid APKs). $\omega_i(p)$ computes the same functionality as p without being exponentially slower or larger than p.

From the perspective of obfuscation as a function, complex obfuscation process is the *composition* of obfuscation methods, e.g., $\omega_j(\omega_i(p))$. Consequently, the application of some $\alpha \in \Omega^+$ is then the result of recursively applying the elements of α to the program, where Ω^+ is the set of all finite sequences of elements of Ω. The details are described in Section 5.2.

We now formulate our task of finding an optimally obfuscated program as an optimization problem. We first formalize our software metrics as a finite vector $M = (m_1, m_2, \ldots)$. Each metric is a function that takes a program and maps it into a totally ordered metrical space like the natural or real numbers. The total score of a program is formalized as a function f that maps a program p to a vector of values that are resulted by applying each function from M to p, i.e., the vector $(m_1(p), m_2(p), \ldots)$. A *target vector* is a vector of desired values, one for each element of M.

Definition 8. *The* optimal obfuscation problem with respect to a program $p \in \mathcal{P}$ and a target vector t *is the problem to find an element of* $\gamma \in \Omega^+$ *such that* $||f(\mathcal{O}(p, \gamma)) - t||$ *is minimal.*

For simplicity, we assume that the norm $||.||$ is the Euclidian distance between vectors.

We now argue why solving this problem is different from existing problems and established search algorithms (like combinatorial optimization or informed A* search) cannot be used [78]. The problem in our context is that the task to generate and measure a new solution requires some effort (i.e., computing the obfuscated program and measuring its properties). Therefore, we require an algorithm that generates as little search candidates as possible. Since only available candidates can be measured, we attempt to *predict* the score of a program. Since the prediction will never exactly match the actual cost, we get a prediction error which makes the search problem even harder.

To summarize, we characterize our problem as searching the obfuscation space under uncertainty while creating only one search candidate per iteration of the search algorithm. Note that we can treat the performance of the program as an abstract software quality metric and can integrate the performance slowdown as a target value in our quest for optimal obfuscation.

6.4 Two Algorithms for Optimal Obfuscation

We first describe a very simple algorithm that serves as the baseline and then a more refined algorithm that exploits probability distributions for metric prediction. We call the first algorithm "simple search" and the second one "naïve Bayes search" since it uses Bayes theorem.

6.4.1 Simple Search with Mean Values

In our research, we have computed the changes in the given 9 complexity metrics when applying the given 11 obfuscation methods on a training set of 400 APKs which we gathered from F-Droid [35], an online open source repository for Android applications.

We apply every obfuscation method to those 400 APKs only once, to produce 11 sets of obfuscation results. Then we use the 9 complexity metrics respectively to measure the metric value of each APK in those 11 result sets. For each result set, we calculate its mean value of metric changes (in percentages) regarding different complexity metrics. Therefore, we can produce the result mean values Table 6.1 for every pair of method and metric, with which obfuscation methods forms rows and complexity metrics forms columns. For example, after the obfuscation method String Encryption has been applied to those 400 APKs, their mean changes of metric DepDegree is 0.0746 (in the second row and seventh column).

We then performed the search as follows: We iterated on a "working copy" of the program, the first working copy being the input program p. In every iteration, we predicted the values of the metrics in our target vector t based on the table of mean values. For each obfuscation method $\omega \in \Omega$, we could therefore estimate the effect that the application of ω would have on our working copy. From the set Ω we chose the method for which the estimate was closest to t (using the Euclidean distance), and then we applied the concrete obfuscation method to the working copy, resulting in a new working copy.

For example, if our target is to obfuscate an Android application in such a way that Cyclomatic Complexity increases by 60% and Dependency Degree increases by 80%, firstly the algorithm checks every obfuscation method on these two metrics in the table, predicts the Euclidean distances to the target metric values using the mean value, and then chooses the method for which the minimum distance is predicted. Then it calculates

Table 6.1: Mean values of metric changes after applying particular obfuscation methods ($n = 400$).

	RFC	NOC	CBO	DIT	WMC	LCOM	DepDegree	LOC	Cyclomatic
Const Extract	0.0036	0.0000	0.0119	0.0000	0.0013	0.0032	0.0394	0.0269	0.0010
String Encrypt	0.0018	0.0000	0.0021	0.0000	0.0013	0.0032	0.0746	0.0455	0.0211
Int Encoding	0.0018	0.0000	0.0021	0.0000	0.0013	0.0032	0.5885	0.5730	0.1528
Compose Locals	0.0019	0.0000	0.0023	0.0000	0.0014	0.0033	0.4705	0.3844	0.1950
Array Index Shift	0.0018	0.0000	0.0021	0.0000	0.0013	0.0032	-0.0053	-0.0047	0.0014
Drop Modifiers	0.0018	0.0000	0.0021	0.0000	0.0013	0.0032	-0.0057	-0.0050	0.0011
Extract Methods	0.2120	0.0000	0.0023	0.0000	0.3434	0.9228	0.0671	0.1016	0.1312
Move Fields	0.0056	0.0000	0.0274	0.0000	0.0012	0.0030	-0.0039	-0.0034	0.0010
Move Methods	0.0835	0.0000	0.3570	0.0000	0.0021	0.3311	0.0503	0.0436	0.0050
Encapsulate Fields	0.3607	0.0000	0.0021	0.0000	0.5863	3.1511	0.0731	0.1271	0.2042
Merge Methods	-0.0539	0.0000	0.0023	0.0000	-0.0715	-0.2126	0.0074	0.0017	0.0018

the increase of corresponding metrics of the transformed APK. It repeats the steps until the Euclidean distance to the target vector stops decreasing.

Overall, if the solution sequence of obfuscation methods α has the length n, then our algorithm generates only $n + 1$ search candidates. Obviously, the greedy strategy using the mean value has to be taken with care: It will find local optima anyway although it is not very accurate because most of the APKs deviate considerably from the mean values. However, we can use this algorithm as a benchmark and a baseline for further improvements.

6.4.2 Naïve Bayes Search

Our second algorithm uses more information than the mean value to make a more informed choice when selecting the next search candidate. When computing the mean values of previous measurements (see above), we found that the histogram of some transformation results looks like the normal distribution. Figure 6.1 shows an example histogram of the metric value distribution. The x-axis is the increase made by the obfuscation while y-axis shows the number of APKs that exhibit this increase.[1] The mean value is 0.4706 and the peak value of APK numbers are also very close to this value. This observation was similar for other metrics. So overall we studied the question, how we could improve the simple algorithm if we can assume that the changes introduced by the transformations follow a normal (Gaussian) distribution.

We decided to use the statistical classification method, the Naïve Bayes Classifier. It can categorize a particular result with a minimum probability of misclassification. The assumption of our algorithm is that all the values in the target vector t (i.e., the values of individual software quality metrics) are considered conditionally independent from each other when given a obfuscation method C_i in the *obfuscation set* $\boldsymbol{C}$ (defined below), which means that the values of all the metrics are considered to contribute independently to the possibility of choosing the transformation C_i, regardless the possible correlations among them.

We now formulate our problem in terms of Bayes' classifier: $\boldsymbol{X} = \{x_1, x_2, \ldots, x_n\}$ is the metric vector, an array of attributes composed by updated values of n metric measurements. In Bayes' terminology, these are termed the "evidence", i.e., the observed result value of the software metric measurements. Furthermore, H is a hypothesis stating that the vector X belongs to (is caused by) one element in class set $C = \{C_1, C_2, \ldots, C_m\}$, where C_i means $\omega_i(p)$ ($\omega_i \in \Omega$). Our search problem can now be phrased as follows: If we want to achieve evidence X (e.g., metric x_1 to increase by 30%), the element C_i in the obfuscation set should be chosen. It means that we should use the method that maximizes the conditional probability $P(H|X)$, i.e., the probability of using C_i (H is true) given X.

[1] Note that we eliminated any source of randomness from the obfuscation methods of Pandora before executing the experiments.

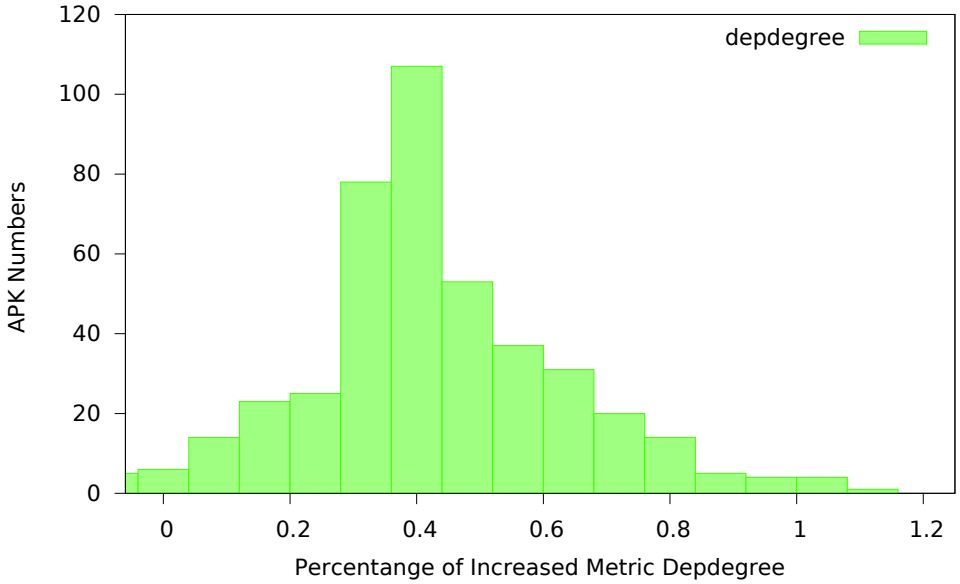

Figure 6.1: Distributions of Dependency Degree of Compose Locals for 430 APKs. The scale of the x-axis is the times the metric has increased.

Now Bayes' theorem allows us to calculate

$$P(\boldsymbol{H}|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|\boldsymbol{H})P(\boldsymbol{H})}{P(\boldsymbol{X})} \tag{6.1}$$

where $P(H|X)$ is the posteriori probability of H to be true under the condition X. In our case, it means that if some metrics has to be increased to a desired percentage, how large is the probability in case we choose the method ω_i. Here, $P(X)$ is the a priori probability independent from H, which means the total probability of the obfuscation result (i.e., the total probability to reach the desired metric value) is the true independent of the obfuscation method we use. Similarly, $P(X|H)$ is the a posteriori probability of reaching our desired metric values in case of choosing obfuscation method ω_i. The value $P(H)$ means the probability of choosing this obfuscation method. Since $P(X)$, $P(H)$ and $P(X|H)$ can be computed from the training set, the value of $P(H|X)$ can be calculated.

For example, we want to know the probability of using the obfuscation method *Extract Methods* to reach the goal of increasing metric DepDegree by 30%. This is expressed as the Hypothesis H: "$X = \{Dep(0.30)\}$ caused by *Extract Methods*". Firstly, we need to calculate the overall probability of reaching this goal regardless of the obfuscation method chosen, which is the total probability $P(X)$. Secondly, the we calculate $P(X|H)$, i.e., the probability of DepDegree increasing by 30% under the condition of *Extract Methods*. After multiplying it with the probability of choosing *Extract Methods* (which is $P(H)$) we have the desired result $P(H|X)$.

