BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2020 Bc. Pavol Michalec

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

SECURITY TESTING OF OBFUSCATED ANDROID APPLICATIONS

BEZPEČNOSTNÍ TESTOVÁNÍ OBFUSKOVANÝCH ANDROID APLIKACÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Pavol Michalec**
AUTOR PRÁCE

SUPERVISOR — **doc. Ing. Lukáš Malina, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Informační bezpečnost**

Ústav telekomunikací

Student: Bc. Pavol Michalec ***ID:*** 185935
Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Bezpečnostní testování obfuskovaných Android aplikací

POKYNY PRO VYPRACOVÁNÍ:

V rámci práce se seznamte s metodami obfuskace aplikací na platformě Android. Zaměřte se především na metody obfuskace Proguard a Dexguard. Jednotlivé metody porovnejte a diskutujte o dopadu obfuskace na testování aplikací. Vytvořte analýzu možností efektivního bezpečnostního testování současných obfuskovaných Android aplikací včetně reverzního inženýrství. Popis a analýza se zaměří na současné metody pro obfuskaci a implementace verifikačních aplikací (utilit), které budou demonstrovat způsoby, jak obejít základní ochrany Runtime Application Self-Protection (RASP) jako SSL pinning, detekce emulátoru, superuživatele a debuggerů. Aplikace dále rozšiřte o způsoby, jak obejít pokročilé ochrany obfuskace jako code hardening a posuďte efektivnost těchto ochran. Zaměřte se na funkcionalitu šifrování textových řetězců.

DOPORUČENÁ LITERATURA:

[1] SCHLEIER, Sven, Bernhard MUELLER a Jeroen WILLEMSEN, 2019. OWASP Mobile Security Testing Guide [online]. [cit. 2019-08-09]. Dostupné z:

https://github.com/OWASP/owasp-mstg/releases/download/1.1.3/MSTG-EN.pdf

[2] WERMKE, Dominik, Nicolas HUAMAN, Yasemin ACAR a Bradley REAVES, 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In: Proceedings of the 34th Annual Computer Security Applications Conference on - ACSAC '18 [online]. New York, New York, USA: ACM Press [cit. 2019-08-09]. DOI: 10.1145/3274694.3274726. ISBN 9781450365697. Dostupné z:

http://dl.acm.org/citation.cfm?doid=3274694.3274726

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 1.6.2020

Vedoucí práce: doc. Ing. Lukáš Malina, Ph.D.
Konzultant: Tomáš Ležovič, ESET, spol. s r.o.

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRACT

The Master´s Thesis is about the security testing of obfuscated Android applications. The theoretical part of the thesis describes the basic theory behind obfuscation and mentions several obfuscators. Impact of obfuscation on penetration testing is also mentioned. Dynamic analysis is proposed as the main tool of this thesis to bypass the obfuscation. The practical part of the thesis describes realtime application self-protection controls and a way to bypass them using dynamic analysis. The second section of the practical part is about advanced obfuscation techniques and their bypassing.

KEYWORDS

Android, security, obfuscation, Allatori, DashO, DexProtector, penetration testing, owasp

ABSTRAKT

Diplomová práca je o bezpečnostnom testovaní obfuskovaných Android aplikácií. Teoretická časť práce opisuje základy obfuskácie a spomína niektoré vybrané obfuskátory. Dopad obfuskácie na penetračné testovanie je taktiež zmienený. Práca navrhuje dynamickú analýzu ako hlavný nástroj pri obchádzaní obfuskácie. Praktická časť práce popisuje ochrany aplikácie v reálnom čase a spôsoby, ako tieto ochrany obísť pomocou dynamickej analýzy. Druhá polovica praktickej časti je venovaná pokročilým technikám obfuskácie a spôsobom ich obídenia.

KLÍČOVÁ SLOVA

Android, bezpečnosť, obfuskácia, Allatori, DashO, DexProtector, penetračné testovanie, owasp

MICHALEC, Pavol. *Security Testing of Obfuscated Android Applications*. Brno, 2020, 85 p. Master's Thesis. Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Advised by doc. Ing. Lukáš Malina, Ph.D.

ROZŠÍRENÝ ABSTRAKT

V dnešnej modernej dobe sú mobilné aplikácie každodennou súčasťou našich životov. Takmer každý z nás využíva aplikácie pre prístup k sociálnym sieťam alebo internet bankingu. Tieto aplikácia pracujú s citlivými dáta a často vykonávajú citlivé akcie ako prevod peňazí a podobne. Práve preto je nevyhnutné aby tieto aplikácie neobsahovali bezpečnostné chyby, ktoré by mohli aplikáciu alebo užívateľa kompromitovať. Jedinou cestou, ako zabrániť zraniteľnostiam, je neustále vykonávať bezpečnostné testy.

Vývojári aplikácií často používajú obfuskáciu, aby zamedzili jednoduchej analýze aplikácie. Avšak počas bezpečnostného testu je obfuskácia prekážkou, nakoľko niektoré testy je komplikované vykonať alebo sa nestihnú z časového hľadiska. Cieľom tejto práce je poukázať na testovanie obfuskovaných aplikácií a navrhnúť riešenia, ako k takýmto aplikáciám pristupovať efektívne.

Obfuskácia sa dá chápať ako transformácia vstupného kódu na výstupný, ktorý je obfuskovaný. Cieľom je odstrániť akékoľvek identifikátory, ktoré by mohli pomôcť pri analýze aplikácie. Napríklad sú odstránené názvy premenných a nahradené bezvýznamnými názvami. Obfuskovaný kód je z hľadiska funkcionality zhodný s pôvodným kódom, avšak je ťažšie pochopiteľný, ak ho číta človek.

Ako základný obfuskátor bol kedysi považovaný ProGuard. Ide o bezplatný program určený pre Java aplikácie a funguje aj s platformou Android. V novších verziách vývojového prostredia Android Studio bol nahradený obfuskátorom od spoločnosti Google s názvom R8.

V praxi sa obfuskácia využíva na ochranu aplikácií proti nežiadúcej analýze alebo dekompilovaniu. Bezpečnostné štandardy požadujú, aby každá aplikácia mala aspoň základnú obfuskáciu. Obfuskácia je však často používaná pri škodlivých aplikáciách. Keďže obfuskácia mení vzhľad kódu, ale nie funkčnosť, je možné týmto spôsobom obísť statické analýzy, ako napríklad detekcia malware pomocou známych signatúr alebo odtlačku (fingerprint).

Pri bezpečnostnom testovaní obfuskácia spomaľuje celý proces, čím sa trvanie testu predlžuje a je finančne náročnejší. Často nie je možné za daný čas otestovať aplikáciu, ktorá ma viaceré ochrany. Práca s obfuskovanou aplikáciou je komplikovanejšia a preto je nutné mať k dispozícií riešenia, ktoré umožnia a zjednoduchšia prácu s takýmito aplikáciami.

Pri statickej analýze je aplikácia dekompilovaná na zdrojový kód a tester sa snaží pochopiť jej princípu a fungovaniu. Statická analýza nie je efektívna pri obfuskovaných aplikáciách, keďže obfuskácia sa zameriava na vzhľad kódu a značne komplikuje jeho čítanie a pochopenie. Avšak stále je statická analýza nevyhnutná a je súčasťou každého testovania. Touto metódou je možné odhaliť použité knižnice alebo názvy funkcií, ktoré budú analyzované dynamicky.

Na rozdiel od statickej analýzy, dynamická analýza aplikáciu spúšťa. Keďže funkcionalita kódu je nezmenená, obfuskácia nemá vplyv na dynamickú analýzu. Hlavným nástrojom tejto práce je Frida Toolkit. Ide o nástroj, ktorý dokáže do aplikácie vnášať vlastný kód a počas chodu aplikácie meniť význam jednotlivých metód, prípadne jednotlivé metódy odchytiť a zmeniť alebo vypísať vstupné a výstupné parametre. Práca s Frida Toolkit je priamočiara a rýchla, jednotlivé skripty, ktoré sú vkladané do aplikácie sú písané v jazyku JavaScript, ktorý je jednoduchý a rýchly na vývoj.