We now describe a single iteration of the algorithm in detail. As mentioned above, we

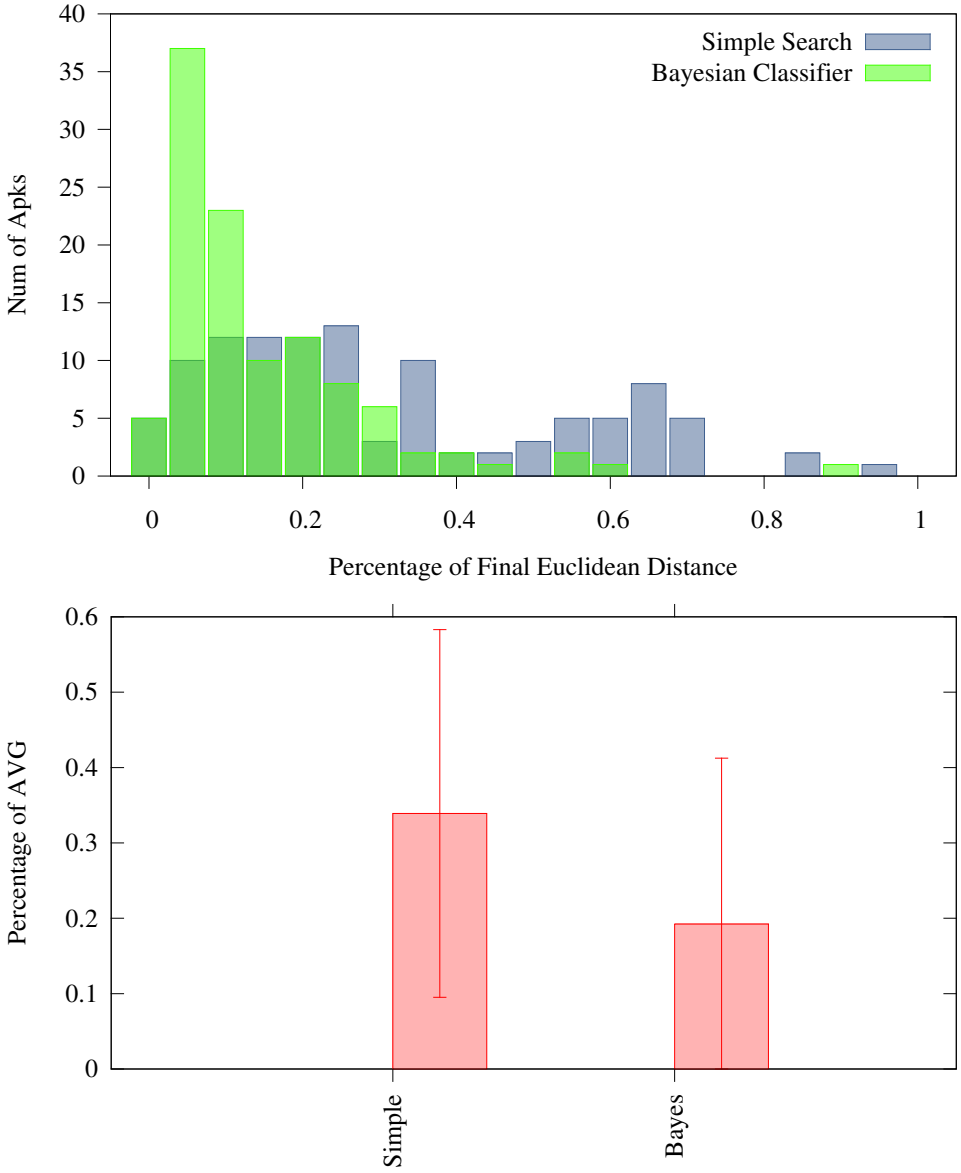

Figure 6.2: Performance comparison between simple search ("mean") and naïve Bayes search ("nbc") algorithms: distance (above) and mean of the final Euclidean distance from target with standard deviation (below).

used a training set of more than 400 Android applications to measure the correlation between the target metrics and the obfuscation methods in this algorithm, i.e., we have measurements $\{x_1, x_2, \ldots, x_i|\omega_j(p)\}$ meaning the changes of metrics $(x_1, x_2, \ldots, x_i)$ after $\omega_j(p)$ have been applied to program p from the training set. These measurements describe statistically the metric increase caused by each (obfuscation) method. Our algorithm takes this training set and the current working copy of program as input and outputs the "next best" obfuscated program:

1. The target is to find the dominance probability of the desired metric increase under a certain obfuscation method, which is $P(C_i|X)$. It is also the maximum posteriori hypothesis resulted from the training set. The algorithm categorizes the tuple X into the class which has the highest posteriori probability, i.e., Naïve Bayes Classifies category X into C_i, iff

   $$P(C_i|X) > P(C_j|X) \quad 1 \leqslant j \leqslant m, j \neq i$$

   For example, if $X = \{Dep(0.30) \wedge Cyc(0.20)\}$ (meaning our goal is to increase Dependency Degree by 30% and Cyclomatic complexity by 20%), then under the condition of these goal metrics X we need to find out which obfuscation method has the highest probability to achieve this.

2. According to Equation 6.1, because $P(X)$ is a constant value to every classification, and all the obfuscation methods have equal possibility to be chosen (i.e., $P(C_1) = P(C_2) = \ldots = P(C_m)$), the value of $P(X|C_i)$ is the only deciding factor for our choice, which means we only need to compute $P(Dep(0.30) \wedge Cyc(0.20)|\omega_i(p))$ from the training set.

3. Since the assumption of the algorithm is class conditional independence, and X (the target metrics) is the collection of every single metric x_k $(1 \leqslant k \leqslant n)$, the a posteriori probability of class C_i with a result X is

   $$P(X|C_i) = \prod_{k=1}^{n} P(x_k|C_i) \tag{6.2}$$

   where $P(x_k|C_i)$ $(1 \leqslant k \leqslant n)$ is estimated from the training set. In our example, $P(Dep(0.30) \wedge Cyc(0.20)|\omega_i(p)) = P(Dep(0.30)|\omega_i(p)) \cdot P(Cyc(0.20)|\omega_i(p))$.
   Because the values x_k are continuous and we assume that they are subject to Gaussian distribution with a mean value μ and standard deviation σ under the condition of $\omega_i(p)$, we can compute $P(Dep(0.30)|\omega_i(p))$ and $P(Cyc(0.20)|\omega_i(p))$.

4. In the last step, Equation 6.2 allows us to estimate the probability of reaching the target metric for every obfuscation method. We use this value to choose the obfuscation method we use in this iteration, i.e., we choose the dominance $P(X|C_i)$. The algorithm predicts that X is the result of class C_i, iff

   $$P(X|C_i) > P(X|C_j) \quad 1 \leqslant j \leqslant m, j \neq i \tag{6.3}$$

   Following the previous step, we have all the probability values $P(Dep(0.30) \wedge Cyc(0.20)|\omega_i(p))$ for every obfuscation method $\omega_i \in \Omega$.

After the obfuscation method is chosen and applied, our algorithm computes the distance of the resulting working copy to our target value. If the distance has deteriorated (i.e., increased), we end our search and return the previous value. If the distance has decreased, we start another iteration.

6.4.3 Total Probability

While the Naïve Bayesian Classifier determines the "best" next transformation, there is still the chance of a prediction error. This is because the algorithm only chooses a dominance class C_i learning from the training set, without knowing whether the overall probability of achieving this goal is high. For example, let ω_1 and ω_2 have the probability of 0.05 and 0.03 to reach the target value $X = \{Dep(0.80)\}$ with some confidence interval. It means that, our algorithm cannot reach the given target values with one transformation based on the current obfuscation methods. Therefore, to solve this problem, we dissect the final obfuscation goal values into different sections X' which the transformation most likely will reach: $X = X'_1 + X'_2 + \ldots + X'_m$ for $m \geqslant 1$. Since the total probability $P(X)$ can be measured from the training set, the dominance *sub-target* values $P(X'_i)$ before our final target can be calculated.

For example, we want an increase of $X = \{Dep(0.80) \wedge Cyc(0.60)\}$ and all the obfuscation methods have very low probability of increasing to these goal metrics. The Bayes Classifier only predicts one step. In this case, we need to split the goal metrics into three steps: $X = \{Dep(0.30) \wedge Cyc(0.20)\} + \{Dep(0.30) \wedge Cyc(0.20)\} + \{Dep(0.20) \wedge Cyc(0.20)\}$. These X'_i are chosen by the calculated highest total probabilities $P(X'_i)$ within the final goal values X.

6.4.4 Empirical Comparison of Algorithms

We ran both algorithms to approximate optimally obfuscated versions. Figure 6.2 (top) gives an overall comparison of the distribution of the final distances for the two algorithms over the same set of more than 110 APKs with the same target metric values. These were a target increase of 60% for Cyclomatic Complexity and an increase of 80% of Dependency Degree. The y-axis shows the number of the APKs and the x-axis shows the distances between the reached final metric values and the designated values. The Naïve Bayesian Classifier is marked in green and the simple search is marked in blue. The graph shows that 5 APKs' final Euclidean distances are less than 0.05 percentage points for both algorithms. Between the distances 0.05 and 0.10 percentage points the Naïve Bayesian Classifier has 37 APKs while for simple search the number is only 10. 70 APKs' distances are below 0.20 for Naïve Bayesian Classifier while 39 APKs are below 0.20 for simple search. So overall, if we use the Naïve Bayesian Classifier in the obfuscation space Ω^+ we get much better results than using simple search. This is supported by Figure 6.2 (right), which shows the mean and standard deviation of final distances between those the algorithm. The difference between them is large and

statistically significant: the mean of simple search is 0.3391 while the mean of Naïve Bayesian Classifier is 0.1923 (t-test rejects the hypothesis that both distributions are identical with p-value 0,0000049). So while the second algorithm does not necessarily produce optimally obfuscated versions, it is much better than the first.

6.5 Measuring Performance of Android Applications

We now show how we measure and compare the performance of Android applications before and after the obfuscation transformations.

We choose as performance metric the average number of exclusive method level CPU cycles on a real device, a Samsung i9100. In the following, we explain the rationale behind this decision and how it is done in our framework.

6.5.1 Measuring CPU Cycles

To measure the performance cost introduced by obfuscation, we chose CPU cycles as basic metric. More precisely, we use the *exclusive cycles* spent within a Java method as a comparison number. Exclusive cycles are those CPU cycles spent on the number of basic instructions within the method, not including idle times or time spent dealing with interrupts. For a given APK, we collect a number of method invocations (resulting from a GUI traversal, see below) and compute the sum of all exclusive cycles spent in all these methods. After obfuscation, we perform the same traversal of the APK and compute again the sum of the exclusive cycles of the methods. A comparison of the resulting two sums gives us the basis for a performance comparison.