Mobilné aplikácie často obsahujú mechanizmy, ktoré v reálnom čase vyhodnocujú prostredie, v ktorom aplikácia beží a kontrolujú ho na možné hrozby. Medzi takéto kontroly často patrí detekcia superužívateľa (root oprávnenia), emulátora alebo prítomnosť debuggera. Prítomnosť niektorého z týchto prvkov by mohla značiť kompromitované zariadenie, čo môže pre aplikáciu predstavovať riziko. Najčastejšie sa však aplikácia chráni pomocou ochrany Certificate Pinning, často označovaná ako SSL Pinning. Ide o ochranu, kedy aplikácia odmietne komunikáciu so serverom, ktorý neobsahuje SSL/TLS certifikát, ktorý má aplikácia v sebe zakomponovaný. Tento mechanizmus však znemožňuje testovanie, nakoľko pri teste je aplikácia vždy napojená na aplikačnú proxy, cez ktorú smeruje komunikácia, aby ju tester mohol vidieť. Aby proxy vedela pracovať aj so šifrovanou komunikáciou, vytvárajú sa dva šifrované kanály, jeden medzi aplikáciou a proxy, druhý medzi proxy a cieľovým serverom. Komunikácia na proxy je teda dešifrovaná a je možné ju modifikovať. Chránená aplikácia pomocou SSL Pinningu však neprijme certifikát od aplikačnej proxy a komunikácia neprebehne, čím je testovanie znemožnené. Taktiež testovanie prebieha výhradne na zariadeniach so superužívateľom, aby mal tester prístup ku všetkým súborom aplikácie a všetkým funkciám. Pre tieto dôvody je nutné tieto mechanizmy pri teste obísť.

Všetky spomenuté mechanizmy je možné obísť pomocou dynamickej analýzy. Často sú tieto mechanizmy implementované pomocou štandardných postupov. Je teda možné predpokladať ako bude vyzerať implementácia v konkrétnej aplikácií. Proces obídenia týchto mechanizmov je jednoduchý. Pomocou Frida Toolkit sa napíše skript, ktorý špecificky obchádza implementáciu daného mechanizmu a tým mu zabraňuje vo funkčnosti, prípadne sa tento mechanizmus oklame, aby si myslel, že všetko beží tak ako má, aj keď je reálne zariadenie upravené. Vďaka dynamickej analýze a Frida Toolkit je možné testovať aj aplikácie, ktoré takéto mechanizmy obsahujú a je ich možné obísť.

Niektoré obfuskátory často poskytujú aj pokročilú funkcionalitu, akou je šifrovanie textových reťazcov alebo napríklad obfuskácia kontrolného toku (control flow) pomocou vnášania nadbytočného kódu alebo viacerých vetvení. Práve textové reťazce často poskytujú testerovi vstupnú bránu do aplikácie, nakoľko je jednoduché

zistiť, aká funkcionalita je poskytovaná danou funkciou. Ak napríklad funkcia obsahuje textový reťazec „Platba úspešná!“, je možné predpokladať, že táto funkcia pracuje s nejakou časťou platobnej funkcionality. Týmto spôsobom sa dá rýchlo a jednoducho lokalizovať hľadaná funkcionalita.

Aplikácie často obsahujú nejakú formu dodatočného zabezpečenia komunikácie so serverom. Často je použitá nejaká forma digitálneho podpisovania požiadaviek. Pre testovanie aplikácie je nevyhnutné otestovať jednotlivé parametre, zasielané v požiadavkách na server, preto je nevyhnutné mať možnosť vytvoriť platný podpis aj pre modifikované dáta.

Pre otestovanie obídenia týchto pokročilých obfuskačných techník bola vytvorená vlastná aplikácia. Táto aplikácia je inšpirovaná mobilným bankovníctvom. Ide teda o aplikáciu, ktorá znázorňuje prevod peňazí medzi užívateľmi. Táto aplikácia bola obfuskovaná pomocou troch pokročilých obfuskátorov a to Allatori, DashO a DexProtector. Cieľom bolo obísť šifrovanie reťazcov a získať možnosť podpisovať ľubovoľné požiadavky. Ako navrhnutá metodológia bola dynamická analýza, konkrétne Frida Toolkit v spolupráci s doplnkom Brida pre známu proxy Burp Suite.

Aplikácia bola analyzovaná vo všetkých troch verziách. Jedna verzia pre každý použitý obfuskátor. Statickou analýzou bol porovnaný vzhľad kódu a jeho zrozumiteľnosť. Taktiež bola identifikovaná metóda, ktorá obsluhuje dešifrovanie šifrovaných textových reťazcov. Po tejto analýze bola identifikovaná metóda, ktorá generuje podpis/odtlačok pre jednotlivé požiadavky. DahO aj DexProtector využívali na obfuskáciu aj štandardný obfuskátor R8.

V prípade Allatori aj DashO bol priebeh podobný. Statickou analýzou boli objavené metódy pre dešifrovanie textových reťazcov, ktoré boli pomocou Frida Toolkit napichnuté (hooked) a pri každom volaní bol ich výstup (dešifrovaný text) zobrazený pre testera. Podobne sa postupovalo pri generovaní platného podpisu. Lokalizovanie tejto metódy je o niečo komplikovanejšie, keďže metódy majú zmenené názvy, avšak nepredstavuje to veľký problém. Táto metóda využíva kryptografické volania pre prácu s algoritmom HMAC. Nájdením metódy, ktorá využíva tieto volania bola objavená hľadaná metóda. Po lokalizovaní bol napísaný Frida skript, ktorý umožnil testerovi ľubovoľne využívať túto funkciu podľa potreby.

DexProtector bol odlišný od zvyšku obfuskátorov, keďže podporoval šifrovanie celých tried a teda neboli k dispozícií dekompilované zdrojové kódy. Šifrovanie textových reťazcov nebolo podporované. Bez prístupu k statickej analýze bol zvolený slepý postup. Pomocou predpokladov a stack-trace výpisov bolo možné detegovať správnu metódu, ktorá bola následne napichnutá a bolo možné ju využívať.

Tieto postupy demonštrovali obídenie aj pokročilých obfuskačných techník a dynamická analýza sa ukázala ako silný nástroj v rukách testera pri testovaní tohto typu aplikácií.

DECLARATION

I declare that I have written the Master's Thesis titled "Security Testing of Obfuscated Android Applications" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno ....

author's signature

ACKNOWLEDGEMENT

I would like to thank the supervisor of this Master´s Thesis, Mr. doc. Ing. Lukáš Malina, Ph.D. for his supervision, consultations, patience and comments. I would also like to thank my consultant from the company ESET, spol. s.r.o., Mr. Tomáš Ležovič for his time, help and consultations. Special thanks to Smardec Inc., PreEmptive Solutions, LLC. and Licel Corporation for providing free access to their respective products.

Tato práce vznikla jako součást klíčové aktivity KA6 - Individuální výuka a zapojení studentů bakalářských a magisterských studijních programů do výzkumu v rámci projektu OP VVV Vytvoření double-degree doktorského studijního programu Elektronika a informační technologie a vytvoření doktorského studijního programu Informační bezpečnost, reg. č. CZ.02.2.69/0.0/0.0/16_018/0002575.

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Projekt je spolufinancován Evropskou unií.

Contents

List of Figures

List of Tables

List of Listings

Introduction

Nowadays, Android applications are more and more common. This includes applications like social media or mobile banking applications. These can contain sensitive information and also perform critical actions like money transfers. Developers of the applications need to have these applications tested for security vulnerabilities as it is the only method to lower the risk of security issues.