From the set of possible methods to measure the performance of an Android application, we chose to use a *tracing based profiler* to measure CPU time of each Java method. CPU time basically can be calculated by multiplying the number of CPU cycles spent within the code with the (wall clock) time the machine spends on executing a single cycle (basically the reciprocal of the frequency in *Hz*). If we fix the CPU frequency, it will be sufficient to just measure CPU cycles. The Android *profiler* is a module of Android debug bridge (adb), to record the running time of each Java method after invoking the *MainActivity*.

6.5.2 Calibrating the Measurement

As mentioned above, the CPU time of a Java method is computed from the number of exclusive CPU cycles divided by the clock frequency. In normal cases, the CPU will choose the working frequency itself only depending on the CPU usage (the *ondemand* mode), which affects the precision of measurement. To confine this source of measurement noise we use the *userspace* mode to fix the working frequency. We then need to

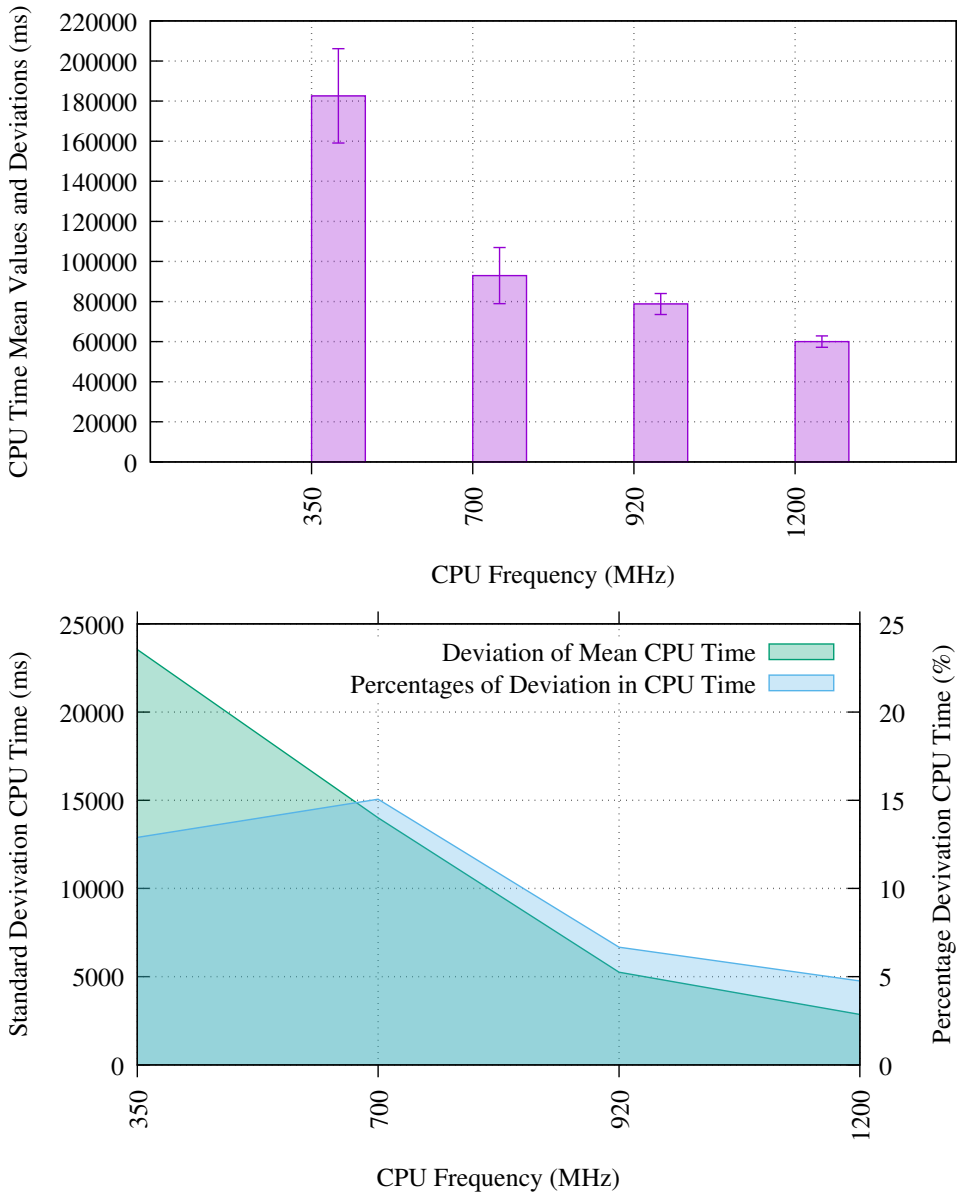

Figure 6.3: Total CPU time and deviation with different CPU frequencies: total CPU time with standard deviation (above) and standard deviation with the ratio in total CPU time(below).

find out to which value we should fix the frequency since there are several *Hz* levels that can be chosen.

To find the desirable value with the highest accuracy in performance measurement, we performed some experiments on our target experiment device Samsung Galaxy S II (GT-I9100, CPU Exynos 4210 dual-core max 1.2GHz, 32/32 KB I/D Cache, 1 MB L2 Cache, 1G RAM, 16G storage, OS Android 5.1.1.). We use *alarmclock.apk* as an example, and ran this APK 20 times under different CPU frequency (ranging from 350 MHz to 1200 MHz). We recorded the averages and standard deviation of the CPU time. The results are shown in Figure 6.3.

In this Figure, the graph on the top shows the averages and deviations of the consumed CPU cycles (y-axis) with the available frequencies (x-axis). From this graph, we can see that the number of used CPU cycles decreases with the increase of the frequency. More importantly, the standard deviation decreases. The graph at the bottom of Figure 6.3 shows the ratio of the total consumed CPU cycles for each frequency. The ratio is 12.8% (350MHz) and drops to 4.9% (1200MHz). To minimize the deviations stemming from frequency variations of the CPU, we fixed the frequency to 1200MHz, which is also the maximum we can choose on our testing device. Moreover, to have stable performance data for each result APK, we measured them for 5 times, and calculated the averages.

6.5.3 The Problem of GUI Traversal

In our framework, we use *Pandora* as an obfuscation engine. Its obfuscation process will cover all the Java methods in every *Activity* within the target Android applications. So if we only invoke the *MainActivity* of the those apps, the majority part of obfuscated Java methods might be excluded from performance change measurement. Therefore, to accurately measure the performance loss made by obfuscation, our testing should cover as many Java methods as possible in the target apps.

In Android, a GUI is almost everything a user can see and interact with. An Android app is composed by one or more GUIs (as the frontend) and all the supporting Java methods for these GUIs (as the backend). A collection of those GUIs can be formulated as a tree graph representing the app. Each node is a visible instance of the GUI itself and the edges are the connections between the current GUI state and the next, which are defined by GUI manipulations, e.g. by clicking a button. The starting node of the tree graph is the GUI at the entry point of the *MainActivity*. We disregard apps which only run as services in the background.

To increase the coverage of Java methods for the APKs we measured, we developed a tool that performed a "Depth-first with one step back" search of the GUI tree graph by clicking buttons. If the tool reaches a leaf node and cannot go any further, or if all the buttons in a particular GUI have been clicked already, the tool will press the back button. During the process of traversing the GUI, our tool logs every GUI element (which is the button) it has clicked. The tool then generates a log file for the target app.

To implement the performance measurements, we generated such log files for each APK in our measurement set. We use *monkeyrunner* to replicate the same GUI traversal for both cleartext APK and obfuscated APK. In both runs we measured and calculated the sum of the CPU time of each Java method. The performance overhead can then be quantified as the additional amount of CPU time caused by the obfuscation.

6.6 The Performance Cost of Obfuscation

Using our framework we can generate different versions of programs with (almost) arbitrary target complexity. We are also able to measure the performance cost of these transformations and even use performance as a target metric in the search. This allows us to empirically study the relationship between obfuscation techniques and software complexity metrics on the one hand, and the relationships of the two concepts with performance cost on the other hand.

In the following, we first study the performance cost from the obfuscation processes when the framework targets specific complexity metrics (Section 6.6.1). Then we investigate the relationship between performance cost and the obfuscation methods. We wish to find out which obfuscation methods are the most costly in terms of performance (Section 6.6.2).

6.6.1 The Performance Cost when Framework Targets Different Software Complexity Metrics

The effect of certain obfuscation techniques in relation to different software metrics can be classified into idempotent and monotonous as described in Chapter 5. Intuitively, monotonous techniques cause an increase in the complexity value with every (repeated) obfuscation step. An idempotent technique causes a change in the metric only once (usually in the first invocation). As mentioned above, we only consider the method metrics that are monotonous regarding the chosen complexity metrics in our framework.

To investigate the performance cost when our obfuscation framework targets various metrics and different complexity values, we performed the following experiment with a sample size of 100 APKs which are randomly selected from the F-Droid [35]. Firstly, we set our obfuscation framework to increase the three method metrics (DepDegree, LOC, Cyclomatic) to values of 30%, 50%, 100%, and 150%. Then we measure the extra performance costs made by the obfuscation. We use the 5% trimmed means. The averages and standard deviations of the performance cost are listed in Table 6.2. For example, a 30% target increase value of the metric DepDegree in our framework causes an average of 16% increase in performance cost (with a standard deviation of 0.33). After that, we performed one-way ANOVA tests among those target complexity values (from 30% to 150%) for every metric. The oneway ANOVA test is used to test whether the average value of the samples in the four groups are significantly different (null hypothesis

H_0). If the null hypothesis is rejected, then the performance cost of obfuscation can be influenced by the increased target values (from 30% to 150%). More precisely, the null hypothesis H_0 and alternative hypothesis H_1 are as follows:

- H_0: The increase of target complexity values does not influence the performance average.
- H_1: The increase of target complexity values influences the performance average.

The results of the test are shown on the right side of Table 6.2. Overall, the null hypothesis is rejected only for DepDegree ($p < 0.05$). This means that increasing the target value of DepDegree has a statistically significant influence on the average performance cost. For LOC and Cyclomatic we cannot reject H_0, which means that we can merely observe their performance cost increases but we cannot prove that they are significantly affected by the increases of target complexity values.

Table 6.2 also shows which of the metrics is the most performance expensive metric in our framework: For all complexity value increases of the three metrics, Cyclomatic always causes the highest performance cost (47% up to 67%). For increases of complexity value below 100%, the "cheapest" metric in terms of performance cost is DepDegree. For 100% increase and above, LOC incurs the least performance cost. To confirm this observation, we did one-way ANOVA test to compare the influence of a particular complexity metric on the performance average. We did this for the four complexity metric increases described above (30% to 150%). The results are all significant. Therefore, to achieve the same metric value increase, choosing different complexity metric significantly affects the performance cost.

We now discuss these results in more details by looking at the histograms of individual measurements from Figures 6.4 to 6.6 and from Figures 6.7 to 6.10.

The three different metrics and their performance costs are shown in the three graphs in Figure 6.4, Figure 6.5, and Figure 6.6. In these graphs, each of the same colored histograms, which are in blue, red, purple, or green, represents the performance cost distribution made by the three metrics with different target increase values of 30%, 50%, 100%, or 150%. For example, the blue histograms are the performance cost distributions when our framework targets the 30% increases of the metric DepDegree, LOC, or Cyclomatic, which are respectively in Figure 6.4, Figure 6.5, and Figure 6.6. With those three metric values varying from 30% to 150%, the histograms — colored from blue to green — are overall increasingly stretched to the right.