Security testing of mobile applications can be relatively straightforward thanks to available testing standards. The issue arises, when the developers are unable to provide an unobfuscated application to the security testers. Bypassing obfuscation can be a tricky and time-consuming process.

The goal of this thesis is to provide the necessary information about obfuscation so the testers can fully understand their enemy. Also, effective tools like dynamic instrumentation are shown as they can sometimes completely bypass obfuscation in some instances.

The first chapter is about the basics of obfuscation on the Android platform and provides details about the reason that obfuscation can be problematic in security testing.

The Second chapter discusses static and dynamic analysis methods to show the advantages and disadvantages of both. Frida toolkit is presented as the primary dynamic analysis/instrumentation framework.

The next chapter presents basic protections that the applications usually include, like SSL pinning or root detection. These need to be bypassed during the test as they would hinder the ability to test the application properly. Methods, how to bypass these protections, using Frida toolkit, are described in detail in chapter 4.

The final two chapters describe advanced obfuscation techniques and methods, how to work around them. These chapters required the use of custom test application to be analysed and to demonstrate the power of the dynamic instrumentation.

1 Obfuscation

Android uses Java as its primary programming language for applications. Java is not directly compiled into assembly code. Instead, it compiles into *byte-code*. This means that Java application is relatively easy to decompile compared to standard assembly code like C/C++ [1]. Java byte-code contains information about fields, method return values and parameters. Variable names and method/class names (identifiers) are also preserved.

Obfuscation is a process of transforming input code I to target code T and can be written as $T = f(I)$ [2]. From the user's perspective, both I and T must be identical (obfuscated application must work the same as non-obfuscated). The only difference is that the obfuscated application looks different (garbled) when decompiled by the reverse engineer. Some obfuscators also add additional junk code, which does not have any functionality and its only purpose is to confuse the reverse engineer.

Main techniques used for obfuscation are: [2]

- Identifier renaming – developers tend to use descriptive names for variables and classes. This information can be helpful during the reverse engineering process. Obfuscator strips away this data and replaces identifiers with meaningless names.
- String encryption – certain obfuscators can encrypt hardcoded strings. These strings are decrypted during runtime.
- Java reflection – reflection is a feature of Java and allows developers to create classes and call methods dynamically using *getMethod()* or *invoke()* calls.

For better understanding, consider code in listing 1. The code contains function

```
1 public KeyPair generateKeyPairForSigning(String alias, String algorithm) throws
  ↪ NoSuchProviderException, NoSuchAlgorithmException,
  ↪ InvalidAlgorithmParameterException {
2     KeyPairGenerator kpg = KeyPairGenerator.getInstance(algorithm,
      ↪ "AndroidKeyStore");
3     kpg.initialize(new KeyGenParameterSpec.Builder(alias,
4             KeyProperties.PURPOSE_SIGN | KeyProperties.PURPOSE_VERIFY)
5             .setDigests(KeyProperties.DIGEST_SHA256,
6                     KeyProperties.DIGEST_SHA512)
7             .build());
8     return kpg.generateKeyPair();
9 }
```

Listing 1: Original code without obfuscation.

named *generateKeyPairForSigning*, which takes two string parameters called *alias*

and *algorithm*. The purpose of this method is to create a key pair for digital signature and verification with the specified alias and using the specified algorithm. For the illustration purposes, code is being compiled without optimizations (*-donotoptimize* flag), and ProGuard with default settings is used for obfuscation.

Without the use of obfuscation, all identifier names are retained (listing 2) and

```
1 public KeyPair generateKeyPairForSigning(String alias, String algorithm) throws
  ↪ NoSuchProviderException, NoSuchAlgorithmException,
  ↪ InvalidAlgorithmParameterException {
2     KeyPairGenerator kpg = KeyPairGenerator.getInstance(algorithm,
      ↪ "AndroidKeyStore");
3     kpg.initialize(new Builder(alias, 12).setDigests(new String[]{"SHA-256",
      ↪ "SHA-512"}).build());
4     return kpg.generateKeyPair();
5 }
```

Listing 2: Decompiled code without obfuscation.

the decompiled code is almost the same as original. With ProGuard obfuscation, decompiled code has renamed identifiers with meaningless names like *a* or *str1* (listing 3). A simple function like this can still be reverse engineered easily, but more complicated code can have a function calls like *c.a.a.a.b()*, which can be frustrating to work with as the reverse engineer.

```
1 public KeyPair a(String str, String str2) {
2     KeyPairGenerator instance = KeyPairGenerator.getInstance(str2,
      ↪ "AndroidKeyStore");
3     instance.initialize(new Builder(str, 12).setDigests(new String[]{"SHA-256",
      ↪ "SHA-512"}).build());
4     return instance.generateKeyPair();
5 }
```

Listing 3: Decompiled code with Proguard obfuscation.

1.1 Android Build Process

Understanding of build process for an Android application is not necessary for the obfuscation purpose itself but is required for reverse-engineering. Android used to use three different tools to produce its final format (*.dex* files – Dalvik Executable) [2]:

- javac – standard Java compilator, creates Java byte-code from the source code.
- ProGuard – shrinks, optimizes, obfuscates and preverifies Java byte-code that was generated by javac (more on this in section 1.2).
- DX – compiles Java byte-code into Android-specific byte-code (Dalvik format[1]).

This build process can be seen in figure 1.1[4].

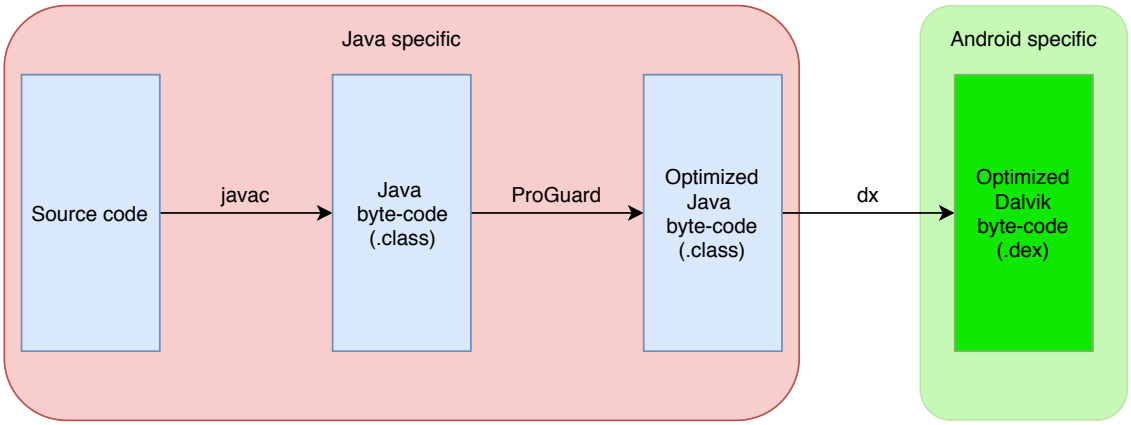

Fig. 1.1: Old build process for Android applications.

Nowadays, the build process is a bit different. Google recognized the downsides of using three separate tools in the build pipeline and decided to streamline the process [4]. After multiple attempts with *Jack and Jill* and *D8* compilers/optimizers, Google finally produced a replacement for *dx* and also for ProGuard – R8. It acts as an optimizer/shrinker/obfuscator and also as a compilator/transformer of Java byte-code into Dalvik executable format (.dex). R8 is an extension of D8 (D8 only transforms Java byte-code into dex format, R8 does the same, but can also optimize the byte-code). This new build process can be seen in figure 1.2. R8 is backwards compatible with ProGuard rules.

1.2 ProGuard

ProGuard[2] used to be the default class file shrinker, optimizer, obfuscator and preverifier shipped with Android SDK and Android Studio. It is a general tool for Java, not specifically for Android applications. ProGuard uses these steps to create output jars out of input jars [5]:

- Shrinking – removes unused classes, fields, methods and attributes effectively resulting in smaller output jar.

[1]Dalvik virtual machine that Android used was superseded by Android Runtime (ART). However, applications are still being built into Dalvik format and are compiled into ART format on-device [3].

[2]https://www.guardsquare.com/en/products/proguard

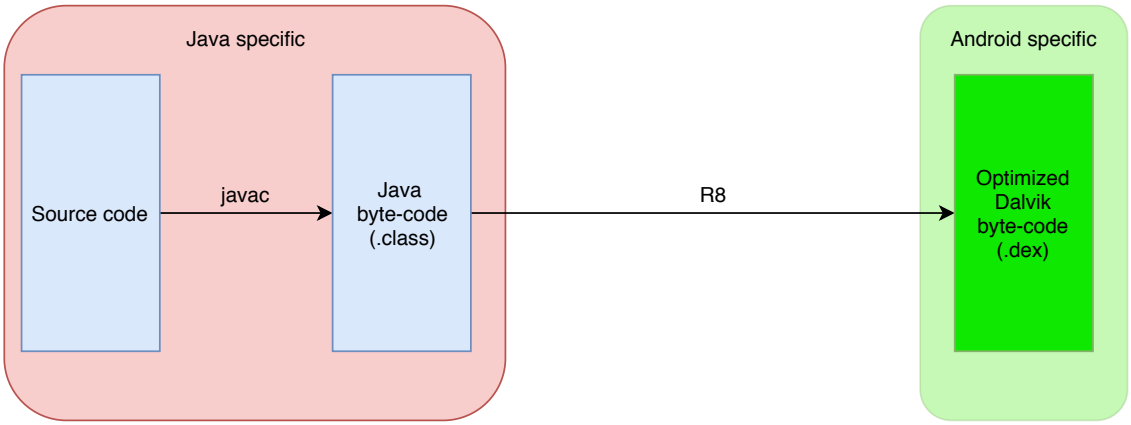

Fig. 1.2: Build process for Android applications using the new R8.

- Optimizing – optimizes byte-code of methods.
- Obfuscation – renames identifiers to meaningless names.
- Preverification – a requirement for Java 6 and higher (also for Java Micro Edition).

This whole process can be seen in figure 1.3.

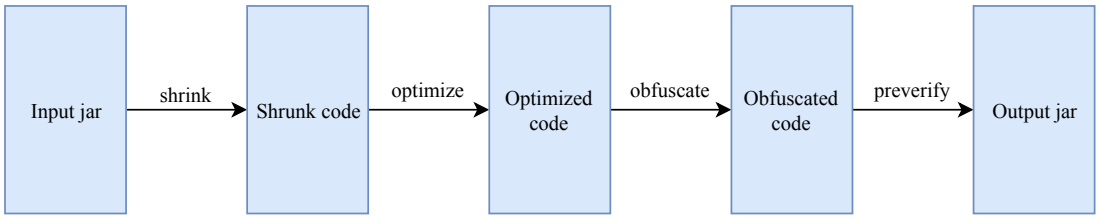

Fig. 1.3: ProGuard workflow.

ProGuard as obfuscator provides only identifier renaming, advanced protections like string encryption or RASP (Runtime Application Self-Protection) are not included. On the other hand, ProGuard is open-source and free to use for both personal and commercial use.

1.3 DexGuard

Both ProGuard and DexGuard[3] are made by the same company (Guardsquare). While ProGuard is an optimizer for Java byte-code, DexGuard is specific for Android [6]. This allows DexGuard to implement advanced features, some of which include:

- Better optimization.

[3]https://www.guardsquare.com/en/products/dexguard

- RASP – includes protections like SSL pinning, debugger detection or rooted Android device.
- Provides advanced obfuscation – unlike ProGuard, DexGuard also obfuscates arithmetic and logical expressions as well as control flow.
- Encryption – DexGuard can encrypt hardcoded strings (these are decrypted during runtime) and also encrypt names of sensitive classes or include reflection to sensitive method calls.
- Protection of all components – ProGuard only obfuscates byte-code, DexGuard can obfuscate/encrypt/optimize other components like manifest file, native libraries, resources and assets.

DexGuard is a commercial tool and requires a paid license to use.

1.4 Usage of Obfuscation in the Real World

This work differentiates two main categories of applications, which use obfuscation – benign applications and malware.

1.4.1 Benign Applications

Benign applications use obfuscation mostly for the following reasons [7]:

- protection of proprietary or sensitive algorithms inside the application,
- multimedia resources protection for piracy reduction (DRM – Digital Rights Management),
- shrinking the application makes them smaller and can run faster.

Statistically, approximately 25% [8] of Android applications are obfuscated (sample of 1,762,868 free applications from Google Play store). ProGuard was the most popular obfuscator, which is understandable as it is the default one. Different research shows that almost 43% [9] of applications in the Google Play store are obfuscated (however, this research had a smaller sample size, only 26,614 applications). For third-party stores (mostly from China), this number rises to 73% [9] (sample of 65,666). This is mostly due to piracy and application cloning threats in China, so protection of source code is more relevant for developers.

Research [8] also proved that applications with higher download count are more obfuscated. For example, applications with 10,000+ downloads are obfuscated in approx. 24% of cases. However, in approx. 50% of applications with 100,000,000+ downloads are obfuscated. A similar trend can be seen with the number of applications per account. Developers with 1-10 released applications mostly do not obfuscate their apps (on average, only ≈21-22%). On the other hand, developers with 500+ released applications have, on average ≈68,41% of obfuscated applications.

It can be concluded that the more applications the developer have, the more likely it is that his/her applications will be obfuscated. The same thing applies for application popularity, more successful application results in higher obfuscation chance.

1.4.2 Malware

The other big group of applications is malware. These applications use obfuscation to [7]:

- hide malicious payloads from anti-malware scanners,
- prevent or delay human analysis of the malware.

Android devices (and mobile devices in general) are performance-limited devices mostly powered by a battery. This means that advanced malware detection techniques are limited by both power and performance. A lot of anti-malware software use signature-based detection (static analysis). In general, this technique works by "fingerprinting" the application and comparing it with the list of known malware. Obfuscation can usually bypass this detection as one malware can be transformed into many applications (each with its signature) using obfuscation. For example, obfuscation can rename classes based on rules, and these rules can be changed for each application. Junk code insertion is also an option (many times using *goto* statements, which alters the control flow and the call graph of the application).

Obfuscation also delays the analysis of malware because it takes longer for the human reverse-engineer to understand the workflow of the malware and create signatures or other forms of the detection technique. The only way around these issues and obfuscation, in general, is dynamic analysis (as obfuscation has no impact on the execution of the application).

Research on obfuscation usage in malware applications [9] showed that ≈64% of malware is obfuscated (authors of the research believe that the rest are entry-level malware authors without proper knowledge or false positives in malware detection). This result makes sense because obfuscation is essential for malware authors to prevent early detection, which allows the malware to spread and infect more devices.

1.5 Impact of Obfuscation on Security Testing

As for the protection of proprietary or sensitive algorithms, consider this real-world example. Mobile banking application, which allows clients to execute money transfers. These transfers require client-server communication, which can easily be tampered with by the attacker using standard application proxy like Burp Suite[4]

[4]https://portswigger.net/burp

(and bypassing SSL pinning, more on this topic later). To prevent modification of data sent by the attacker, digital signatures together with encryption are used in (usually) custom way. Mobile banking applications are sensitive because they handle sensitive data (money). Usually, this means that banks want to have their applications tested for security vulnerabilities before release. Some banks will provide both obfuscated and unobfuscated (debug) versions to security testers (as they should). However, this is not always the case. Some banks refuse to provide unobfuscated versions for penetration tests, and it is up to penetration testers to still test the application properly. The only way to properly verify that the signing/encryption algorithm used for money transactions is not susceptible to tampering is by understanding how it works (security by obscurity does not work). This is a significant problem if the application is obfuscated and understanding to decompiled code is more complicated (especially with encryption algorithms, which usually use a lot of mathematical operations). Penetration tests of mobile applications (or any application) are expensive, and clients do not want to spend much money. As such, these tests usually take only several days. This is not much time, especially when it is needed to test the communication of the application with backend server as well as the security of the application itself. In this case, obfuscation prevents testers to accurately and in-depth test the entire application.