To make them more comparable to each other, we also display the Gaussian fit curves colored the same with the corresponding histogram. The x-value of the peak of the curve is the average value of the histogram, and the extension degree ("the stretch") along the x-axis of the fit curve can be used to compare the standard deviations (the longer the stretch, the larger the standard deviation).

For example, metric DepDegree is described in Figure 6.4. From 30% to 150%, the average values are 0.1681 (in blue curve peak x-value), 0.2310 (in red), 0.3908 (in purple),

and 0.5512 (in green), respectively, as shown in the first row of Table 6.2. In this table, from the 50% metric value increase of DepDegree, the average performance costs will basically increase linearly by around 0.16 when the metric values increase every 50%. Along with the averages, the standard deviations are also increasing.

Figure 6.5 shows the correlations between the Lines of Codes (LOC) and the performance: We can observe that the histograms and the Gaussian fit curves are rather similar and the average performance cost at 30% metric increase is slightly higher than that of 50% with the value 0.0344. And their standard deviations differ by 0.0646. However, the 30% increase and the 50% are very close to each other that their averages only differ by 0.0228, and that deviations differ by 0.0009. Therefore, in the second graph, we can also observe that the blue curve basically overlaps with the purple. Meanwhile, the red curve is stretching more to the left comparing to both of the blue and purple. But from 50% to 150%, each step of metric increase has around 0.05 more of average performance cost, which is much less than that of DepDegree. As shown in this graph, the x-value of the peaks from of the red, the purple to the green curves steadily increase with a small amount.

In Figure 6.6, the extra performance overhead of the metric Cyclomatic is shown. From the 30% to 100%, the growing rate is relatively low. From 100% to 150% the performance average increases by 0.1391 to 0.6786. Therefore the peaks of blue, red, purple curves are relatively close to each other. The green is distinctively further to the positive direction of the x-axis.

We now discuss in more details the four measurement samples for the four classes of complexity metric increase (30% to 150%). The histograms together with the Gaussian fit curves for these four cases are shown From Figure 6.7 to Figure 6.10. Each of the graph in figure describes the distributions of the three complexity metrics when their metric increase values fix to a ratio, for example, Figure 6.7 describes the distribution differences of the metrics DepDegree, LOC, and Cyclomatic with 30% metric increases. It allows us to compare the performance cost of different complexity metrics for the same level of increase. In this figure, the blue histograms and curves represent DepDegree, the red represent LOC, and the purple represent Cyclomatic. From each of graphs, we intuitively observe that the purple curve (Cyclomatic) always stretches out furthest to the x-axis positive direction. It means Cyclomatic is always the most expensive in performance cost when targeting the same metric value increase. But according to Table 6.2, the difference in average performance cost between Cyclomatic (purple) and DepDegree (blue) gets smaller with larger complexity metric increases (from 0.31 for 30% to 0.12 for 150%). Thus, in the four graphs, we can observe that the blue curves and the purple curves get closer from the top left to the bottom right. In the first two graphs (for metric value increase of 30% and 50%), the performance cost average of the DepDegree (blue) is lower than that of LOC (red). However, starting from 100% metric value increase (Figure 6.9) it is the other way round. Therefore, from Figure 6.9, the peaks x-values of blue curves exceed that of red. It means that the increase of performance cost of DepDegree outpaces that of LOC when our framework targets 100% metric increase

Table 6.2: Performance cost of reaching particular complexity metric increases ($n = 100$).

Metrics	AVG&STD	30%	50%	100%	150%	ANOVA f-value	p-value
DepDegree	AVG	0.1681	0.2310	0.3908	0.5512	12.9776	4.3071e-08
	STD	0.3325	0.4346	0.4803	0.6050		
LOC	AVG	0.3158	0.2814	0.3386	0.3881	0.7677	0.5126
	STD	0.5338	0.4692	0.5347	0.4829		
Cyclomatic	AVG	0.4745	0.4893	0.5395	0.6786	2.2283	0.0844
	STD	0.6286	0.5287	0.5976	0.7042		
ANOVA	f-value	8.9296	7.7838	3.2601	5.6526		
	p-value	0.0001	0.0005	0.0399	0.0039		

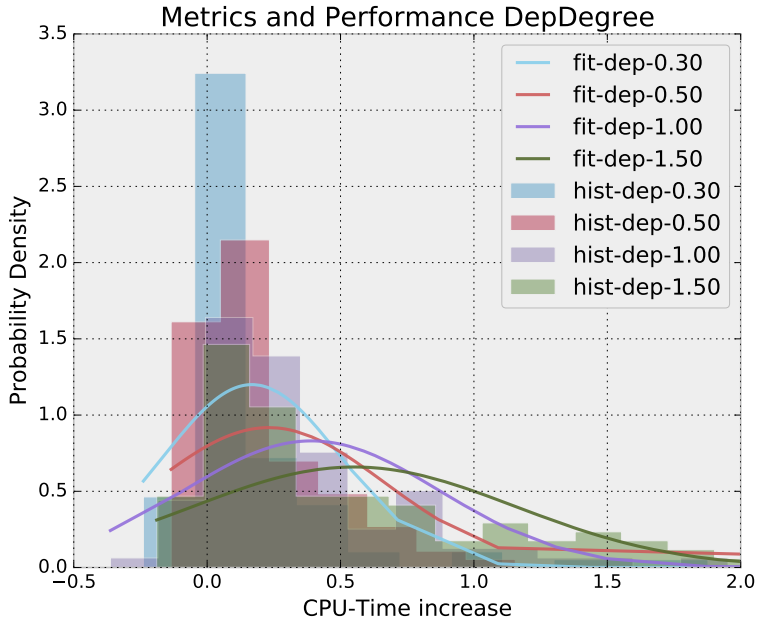

Figure 6.4: Performance overhead increase histogram and the fit curves within the metrics DepDegree with target increase 30%, 50%, 100%, and 150%

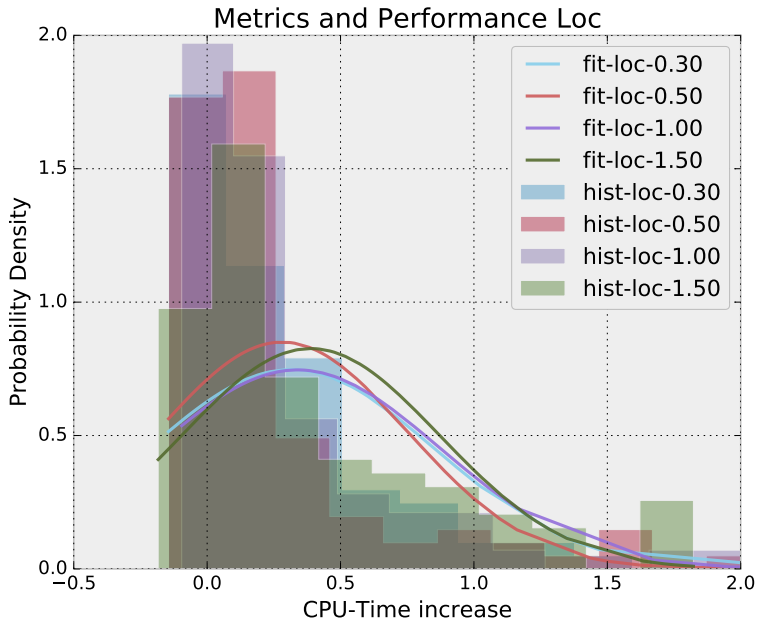

Figure 6.5: Performance overhead increase histogram and the fit curves within the metrics LOC with target increase 30%, 50%, 100%, and 150%

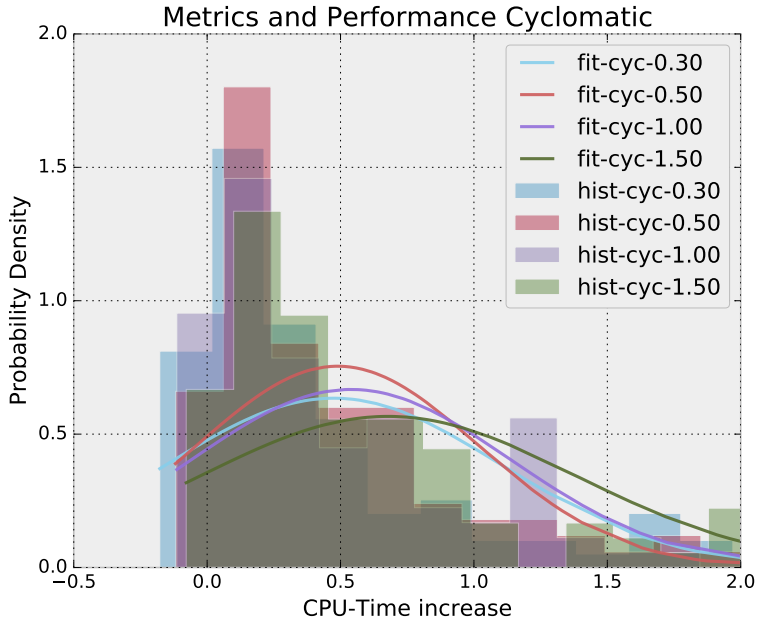

Figure 6.6: Performance overhead increase histogram and the fit curves within the metrics Cyclomatic with target increase 30%, 50%, 100%, and 150%

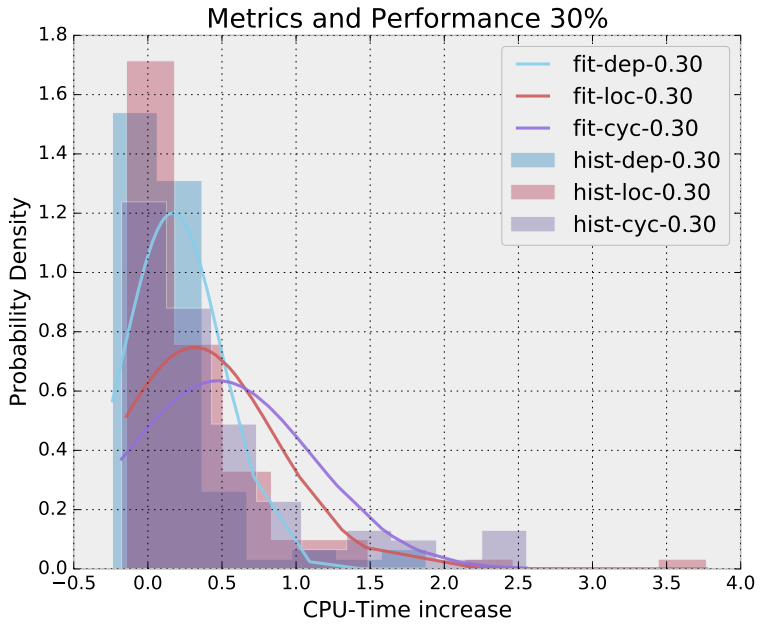

Figure 6.7: Performance overhead increase histogram and the fit curves among the metrics DepDegree, LOC, and Cyclomatic with target increase 30%