2 Analysis of Effective Security Testing and Reverse Engineering

In order to test the application, testers have multiple ways to test the application. These primarily include static and dynamic analysis. Both have their pros and cons. Usually, these are used together to create one complex analysis. This text presumes obfuscated application and as such only black-box testing will be considered (without access to original/unobfuscated source code). To obtain the source code of the application, the decompilation of the apk file is required.

2.1 Disassembling and Decompilation

Android applications can be disassembled and decompiled back into Java. Application in apk format can be obtained from the application developer, from the device (requires root) or can be downloaded from websites like *apkpure.com* (requires caution, potentially can contain malware). There are many tools to decompile the apk file. The easiest way is to use jadx-gui[1], which disassembles and decompiles the application and shows the source code in one click.

Jadx-gui does not allow editing the source code, only reading. To fully decompile the application on the file system, apktool[2] or apkx[3] can be used. Apkx disassembles and decompiles the application straight into Java (apktool requires an additional step to convert dex files into Java). The decompiled application can be imported into Android Studio (or other IDE – Integrated Development Environment), which provides quality of life features like refactoring. After making required changes to the source code (like enabling debugging flag or SSL pinning bypass), the application needs to be repackaged back into apk format. Apktool can be used once again. It creates apk file, but modifications to the source code mean that the digital signature inside the apk file will not be correct. Apk file needs to be re-signed with our signing key (self-signed RSA key is sufficient). After re-signing, the application can be installed onto an Android device.

2.2 Static Analysis

Also known as code review, static analysis revolves around reading the decompiled source code and trying to understand its purpose. With obfuscation, this is harder

[1] https://github.com/skylot/jadx
[2] https://ibotpeaches.github.io/Apktool/
[3] https://github.com/b-mueller/apkx

because identifiers like class or variable names are changed to meaningless names. It is best to use IDE like Android Studio or JetBrains IntelliJ as these tools provide testers with the ability to rename identifiers and tracking down method calls.

Manual static analysis of the obfuscated application is based on going through the source code of the application [10] (ideally from the beginning – MainActivity) and trying to understand the code and rename every method call and variable on the way to a more meaningful name. With small and simple applications, this process is doable in a reasonable amount of time, but more complex applications (with a lot of functionality) would take too much time to analyze and understand. This is because obfuscation primarily targets source code and as such hinders static analysis a lot.

Several tools exist for automatic analysis of android applications like MobSF[4] or QARK[5]. These tools are similar to automatic vulnerability scanners, and their output should not be considered reliable. At the same time, these tools can direct the tester in a specific way or find easy-to-find vulnerabilities like the debuggable flag or similar. Automatic tools are also useful to list strings or libraries inside the application. The tester should always do a manual review of the generated report.

2.2.1 Native Libraries

Android applications can contain native libraries written in C/C++. Compared to Java libraries, native libraries are faster as they are directly compiled to assembly language. Android/Java part of the application communicates with the native library over the Java Native Interface (JNI) [10]. The benefit of using native libraries is re-usability of already written applications (only small changes to the library are required for JNI to function correctly).

Native libraries can be loaded into disassemblers like IDA Pro, radare2 or Ghidra. Assembly language is harder to understand than decompiled Java language. Some disassemblers can attempt to recreate the original C/C++ source code (or its representation), but this code can many times be far from the original. Malware authors tend to use native libraries to slow down the manual analysis of the sample.

2.3 Dynamic Analysis

The dynamic analysis does not rely on the reading of the source code. Instead, the application is running and its behaviour is observed [10]. Compared to static analysis, dynamic analysis allows the easier discovery of specific vulnerabilities, for example, data storage vulnerabilities (it is easy to look inside shared preferences or

[4]https://github.com/MobSF/Mobile-Security-Framework-MobSF
[5]https://github.com/linkedin/qark

on the file system what information is being stored by the application and if this information is protected). Some vulnerabilities can only be found during dynamic analysis like vulnerable communication with the server.

It is also possible to debug an Android application during runtime. Android provides two different types of debugging:

- Java Debug Wire Protocol (JDWP) – protocol for communication between Java Virtual Machine and the debugger
- Linux/Unix-style ptrace-based debugging – mostly used to debug native libraries

Unless the tested application contains many native libraries, JDWP is usually used. JDWP allows to set breakpoints, step through the Java code and inspect/modify variables. Standard Java debugger *jdb* is often used. Adding decompiled sources into IDE like Android Studio allows using standard debugging techniques, which are used during development. For example, setting the breakpoint directly in the GUI and on the specific line rather than using a command-line interface with jdb. The only downside is that this technique may not work as the decompiled sources are not the same as the original sources [10]. Both techniques work well and can be utilized.

To allow debugging, the Android application needs to have *android:debuggable = "true"* in its manifest file. However, the manifest file can be modified by the tester, and the debuggable flag can be manually inserted.

As obfuscation targets the source code, dynamic analysis is not hindered by obfuscation. This work mostly revolves around dynamic analysis rather than static. The following chapters will contain more details into specific methods used as well as examples.

2.4 Frida Toolkit

Frida[6] is a free, open-source dynamic instrumentation toolkit written in C language. It works by injecting JavaScript engine (Duktape and V8) into the instrumented process [10], which allows the execution of JavaScript code inside application. The injected code is written directly into process memory. The architecture of Frida can be seen in figure 2.1.

Frida works in three different modes of operation:

- Injected – frida-server needs to be running as a daemon on the device (requires rooted device). This is the most common mode. No modification to the application itself is necessary.

[6] https://www.frida.re/

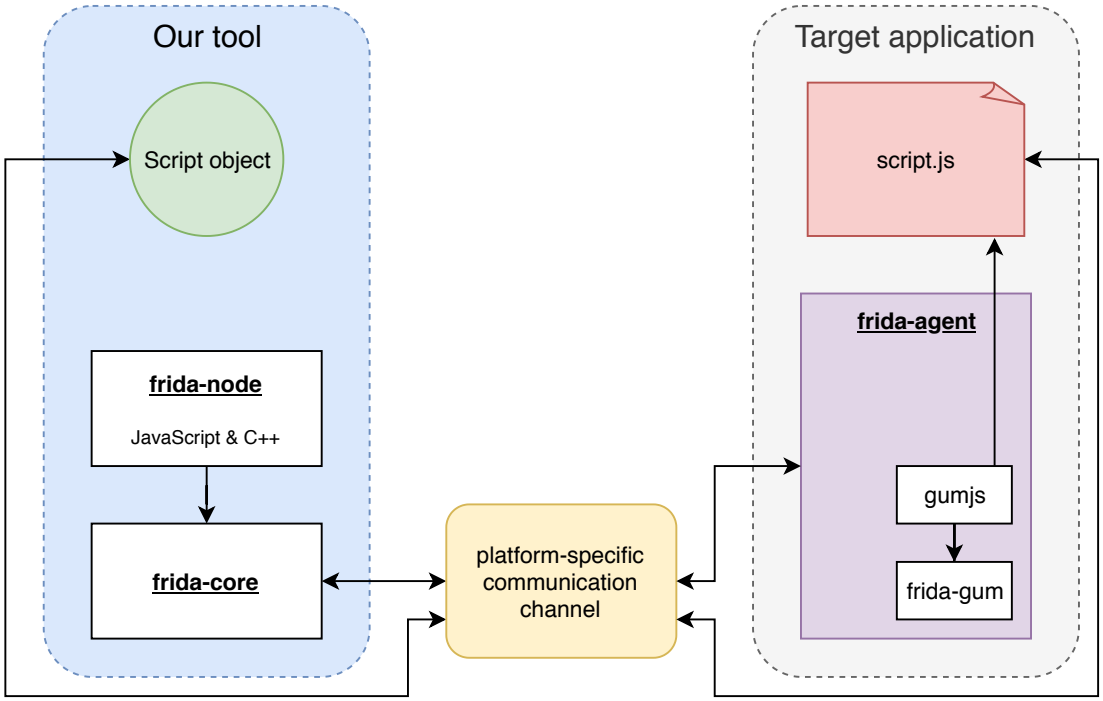

Fig. 2.1: Frida architecture.