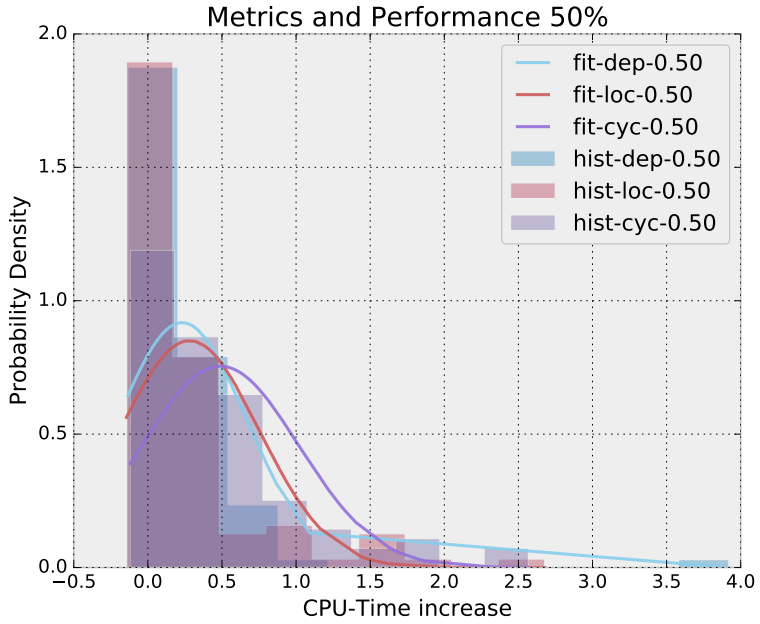

Figure 6.8: Performance overhead increase histogram and the fit curves among the metrics DepDegree, LOC, and Cyclomatic with target increase 50%

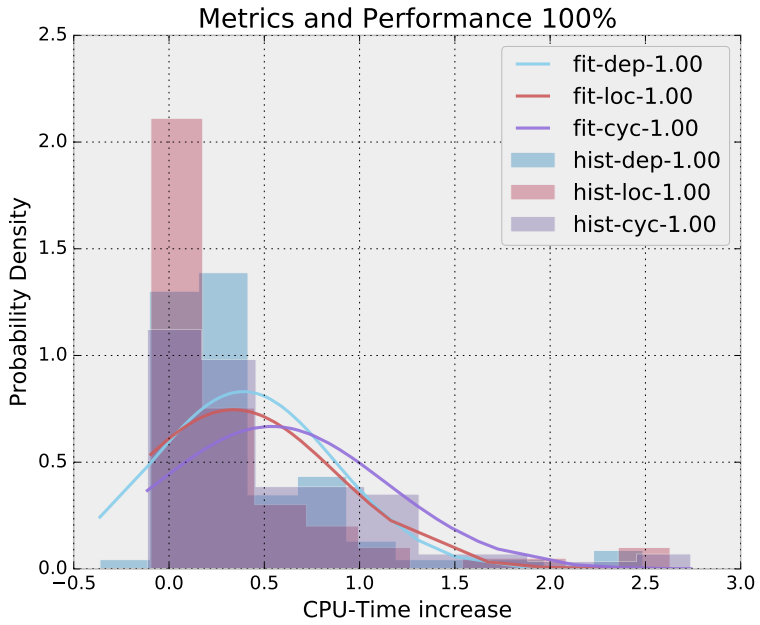

Figure 6.9: Performance overhead increase histogram and the fit curves among the metrics DepDegree, LOC, and Cyclomatic with target increase 100%

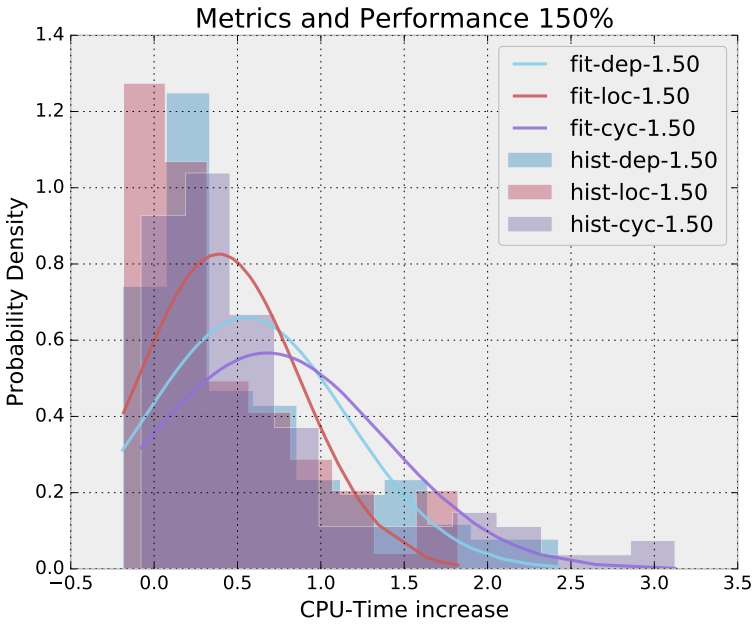

Figure 6.10: Performance overhead increase histogram and the fit curves among the metrics DepDegree, LOC, and Cyclomatic with target increase 150%

and hereafter.

6.6.2 The Performance Cost of Obfuscation Methods

We now investigate the performance overhead caused by different obfuscation methods. As mentioned above, we selected 7 obfuscation methods out of 11 because they are monotonous with respect to the three metrics discussed in Section 6.2.1. The results are shown in Table 6.3.

For DepDegree, the highest average increase is caused by Integer Encoding (0.5885) which causes an extra performance cost of only 0.0789 on average. The second highest average increase comes from Compose Locals (0.4750). Its performance penalty however is 0.3398, which is almost 5 times of that of Integer Encoding. The performance cost of String Encryption, though only causing a 0.0746 metric increase, is up to 0.5844. This is similar to the effect of Move Methods.

The metric increases and performance penalty for Lines of Code (LOC) are similar to those of DepDegree. Integer Encoding is the highest in metric change but lowest in the performance cost. Compose Locals is the second highest regarding metric increase but scores relatively high in performance penalty. String Encryption and Move Methods are rather costly in terms of performance but very low in metric increase.

Regarding Cyclomatic the results are quite different from DepDegree and LOC. Here, Encapsulate Fields is the method which changes the metric most (0.2042) and is cheap in performance (0.0450). In this metric most of the average values are below 0.20, which is less than the previous metrics. So, to reach similar values, more steps of obfuscation are needed. If one relates performance cost to metric increase, Cyclomatic is on average the most performance expensive metric.

We now analyze the two obfuscation methods, String Encryption and Compose Locals, in more details. Figure 6.11 shows histograms of the metric increases of String Encryption in our experiments. It shows that most APKs do not change much regarding the three metrics we considered. The histogram of performance cost, however, does not show a particular peak. Figure 6.12 shows the same graphs for Compose Locals: the metric changes are comparatively larger and the performance increase shows a peak around the value 0.2. Similar measurements were taken for Integer Encoding. Overall, Integer Encoding and Compose Locals appear to be rather cheap in terms of performance overhead.

Table 6.3: Average (AVG) metric increases with standard deviation (STD) and performance overhead for each obfuscation method ($n = 100$)

Obfu Methods	AVG&STD	Str Encrypt	Int Encode	Comp Local	Extract Meth	Move Meth	Encap Field	Merge Meth
DepDegree	AVG	0.0746	0.5885	0.4705	0.0671	0.0503	0.0731	0.0074
	STD	0.2044	0.3220	0.2124	0.0755	0.0859	0.0758	0.0648
LOC	AVG	0.0455	0.5730	0.3844	0.1016	0.0436	0.1271	0.0017
	STD	0.1178	0.3524	0.1851	0.0682	0.0671	0.0798	0.0482
Cyclomatic	AVG	0.0211	0.1528	0.1950	0.1312	0.0050	0.2042	0.0018
	STD	0.0839	0.1776	0.1897	0.0916	0.0639	0.1495	0.0644
Performance	AVG	0.5844	0.0789	0.3398	0.0358	0.5196	0.0450	0.0159
	STD	0.6926	0.1654	0.3675	0.1116	0.9528	0.1011	0.1126

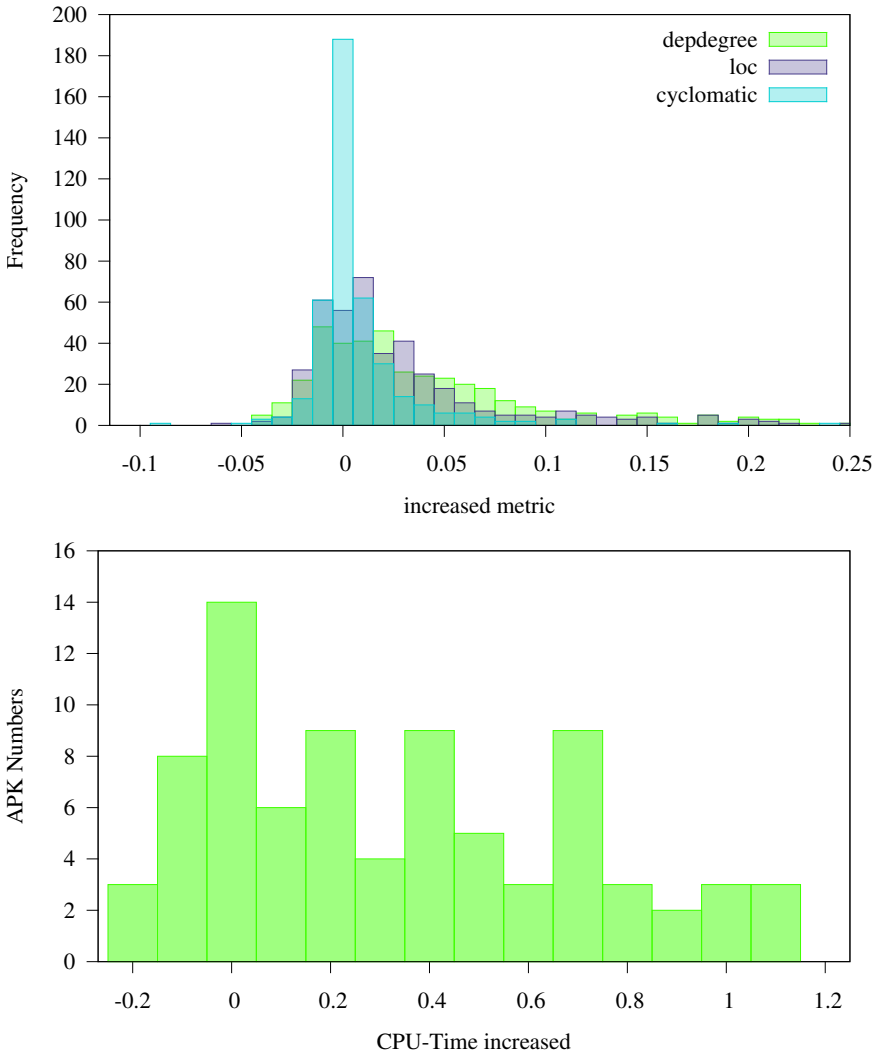

Figure 6.11: Histogram of the method metric increase and the performance overhead increase regarding String Encryption.