- Embedded – frida-gadget library needs to be injected into the application itself by embedding it into it manually. Works on non-rooted devices but requires modification to the application.
- Preloaded – frida-gadget library can be configured to run autonomously by loading scripts from the filesystem.

When Frida is attached to a running application, it uses *ptrace* to hijack a thread of the process and uses it to allocate memory and populate it with mini-bootstrapper [10]. Bootstrapper starts a new thread and connects to frida-server running on the device. The shared library is loaded that contains the Frida agent (frida-agent.so). The agent creates a bi-directional communication channel (platform-specific) back to our tool (either Frida REPL or custom script). Hijacked thread resumes normally.

Frida supports JavaScript for injected code and Python to write Frida tools. A combination of Python and JavaScript was chosen for quick development and risk-free API. Frida tools can also be written directly in C or in one of the language binding on top of C (Node.js, Swift, .NET, Qml, etc.).

On the Android platform, Frida provides several features [10]:

- Create Java Objects, call static/non-static class methods.
- Change Java function implementation.
- Enumerate instances of the specified class.
- Search for a string inside process memory.

- Intercept calls to native functions at entry and exit.
- Supports modification of native code.

Several tools are built on top of Frida such as Fridump[7] (memory dumping tool) or Objection[8] (runtime mobile exploration toolkit).

2.4.1 Example Usage

To demonstrate the power of Frida, "crackme" developed by OWASP is used. The application name is UnCrackable-Level1 and can be downloaded from github[9]. It is a straightforward application with one text area for user input and a button to verify this input. String inputted by the user is checked against the predefined string and if the correct string was inputted, the application prints success message. A failed attempt is visible in figure 2.2.

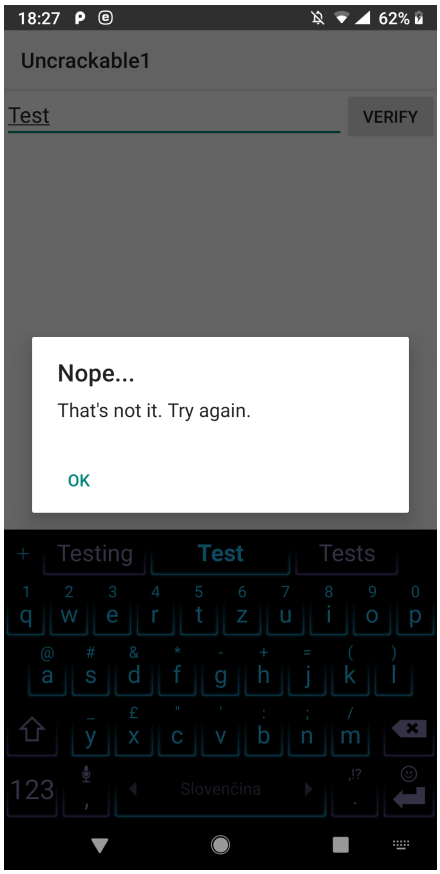

Fig. 2.2: UnCrackable-Level1 GUI.

To keep it simple and short, only relevant parts will be shown. Input from the user is sent to the method (listing 4) *sg.vantagepoint.uncrackable1.a.a* (notice that

[7]https://github.com/Nightbringer21/fridump
[8]https://github.com/sensepost/objection
[9]https://github.com/OWASP/owasp-mstg/blob/master/Crackmes/Android/Level_01

the names are obfuscated). In the return statement (line 14), string *str* is checked against sting created from byte array *bArr*. String *str* is the input from the user and *bArr* is the byte representation of the secret. On line 6, byte array *bArr* is populated with data by calling method *sg.vantagepoint.a.a.a* with two parameters. The first parameter is the secret key used to decrypt the second parameter. Inside the method, AES decryption is performed. It would be possible to manually follow these methods and decrypt the given strings manually, but it would be time-consuming.

```
1  public static boolean a(String str) {
2      byte[] bArr;
3      String str2 = "8d127684cbc37c17616d806cf50473cc";
4      byte[] bArr2 = new byte[0];
5      try {
6          bArr = sg.vantagepoint.a.a.a(b(str2),
           ↪  Base64.decode("5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", 0));
7      } catch (Exception e) {
8          StringBuilder sb = new StringBuilder();
9          sb.append("AES error:");
10         sb.append(e.getMessage());
11         Log.d("CodeCheck", sb.toString());
12         bArr = bArr2;
13     }
14     return str.equals(new String(bArr));
15 }
```

Listing 4: Input verification function.

Frida can be utilized and method *sg.vantagepoint.a.a.a* can be hooked to print the return value (which is the secret value). To hook the method, a script written in javascript (listing 5) is created and injected into the process. This script executes immediately after the application is started (because of *setImmediate()* function). It first finds class *sg.vantagepoint.a.a* and then finds the implementation of its method *a* and replaces it (line 5) with our own method. The first line of the modified method (line 6) is a call to the original method (to obtain the decrypted secret – byte array). The following lines convert each byte value inside byte array into string characters and then prints the entire string (in this case the secret value). Noticed that the method was changed, however, we also call the original method and return its value, so the workflow of the application is not modified (calling this modified method will still return the indented value, so the application works as normal).

Listing 6 shows the execution of the Frida hook and also its output (application needs to run and button inside the application must be clicked with random input).

```
setImmediate(function() {
    console.log("[*] Script started");
    Java.perform(function() {
        class = Java.use("sg.vantagepoint.a.a");
        class.a.implementation = function(key, encryptedSecret) { //change the
        ↪  implementation of the method
            returnValue = this.a(key, encryptedSecret); //call to original
            ↪  method
            secret = ''
            for(i = 0; i < returnValue.length; i++) {
               secret += String.fromCharCode(retval[i]); //use return value of
               ↪  the original method and convert from byte array to String
            }
            console.log("[*] Secret: " + secret); //print the secret
            return returnValue;
        }
    });
});
```

Listing 5: Frida script to print the secret value.

The decrypted secret is *I want to believe*. To ensure this is the correct secret, we input it into the application and success message appears (figure 2.3).

```
frida -U -l uncrackable1.js owasp.mstg.uncrackable1

[*] Script started
[*] Secret: I want to believe
```

Listing 6: Output of the Frida script.

2.5 OWASP Mobile Security Testing Guide

Companies use certain standards for security testing of mobile applications. One of such standards is *OWASP Mobile Security Testing Guide* [10] (MSTG). It is a comprehensive manual for mobile application security testing (both for iOS and Android). As is common with OWASP, anyone can contribute to this project. MSTG is created by many contributors, reviewers and editors around the world and can be used for free.

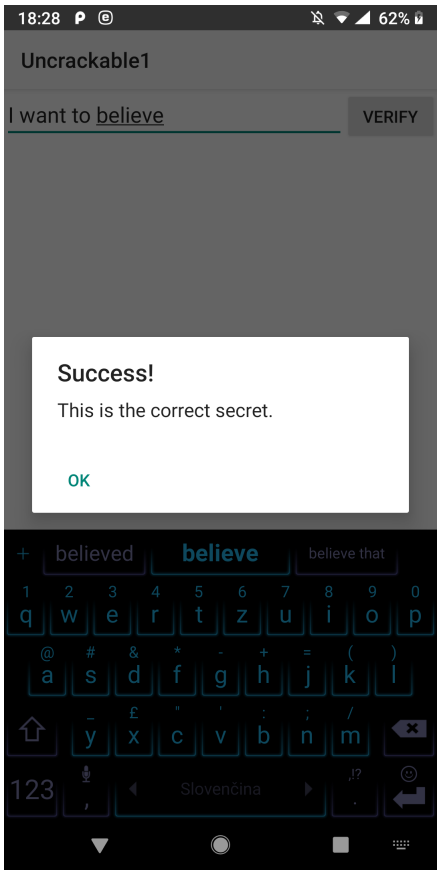

Fig. 2.3: Verification of the obtained secret.