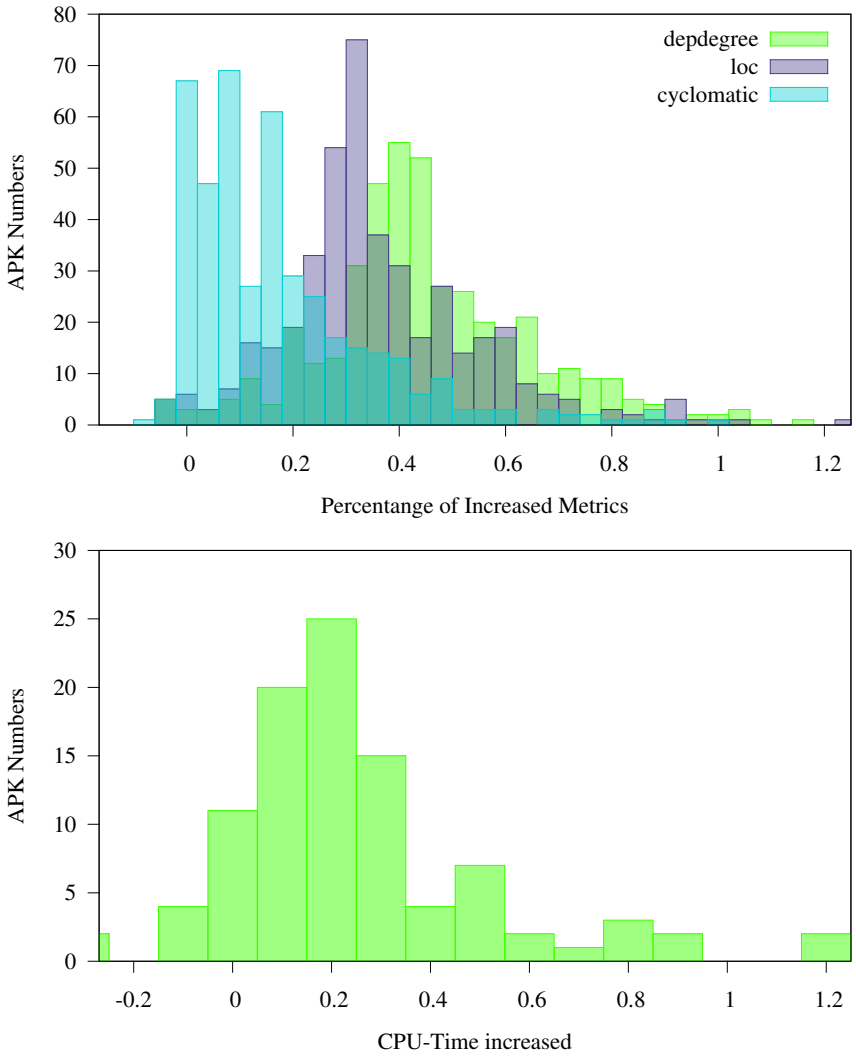

Figure 6.12: Histogram of the method metric increase and the performance overhead increase regarding Compose Locals.

6.6.3 Performance as a Special Metric

Given the fact that some obfuscation methods are cheaper regarding performance than others, we can use this insight to create obfuscated programs that reach the same metric value increase at a smaller performance overhead. We therefore use the statistical data from the previous section in our algorithm based on the Naïve Bayesian Classifier. We adapted our framework such that the maximum performance overhead could be given as an additional metric value. For example, we could now start the search for a version of the program with an increase of DepDegree of 100% but where the performance cost should be limited to a 35% increase.

Table 6.4: Performance cost ("perf") of obfuscated programs resulting from the Naïve Bayesian algorithm by limiting performance loss to the average performance increase ("norm") of earlier measurements.

Metrics		30%		50%		100%		150%	
		perf	norm	perf	norm	perf	norm	perf	norm
DepDegree	AVG	0.0172	0.8971	0.0394	0.8853	0.1108	1.0043	0.2112	1.1567
	STD	0.1039	1.7770	0.0836	1.4309	0.1283	0.7738	0.2063	0.5140
LOC	AVG	0.1002	0.8374	0.0840	1.0187	0.1670	1.1973	0.1808	1.2143
	STD	0.1398	0.7053	0.1054	0.7084	0.1191	1.4624	0.1375	1.2750
Cyclomatic	AVG	0.1528	1.5055	0.3202	1.4301	0.4146	1.4508	0.5053	1.9411
	STD	0.2103	1.1109	0.2361	1.3185	0.2072	0.9974	0.2651	1.5605

Recalling the measurements from Section 6.6.1, we now generated obfuscated versions of 100 APKs for different metric value increases (from 30% to 150%) and we used the average performance cost from the original measurements as a boundary in guiding the search. The results are shown in the Table 6.4. For each metric (DepDegree, LOC, Cyclomatic) the table lists the average performance increase (AVG) and standard deviation (STD) that was measured before performance optimization ("norm") and contrasts it with the average performance increase when low performance penalty is used as additional search criterion in the the Naïve Bayesian Classifier algorithm ("perf"). The table shows that the same metric increase can be achieved with dramatically lower performance costs. We observe that the ratio between optimized performance increase and "normal" performance increase gets smaller as metric values (of DepDegree, LOC, Cyclomatic) increase. The largest savings in performance can therefore be achieved for small metric value increases. For example, at a 30% increase in DepDegree the optimized program runs 50 times faster that the non-optimized version, whereas the narrowest margin takes place when the metric Cyclomatic is increased to 150%. In this case we are "only" faster by a factor of 4. Overall, DepDegree can be optimized the most, i.e., it has the highest chance to be optimized regarding performance. Even after performance optimization, Cyclomatic remains the most expensive metric regarding average .

From Figure 6.13 to 6.15, the blue plots are the performance of the result APKs which are obfuscated respectively to the target metric (DepDegree, LOC, Cyclomatic) and their desired values (30%, 50%, 100%, 150%). (e.g. in DepDegree 150%, we select the APKs with the performance lager than the average value 0.5319 from the Table 6.2.). The red are those same APKs under the performance optimized obfuscation. In these graphs, the axis of "Actual Metric Values" are the metric values they have reached after obfuscation. The "Target Metric Values" in the graphs are the desired values, and the goal of our obfuscation framework. As we can see in those graphs, all the blue plots and red plots are basically vertical distributed. For the blue plots, we can see that they mainly concentrate around the axis of target values, but stretched to larger performance

values. For instance in the Figure 6.13 (DepDegree), when the target metric values are pointing to 150% (which is the most inside layer of distribution in this graph), the blue plots gather around "Actual Metric Values" of 1.5. Meanwhile, their "Performance" values are getting larger.

Representing the performance optimized APKs, the majority of the red plots are distributed under a certain performance values, which are the average values we acquired from the Table 6.2. Because during the obfuscation process, we do statistics based on all the results performance data from Section 6.6.2, and then use them in our Naïve Bayesian Classifier, same as the other complexity metrics. After calculation, we would choose the dominant obfuscation method (which is performance and metric economy) as introduced in Section 6.4.2. In this way, we can limit the increase in performance cost under the averages.

Before our obfuscation framework is performance optimized, we only confine the complexity increase. Because when we obfuscate blindly only by the complexity metrics, the result APKs could be very possibly over obfuscated. It means that the performance is overburdened by too many increased metrics values (Our theory assumes that more complexity metric requires more computing resources). After optimization, the concern of performance overburden is solved. We could set the framework with a performance overhead limitation, the result metric could be over the target value and as high as possible. As a result, the red plots in these Figures can be very high in the metric values while limited in performance overhead. We take the same example when DepDegree targeting 150% in Figure 6.13, the red plots are stretched to the larger values of the "Actual Metric Values" axis, but all of them are below performance values 0.5319 which is the average of extra performance cost from the Table 6.2. A minimal of the APKs cannot reach the 80% or 90% of the target metric values after we use the performance as another metric for optimization, the ratio of those APKs and the total sample space are shown in Table 6.5.

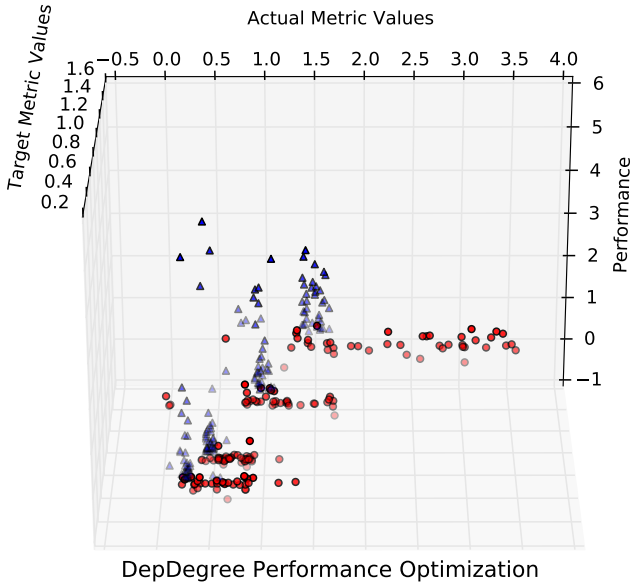

Figure 6.13: Plotting of the Performance and DepDegree

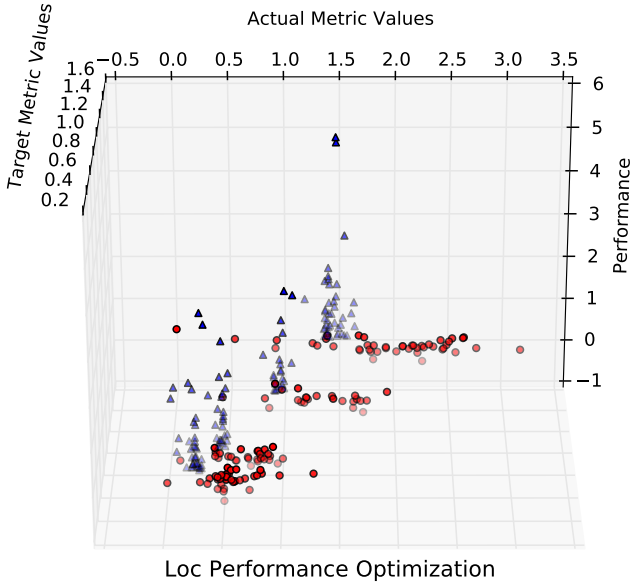

Figure 6.14: Plotting of the Performance and LOC

Table 6.5: The ratio of the APKs that can NOT reach the 0.8 and 0.9 times target complexity metric values.