The first part of the MSTG is about the general (platform-independent) security testing. This chapter applies to both iOS and Android applications and consists of sections regarding authentication architectures, network communication, cryptography or reverse engineering.

The main part is the platform-dependent chapter (either iOS or Android). This thesis will not discuss the iOS part. Android part of the testing guide starts with the basics of Android and its architecture. MSTG is written very in-depth and provides both theoretical knowledge with practical examples (often also snippets of code). MSTG describes these major points in Android section:

- Platform overview – provides basic knowledge of Android platform.
- Testing environment setup – describes how to setup the testing environment with both physical or virtual device. Also specifies tools used throughout the testing guide and how to install them.
- Data storage – talks about the principles of secure data storage, which data to store where and how and what data not to store anywhere.
- Cryptographic APIs – best practices regarding cryptographic APIs and also specifies common flaws and incorrect usage.

- Local authentication – describes how the applications should locally authenticate the user.
- Network and platform APIs – provides details on how the application should use network and platform APIs, including endpoint verification (certificates) or Android permissions.
- Code quality and build settings – details about application signatures, debug flags and more.
- Tampering and reverse engineering – provides information about the reverse engineering of Android applications and modifications. This section is the primary source of information for this thesis.
- Anti-reversing defenses – describes how the application can and should defend itself against reverse-engineering.

Related to MSTG is the *Mobile Application Security Verification Standard*[10] (OWASP MASVS). Compared to testing guide, MASVS is more of a checklist-style and is based on the testing guide. It also specifies three security levels for mobile applications:

- Level 1 (MASVS-L1) – security controls that every application should have.
- Level 2 (MASVS-L2) – additional defense-in-depth controls for applications that handle highly sensitive data (like mobile banking applications).
- Resiliency against reverse engineering and tampering (MASVS-R) – can be used with L1 or L2 as an addition.

[10] https://github.com/OWASP/owasp-masvs

3 Realtime Application Self-Protection

Realtime application self-protection (RASP) aims to ensure the security of the application during runtime, even on compromised devices or during attacks on the application itself. The device can be compromised (or insecure) due to numerous reasons. For example, the device can be old and the manufacturer no longer publishes security or operating system (OS) updates. The user can also root his Android device, which increases the attack surface. Even fully updated mobile devices can still be targeted by newly discovered vulnerabilities or attacks [11]. RASP also protects the application against the attacker that may attempt to gain unauthorized access into the application or to data used by the application.

RASP involves many protections, and this thesis will go into detail with these (the most commonly seen in the wild):

- Certificate pinning.
- Root detection.
- Emulator detection.
- Debugger detection.

3.1 Certificate Pinning

One of the most used RASP protection is certificate pinning (usually called SSL pinning). Many applications nowadays use this technique (from the experience of ESET security specialists, almost every application undergoing security tests is protected by SSL pinning).

SSL pinning is an association of the backend server (that the application needs to communicate with) with its SSL/TLS certificate of the public key. This prevents man-in-the-middle attacks that could spoof the certificate to the user and read the entire communication unencrypted [10]. The application will only communicate with the server that has the "pinned" certificate.

This thesis will describe two general (available on Android without external library) configurations for SSL pinning. Other possibilities include custom libraries or library-specific configuration.

3.1.1 Network Security Configuration

Starting with Android 7.0, developers can use *Network Security Configuration* (NSC) to declare public key pins. The configuration file for network security configuration is usually located in *res/xml/network_security_config.xml* (this can be changed, the path needs to be specified in the manifest file. An example of such configuration can

be seen in listing 7. Network communication will work after the expiration date but SSL pinning will be disabled [12]. Using this configuration, developers can choose to trust a certain self-signed certificate (for example their internal certificate authority – CA or certificates used for debugging purposes). By default, Android 7.0 does not trust user-installed certificate authorities (manually installed by the user into the device), only preinstalled ones. Below Android 7.0 user-installed CAs are trusted.

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
<domain-config>
<!-- Use certificate pinning for vutbr.cz website including subdomains-->
<domain includeSubdomains="true">vutbr.cz</domain>
<pin-set expiration="2020/8/28">
<!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
the vutbr.cz website server certificate -->
<pin digest="SHA-256">GA5osZ/jMecigUue4O62Fg3/uClcsstU8VxXIiYUaMk=</pin>

<!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
the Intermediate CA of the vutbr.cz website server certificate -->
<pin digest="SHA-256">XaQOs7GKv4Gx4JRA8ZmihabSl9wxIPx+hQBmJ54WmCs=</pin>

<!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
the Root CA of the vutbr.cz website server certificate -->
<pin digest="SHA-256">WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18=</pin>
</pin-set>
</domain-config>
</network-security-config>
```

Listing 7: Sample public key pin configuration.

NSC can also minimize attack surface regarding CA compromises. The application can be set not to trust any CA except the specified (trusted) ones. For example, developers can only trust their own custom CA and not trust any other public CA. In case of a compromise of specific CA, the application is not affected. It is also possible to forbid unencrypted communication (HTTP) with the specified server.

3.1.2 TrustManager

Android uses a TrustManager component to validate certificates. It is possible to load the CA certificate (that should be trusted) from the file packaged with the application and create KeyStore entry. The application can then create its own

TrustManager with specified KeyStore entry and the supplied certificate will be valid and trusted. Sample process of creating custom TrustManager with specified CA that needs to be trusted by the application can be seen in listing 8.

```
// Load CAs from an InputStream (e.g. from resources)
CertificateFactory cf = CertificateFactory.getInstance("X.509");
InputStream caInput = new BufferedInputStream(new FileInputStream("ca.crt"));
Certificate ca;
try {
    //Create Certificate object
    ca = cf.generateCertificate(caInput);
} finally {
    caInput.close();
}

// Create a KeyStore containing our trusted CAs
String keyStoreType = KeyStore.getDefaultType();
KeyStore keyStore = KeyStore.getInstance(keyStoreType);
keyStore.load(null, null);
// Insert our custom CA into KeyStore
keyStore.setCertificateEntry("ca", ca);

// Create a TrustManager that trusts the CAs in our KeyStore
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);