Metrics	Target Values	30%		50%		100%		150%	
	Range&Total	<0.9	<0.8	<0.9	<0.8	<0.9	<0.8	<0.9	<0.8
DepDegree	Range	13.5%	2.7%	6.52%	4.35%	5.26%	5.26%	11.36%	11.36%
	Total	37		46		38		44	
LOC	Range	33.33%	12.82%	20.69%	13.79%	4%	%	2.27%	2.27%
	Total	39		29		25		44	
Cyclomatic	Range	52.77%	5%	7.69%	5.13%	16.28%	11.63%	23.91%	18.6%
	Total	36		39		43		46	

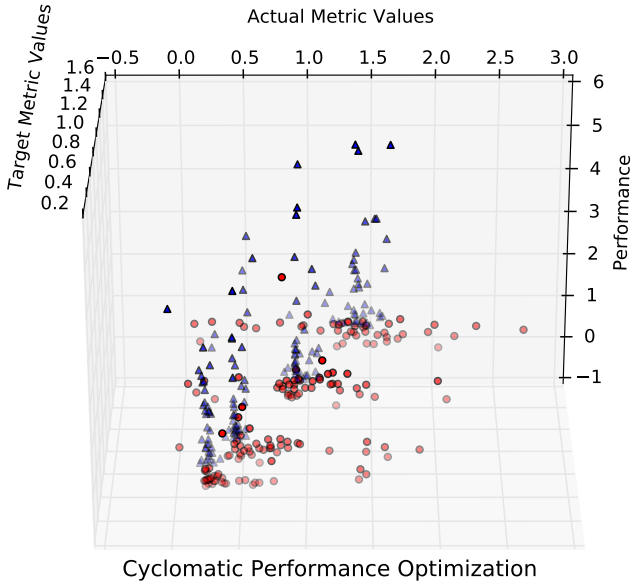

Figure 6.15: Plotting of the Performance and Cyclomatic

6.7 Conclusion

In this chapter we defined the problem of optimal software obfuscation and empirically evaluated two search algorithms for obfuscated versions in the context of Android APKs. With a training set of more than 400 APKs and measurements on 110 APKs, we could show that the second algorithm based on the Naïve Bayesian Classifier outperforms a simple greedy search considerably. We also investigated the performance cost of different

types of software obfuscation in this context. We showed that some obfuscation methods (like String Encryption) are more performance costly than others to achieve the same metric value increase. We also found that especially for small metric value increases, large performance benefits can be achieved if the right obfuscation techniques are used, i.e. those that are less costly in terms of performance.

Chapter 7

Conclusion

7.1 Summary

In Chapter 4, we have given an overview of the composition and the work flows of our obfuscation framework, which we define as the process of creating all the optimized result APKs in both complexities and performance. It produces all the result in the following chapters. We have introduced the working principle of the obfuscation methods and software complexity metrics and the classification of the attributes of the methods and the metrics, which are categorized into the method level (data control flows) and the class level (Object-Oriented Design). We pointed out the rationale of measuring "difficulty" and complex level caused by the obfuscation methods by the software complexity metrics. We have developed an automatic testing tool to generate the testing cases for performance analysis. The testing cases refer to the clicking or sliding action used to invoke the active GUI elements on the testing device. Because the tree based data structure of the GUI elements and the Android activities, the testing tool must implement "Depth-First search with one step back" to traverse the tree, which can cover most of the active elements, hence achieving the maximum code coverage. The obfuscation management layer is designed to coordinate and control all the modules in the framework. We offered an overview on its mechanism: Based on the learning set data, it will evaluate and apply different obfuscation method for different scenarios.

To further optimize the obfuscation process with complexity metrics, a number of transformation results of each obfuscation method need to be analyzed, and used as the learning set. In Chapter 5, we have given an overview that is based on different transformation theories and mechanisms. The obfuscation methods show different but structural attributes when evaluated by an array of software complexity metrics. To demonstrate the attributes we have implemented our framework to iteratively apply the same obfuscation methods onto the APKs, and evaluated the difference before and after transformation with metrics and similarity. Using these proof-of-concept implementations, we have performed the evaluation of 7 obfuscation methods with 9 complexity metrics. We showed from the pictures that the results are rather complex since a single obfuscation method usually exhibits different properties (i.e. monotonicity or idempotency) and values regarding different complexity metrics. Meanwhile, most of the monotonicity obfuscation methods exhibit a rather "stable" attribute, that complexity value rises constantly by a certain ratio after each iteration. However, a few obfuscation methods demonstrate unstable properties regarding some of the metrics. The analysis of the obfuscation methods properties is the basis of our further research, which is to optimize the APKs to the designated metrics with minimum obfuscation iterations.

In Chapter 6, we define an obfuscation as "successful" when it transforms the APK to a certain level of "difficulty", without significantly slowing it down. Instead of the abstract "difficulty" level, we use software complexity metrics values to measure concrete obfuscation results. To reach these concrete metrics values, we implement algorithms, which are either the *Naïve Bayes Classifier* or the *mean values based simple search*, in the Obfuscation Management Layer of the framework to guide the obfuscation process to select the suitable obfuscation methods. The algorithms determine which method to use according to the complexity metrics data of the learning set from obfuscation history. By comparing the results of the two algorithms, the *Naïve Bayes Classifier* outperforms the *mean values based simple search*. We then use the *Naïve Bayes Classifier* to guide the obfuscation methods selections to optimize the complexity metrics into target values and confine the performance penalty. To start the performance measurement, we implement the performance measurement modules in the framework during obfuscation process. We have evaluated performance penalty of different obfuscated methods and also that in relation to three metrics at different increased complexity values, i.e. DepDegree, LOC, Cyclomatic Complexity. Our evaluation showed that DepDegree is the most performance saving metric while having the same target metric values as Cyclomatic Complexity, which is the most performance expensive metric. Meanwhile, some of the obfuscation methods are significantly more costly in performance than others when the metric value changes are similar, i.e. Strings Encryption, Compose Locals, and Move Methods. Therefore we conclude that, considering performance as a specific target metric during obfuscation, the Obfuscation Management Layer in our framework will produce the result APKs with far lower performance cost, meanwhile satisfying the target complexity metrics.

7.2 Future Work

While the techniques we have developed in this thesis have extended optimized obfuscation from complexity to performance, there is still potential for further research. To maintain user experience after obfuscation, the Obfuscation Management Layer can statistically guide the obfuscation methods selection in the transformation process with the minimum performance cost. However, in mobile devices, not only performance, but energy consumption is also an important criterion for the user. Because obfuscation transformation causes more complexity for the software, more computing power is consumed. Measured by the CPU cycles, performance expenses generated by obfuscation cannot fully reflect extra power consumptions. Therefore, it is necessary to measure and profile the energy consumption for the target application. The hardware based energy profiling toolset, the Monsoon power monitor [94], for example, can be implemented to measure energy consumption more exactly than the power modeling or estimation software [48, 68, 47]. Because the Monsoon power monitor is only an energy measurement tool, we then need to align its time indexed energy consumption curves exactly to our performance profiling results in order to show accurately the time schedule of each Java method. As a result, we could investigate the correlation between energy consumption

and different obfuscation methods or complexity metrics at different values.

Furthermore, we suggest researching on the correlation between complexity metrics values and the "difficulty" level from the human perspective. While this approach is much more complicated, it is interesting to know to what extent complexity values add "difficulty" to reverse engineer the software. This can provide a more accurate complexity target to guide our obfuscation process.

Bibliography

[1] LLVM. http://llvm.org/, 2007.

[2] Dex2jar. https://sourceforge.net/projects/dex2jar/, 2009.

[3] Snappy. https://github.com/google/snappy, 2011.

[4] apk-view-tracer. https://code.google.com/archive/p/apk-view-tracer/, 2012.

[5] Profiling with traceview and dmtracedump. https://developer.android.com/studio/profile/traceview.html#traceviewLayout, 2012.

[6] IDA Pro homepage. https://www.hex-rays.com/products/ida/index.shtml, 2015.

[7] ISO/IEC 9126-2. Software engineering – product quality – part 2: External metrics. ISO/IEC, June 2001.

[8] Ivan Arce and Elias Levy. An analysis of the slapper worm. *IEEE Security & Privacy*, 1(1):82–87, 2003.

[9] John Aycock. *Computer Viruses and Malware*, volume 22 of *Advances in Information Security*. Springer, 2006.

[10] Barak, Goldreich, Impagliazzo, Rudich, Sahai, Vadhan, and Yang. On the (im)possibility of obfuscating programs. In *CRYPTO: Proceedings of Crypto*, 2001.

[11] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[12] Victor R. Basili and Barry T. Perricone. Software errors and complexity: An empirical investigation. *Commun. ACM*, 27(1):42–52, 1984.

[13] Philippe Beaucamps. Advanced metamorphic techniques in computer viruses. Author manuscript, published in "International Conference on Computer, Electrical, and Systems Science, and Engineering - CESSE'07 (2007), November 14 2008.

[14] M. Becher, F. C. Freiling, and B. Leider. On the effort to create smartphone worms in windows mobile. In *2007 IEEE SMC Information Assurance and Security Workshop*, pages 199–206, June 2007.

[15] Michael Becher. *Security of Smartphones at the Dawn of Their Ubiquitousness*. Phd thesis, University of Mannheim, Germany, 2009.

[16] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Márk Félegyházi. The cousins of stuxnet: Duqu, flame, and gauss. *Future Internet*, 4(4):971–1003, 2012.

[17] Dirk Beyer and Ashgan Fararooy. Depdigger: A tool for detecting complex low-level dependencies. *International Conference on Program Comprehension*, 00:40–41, 2010.

[18] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr M. Youssef, Mourad Debbabi, and Lingyu Wang. On the analysis of the zeus botnet crimeware toolkit. In *PST*, pages 31–38. IEEE, 2010.

[19] Bitdefender. Malware history. Technical report, 2008.

[20] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In Dirk Grunwald, Richard Han, Eyal de Lara, and Carla Schlatter Ellis, editors, *MobiSys*, pages 225–238. ACM, 2008.

[21] Ran Canetti and Ronny Ramzi Dakdouk. Obfuscating Point Functions with Multibit Output. In *EUROCRYPT '08 (Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology)*, pages 489–508, Istanbul, Turkey, April 2008. Springer-Verlag, Berlin.

[22] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. A large study on the effect of code obfuscation on the quality of java code. *Empirical Software Engineering*, 20(6):1486–1524, 2015.

[23] Mariano Ceccato, Massimiliano Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, pages 1–35, 2013.

[24] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. Towards experimental evaluation of code obfuscation techniques. In Andy Ozment and Ketil Stølen, editors, *QoP*, pages 39–46. ACM, 2008.

[25] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *ICPC*, pages 178–187, 2009.

[26] Silvio Cesare and Yang Xiang. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing - Volume 107*, AusPDC '10, pages 61–70, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.

[27] Cilibrasi and Vitani. Clustering by compression. *IEEETIT: IEEE Transactions on Information Theory*, 51, 2005.

[28] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking and Tamperproofing for Software Protection*. Addison-Wesley Longman, Amsterdam, July 2009.

[29] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences, The University of Auckland, July 1997.

[30] Jedidiah R. Crandall, Roya Ensafi, Stephanie Forrest, Joshua Ladau, and Bilal Shebaro. The ecology of malware. In Matt Bishop, Christian W. Probst, Angelos D. Keromytis, and Anil Somayaji, editors, *NSPW*, pages 99–106. ACM, 2008.

[31] Jonathan Crussell, Clint Gibler, and Hao Chen 0003. Attack of the clones: Detecting cloned applications on android markets. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *ESORICS*, volume 7459 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2012.

[32] Bill Curtis, Sylvia B. Sheppard, and Phil Milliman. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, pages 356–360, Piscataway, NJ, USA, 1979. IEEE Press.

[33] Tom deMarco. *Controlling Software Projects: Management, Measurement, and Estimates*. Springer-Verlag, 1986.

[34] Anthony Desnos. Androguard. `https://code.google.com/p/androguard/`, 2012.

[35] F-Droid Ltd. F-droid. `https://f-droid.org/`, 2010.

[36] F-Secure. Bluetooth-worm:symbos/cabir. http://www.f-secure.com/v-descs/cabir.shtml, 2004.

[37] F-Secure. Virus:w32/duts.1520. http://www.f-secure.com/v-descs/dtus.shtml, 2004.

[38] Rik Ferguson. The botnet chronicles. Technical report, Trend Micro, Incorporated, 2010.

[39] Peter Szor Frdric Perriot. An analysis of the slapper worm exploit. Technical report, Symantec Security Response, 2003.

[40] Felix C. Freiling, Mykola Protsenko, and Yan Zhuang. An empirical evaluation of software obfuscation techniques applied to Android APKs. In *International Workshop on Data Protection in Mobile and Pervasive Computing (DAPRO 2014)*, 2014.

[41] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *IACR Cryptology ePrint Archive*, 2013:451, 2013.

[42] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[43] Roberto Giacobazzi and Isabella Mastroeni. Making abstract interpretation incomplete: Modeling the potency of obfuscation. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 129–145, Berlin, Heidelberg, 2012. Springer-Verlag.

[44] Alexander Gostev. Mobile malware evolution: An overview, part 1. http://www.securelist.com/en/analysis?pubid=200119916, 2006.

[45] Alexander Gostev. Mobile malware evolution: An overview, part 2. http://www.securelist.com/en/analysis?pubid=201225789, 2006.

[46] Steve Hanna, Ling Huang, Edward XueJun Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In Ulrich Flegel, Evangelos P. Markatos, and William K. Robertson, editors, *DIMVA*, volume 7591 of *Lecture Notes in Computer Science*, pages 62–81. Springer, 2012.

[47] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating android applications' CPU energy usage via bytecode profiling. In Rick Kazman, Patricia Lago, Niklaus Meyer, Maurizio Morisio, Hausi A. Müller, Frances Paulisch, Giuseppe Scanniello, and Olaf Zimmermann, editors, *GREENS*, pages 1–7. IEEE, 2012.

[48] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *ICSE*, pages 92–101. IEEE Computer Society, 2013.

[49] Thorsten Holz, Markus Engelberth, and Felix C. Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. In Michael Backes and Peng Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009.

[50] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. A framework for evaluating mobile app repackaging detection algorithms. In Michael Huth, N. Asokan, Srdjan Capkun, Ivan Flechais, and Lizzie Coles-Kemp, editors, *TRUST*, volume 7904 of *Lecture Notes in Computer Science*, pages 169–186. Springer, 2013.

[51] R. Jain. The art of computer systems performance analysis. *SIGMETRICS Performance Evaluation Review*, 18(3):21–22, 1990.

[52] Dennis G. Kafura and Geereddy R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering*, 13(3):335–343, March 1987.

[53] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[54] Taghi M. Khoshgoftaar and John C. Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, 1990.

[55] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981.

[56] D. Krishnaswamy, R. N. Hasbun, and J. P. Brizek. Secure manageable mobile handset platform architectures. *Comm. Mag.*, 44(9):158–165, September 2006.

[57] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened ASLR on android. In *IEEE Symposium on Security and Privacy*, pages 424–439. IEEE Computer Society, 2014.

[58] Anirban Majumdar, Stephen Drape, and Clark D. Thomborson. Slicing obfuscations: design, correctness, and evaluation. In Moti Yung, Aggelos Kiayias, and Ahmad-Reza Sadeghi, editors, *Digital Rights Management Workshop*, pages 70–81. ACM, 2007.

[59] Denis Maslennikov. ZeuS-in-the-Mobile Facts and Theories. 2011.

[60] Daniel A. Menasce, Lawrence W. Dowdy, and Virgilio A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[61] Jonathan Meyer and Daniel Reynaud. Jasmin. `http://jasmin.sourceforge.net/`, 2004.

[62] Trend Micro. Repeating history. Technical report, 2012.

[63] Michael Miller. *Absolute PC Security and Privacy*. 2002.

[64] Carey Nachenberg. Computer virus-antivirus coevolution. *Commun. ACM*, 40(1):46–51, January 1997.

[65] Eric Chien Nicolas Falliere. Zeus: King of the bots. Technical report, Symantec, 2009.

[66] Hilarie K. Orman. The morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, 1(5):35–43, 2003.

[67] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In Pascal Felber, Frank Bellosa, and Herbert Bos, editors, *EuroSys*, pages 29–42. ACM, 2012.

[68] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang. Fine-grained power modeling for smartphones using system call tracing. In Christoph M. Kirsch and Gernot Heiser, editors, *EuroSys*, pages 153–168. ACM, 2011.

[69] Phillip A. Porras, Hassen Saïdi, and Vinod Yegneswaran. An analysis of the ikee.B iphone botnet. In Andreas U. Schmidt, Giovanni Russello, Antonio Lioy, Neeli R. Prasad, and Shiguo Lian, editors, *MobiSec*, volume 47 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 141–152. Springer, 2010.

[70] Pouik and G0rfi3ld. Similarities for fun and profit. `http://phrack.org/issues/68/15.html`, April 2014.

[71] Mila Dalla Preda and Roberto Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2005.

[72] Mila Dalla Preda and Roberto Giacobazzi. Semantics-based Code Obfuscation by Abstract Interpretation. *Journal of Computer Security (JCS)*, 17(6):855–908, 2009.

[73] Mila Dalla Preda, Roberto Giacobazzi, Matias Madou, and Koen De Bosschere. Opaque predicates detection by abstract interpretation, 2006.

[74] Mykola Protsenko and Tilo Müller. Pandora applies non-deterministic obfuscation randomly to android. In *MALWARE*, pages 59–67. IEEE, 2013.

[75] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser: Analysis of web-based malware. In *First Workshop on Hot Topics in Understanding Botnets, HotBots'07*. USENIX, 2007.

[76] Krishna K. Ramachandran and Biplab Sikdar. Modeling malware propagation in networks of smart cell phones with spatial dynamics. In *INFOCOM*, pages 2516–2520. IEEE, 2007.

[77] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating android anti-malware against transformation attacks, July 15 2013.

[78] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[79] Antonino Sabetta and Heiko Koziolek. Measuring performance metrics: Techniques and tools. In Irene Eusgeld, Felix C. Freiling, and Ralf H. Reussner, editors, *Dependability Metrics*, volume 4909 of *Lecture Notes in Computer Science*, pages 226–232. Springer, 2005.

[80] Sebastian Schrittwieser, Stefan Katzenbeisser 0001, Johannes Kinder, Georg Merzdovnik, and Edgar R. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv*, 49(1):4, 2016.

[81] SECURELIST. Trojan.symbos.mosquit.a. http://www.securelist.com/ en/descriptions/old58141, 2004.

[82] V. Y. Shen, T. J. Yu, S. M. Thebaut, and L. R. Paulsen. Identifying error-prone software - an empirical study. *IEEE Transactions on Software Engineering*, SE-11(4):317–25, 1985.

[83] Rich Skrenta. Elk cloner. `http://www.skrenta.com/cloner/`, 2013.

[84] Connie U. Smith and Lloyd G. Williams. Performance solutions: A practical guide to creating responsive, scalable software. In *Int. CMG Conference*, pages 355–358. Computer Measurement Group, 2001.

[85] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. On the infeasibility of modeling polymorphic shellcode - rethinking the role of learning in intrusion detection systems. *Machine Learning*, 81(2):179–205, 2010.

[86] Michael Spreitzenbarth. *Dissecting the Doid: Forensic Analysis of Android and its Malicious Applications*. Phd thesis, Department of Computer Science, Friedrich-Alexander-University, Erlangen, Germany, 2013.

[87] State. review of bruen forcinito, cryptography, information theory, and error-correction: A handbook for the 21st century (wiley-interscience, 2004). *COMPREVS: ACM Computing Reviews*, 47, 2006.

[88] symantec. Dialer.trojan. `http://www.symantec.com/security_response/writeup.jsp?docid=2001-010916-4630-99`. January, 2001.

[89] symantec. Wince.pmcryptic. `http://www.symantec.com/connect/blogs/smart-worm-smartphone-wincepmcryptica`. last accessed: March 25, 2013.

[90] Symantec. Backdoor. brador. a. http://www.symantec.com/security_response/writeup.jsp?docid=2004-080516-3455-99&tabid=2, 2004.

[91] Ken Thompson. Reflections on trusting trust. chapter Reflections on trusting trust. ACM, New York, NY, USA, 2007.

[92] Raja Vallée-Rai. Soot: A java bytecode optimization framework, 2000.

[93] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[94] Mario Linares Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in android apps: an empirical study. In Premkumar T. Devanbu, Sung Kim, and Martin Pinzger, editors, *MSR*, pages 2–11. ACM, 2014.

[95] A. Watson and T. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Technical report, National Institute of Standards and Technology, Washington, D.C., 1996.

[96] Hoeteck Wee. On Obfuscating Point Functions. In *STOC '05 (Proceedings of the thirty-seventh annual ACM symposium on Theory of computing)*, pages 523–532, Baltimore, MD, USA, April 2005. ACM, NY.

[97] Cui Xiang, Fang Binxing, Yin Lihua, Liu Xiaoyi, and Zang Tianning. Andbot: towards advanced mobile botnets. In *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats*, LEET'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[98] Sheng Yu and Shijie Zhou. A survey on metric of software complexity. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*. IEEE, 2010.

[99] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. ADAM: An automatic and extensible platform to stress test android anti-virus systems. In Ulrich Flegel, Evangelos P. Markatos, and William K. Robertson, editors, *DIMVA*, volume 7591 of *Lecture Notes in Computer Science*, pages 82–101. Springer, 2012.

[100] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In Elisa Bertino and Ravi S. Sandhu, editors, *CODASPY*, pages 317–326. ACM, 2012.

[101] Yan Zhuang and Felix Freiling. Approximating Optimal Software Obfuscation for Android Applications. In Günther Pernul, Guido Schryen, and Rolf Schillinger, editors, *Proceedings 2nd Workshop on Security in Highly Connected IT Systems*, pages 46–50, Piscataway, NJ, USA, 2015.

[102] Yan Zhuang, Mykola Protsenko, Tilo Müller, and Felix C. Freiling. An(other) exercise in measuring the strength of source code obfuscation. In *DEXA Workshops*, pages 313–317. IEEE, 2014.

[103] H. Zuse and P. Bollmann. Software metrics: Using measurement theory to describe the properties and scales of static software complexity metrics. *SIGPLAN Not.*, 24(8):23–33, August 1989.