// Create an SSLContext that uses our TrustManager
SSLContext context = SSLContext.getInstance("TLS");
context.init(null, tmf.getTrustManagers(), null);
```

Listing 8: TrustManager implementation with custom trusted CAs.

3.1.3 Problem with SSL Pinning

Testers use application proxy to intercept network communication between the application and the backend server. Burp Suite or OWASP Zap is usually used. By default, Burp Suite establishes two SSL/TLS connections. One between the client and the proxy, second between the proxy and the requested server. Applications protected with SSL pinning will not accept certificate supplied by the application proxy (TLS handshakes are visible in figure 3.1). In order to intercept network communication using application proxy, SSL pinning must be bypassed.

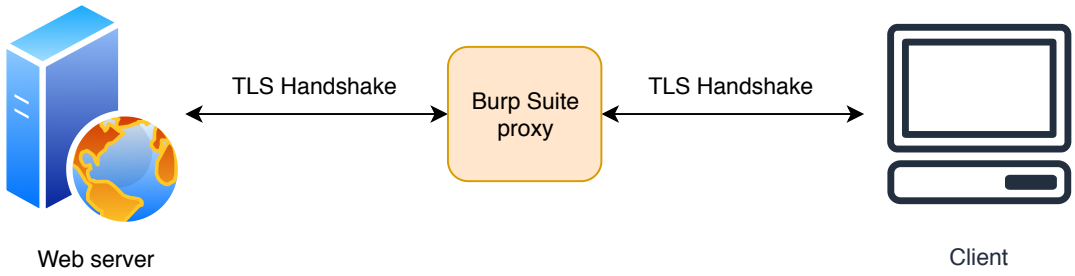

Fig. 3.1: TLS handshake with application proxy

3.2 Root Detection

The goal of root detection is to prevent (or restrict) the application from running on a rooted device. This protects the application itself because the attacker is not able to use specific tools and does not have access to the entire file system. Root detection (just like any other detections on Android) is not very reliable and effective [10]. Multiple root checks should be used.

3.2.1 SafetyNet

SafetyNet (more specifically its Attestation API) provides a set of services to detect security threats to the application like root detection or custom ROM detection. SafetyNet Attestation API assesses the device's integrity by examining the software and hardware environment [13].

SafetyNet downloads binary package (JAR file) from the Google server, validates its integrity and loads it dynamically using reflection. This package attempts to detect if the device is rooted (specific method of how this is achieved is currently unknown [10]). Among root detection, several other checks are conducted like unlocked bootloader, custom system image, device without Google certification, emulator detection and signs of tampering (API hooks).

The result of the attestation is a JSON response from Google server. Sample response is in listing 9. The main parts of the result is "*ctsProfileMatch*" and "*basicIntegrity*". Both values describe the integrity status of the device with the *ctsProfileMatch* being the more stricter. For example devices with unlocked bootloader could pass *basicIntegrity* check but not *ctsProfileMatch*. Rooted devices typically fail *basicIntegrity*.

3.2.2 Programmatic Detection

Android provides several ways for root detection. One of the ways is looking if specific binaries or packages are present on the device. The following is a sample of

```
{
  "timestampMs": 9860437986543,
  "nonce": "R2Rra24fVm5xa2Mg",
  "apkPackageName": "com.package.name.of.requesting.app",
  "apkCertificateDigestSha256": ["base64 encoded, SHA-256 hash of the
                                  certificate used to sign requesting app"],
  "ctsProfileMatch": true,
  "basicIntegrity": true,
}
```

Listing 9: Sample attestation result.

files/directories, which are commonly checked:

- /system/app/Superuser.apk,
- /system/etc/init.d/99SuperSUDaemon,
- /dev/com.koushikdutta.superuser.daemon/,
- /system/xbin/daemonsu.

Also, *su* binary presence is usually checked in different locations (or on the PATH variable by executing it with *Runtime.getRuntime.exec() method*):

- /sbin/su,
- /system/bin/su,
- /system/bin/failsafe/su,
- /system/xbin/su,
- /system/xbin/busybox,
- /system/sd/xbin/su,
- /data/local/su,
- /data/local/xbin/su,
- /data/local/bin/su.

Supersu (rooting tool) runs daemon with the name of *daemonsu* [10]. Checking for this process using *ActivityManager.getRunningAppProcesses* or *getRunningServices* calls. Android package manager allows obtaining a list of installed packaged. This list can be checked for the presence of popular rooting tools (for example com.topjohnwu.magisk).

The different method relies on unusual permissions on system directories. These directories are usually read-only [10] but can be mounted as read-write on a rooted device.

3.3 Emulator Detection

Emulator detection forces the tester to use a physical device instead of an emulator. The physical device is more expensive and harder to root, which barres the access for large-scale device analysis [10].

3.3.1 build.prop

Android provides several APIs to detect the emulated device. Some of the values are located in *build.prop* file (table 3.1 [10], sample only). This file can be modified on a rooted device or during a custom compilation of the Android Open Source Project from the source.

Tab. 3.1: Emulator detection using build.prop.

API Method	**Value**	**Meaning**
Build.ABI	armeabi	possibly emulator
Build.ABI2	unknown	possibly emulator
Build.BOARD	unknown	emulator
Build.Brand	generic	emulator
Build.DEVICE	generic	emulator
Build.FINGERPRINT	generic unknown	emulator
Build.Hardware	goldish ranchu	emulator
Build.Host	android-test	possibly emulator
Build.ID	FRF91	emulator
Build.MANUFACTURER	unknown Genymotion	emulator
Build.MODEL	sdk google_sdk Emulator Android SDK built for x86	emulator
Build.PRODUCT	sdk vbox86p emulator	emulator
Build.RADIO	unknown	possibly emulator
Build.SERIAL	null	emulator
Build.USER	android-build	emulator

3.3.2 TelephonyManager

Android emulators have fixed values in TelephonyManager APIs (table 3.2).

Tab. 3.2: Emulator detection using TelephonyManager.

TelephonyManager API	Value	Meaning
getDeviceId()	0's	emulator
getLine1Number()	155552155	emulator
getNetworkCountryIso()	us	possibly emulator
getNetworkType()	3	possibly emulator
getNetworkOperator() .substring(0,3)	310	possibly emulator
getNetworkOperator() .substring(3)	260	possibly emulator
getPhoneType()	1	possibly emulator
getSimCountryIso()	us	possibly emulator
getSimSerial Number()	89014103211118510720	emulator
getSubscriberId()	310260000000000	emulator
getVoiceMailNumber()	15552175049	emulator

3.4 Debugger Detection

Debugging allows the attacker/tester to analyze the application during runtime and step through the code, inspect variable values, etc. Android applications support two debuggers, JDWP for java code and ptrace-based debugger for native code debugging.

3.4.1 JDWP Debugger Detection

Attaching JDWP debugger to application requires *android:debuggable* attribute set in the manifest file. One of the ways to detect the debugger is to check if this attribute is set using code (*ApplicationInfo.FLAG_DEBUGGABLE*). Another way would be to check if the debugger is already connected using *isDebuggerConnected()* method from the *Debug* class.

Debugging slows down the process execution time. It is possible to check the amount of time the current thread has been executing code. Long execution times indicate attached debugger.

A more complicated way to prevent debugging is to modify JDWP related data structures. It is easier to do in Dalvik than in ART. In Dalvik, setting function pointer *gDvm.methDalvikDdmcServer_dispatch* to NULL is sufficient to crash the JDWP thread. ART is a bit more difficult, and one of the ways to prevent debugging is to overwrite the address of the function *JdwpAdbState::ProcessIncoming* with the address of function *JdwpAdbState::Shutdown*. The debugger will disconnect immediately.

3.4.2 Native Debugger Detection

When the debugger is attached to the application, its *TracerPid* field inside the status file (/proc/self/status) will have a value other than zero. The default value is zero, which means no process attached, so when the ptrace-based debugger is attached to the application, the application will have non-zero TracerPid, which can be considered as a sign of debugging (or something related to the ptrace).

Only one debugger can be attached to the application at any point. This can be exploited by the application. The application can fork child process and attach it to the parent as a debugger to prevent any other debugger from attaching to the main process. It is possible to bypass this functionality just by killing the child process. A more complex way would be to fork multiple processes that attach to one another as debuggers and keeping track of child processes to make sure they are still alive. Values in status file can also be monitored for any unexpected changes.

4 Bypassing RASP

This chapter demonstrates how real-time security controls can be bypassed by dynamic instrumentation. All scripts were tested with Android 10 emulator and Frida version 12.7.26 (Python 3.7). Solutions presented in this chapter are general and will work in most cases, where RASP is implemented using native Android API. Defense mechanisms implemented in custom libraries can not be bypassed by general script, only by bypass specific to the implementation. Custom bypasses will not be presented.

4.1 Certificate Pinning

Section 3.1 described two ways, how to implement SSL/Certificate pinning. Frida is used to bypass both types.

4.1.1 Network Security Configuration

Bypassing network security configuration (NSC) is inspired by related research [14]. The proposed solution (Frida script) has several limitations. First, the user certificate would only be trusted if the application did not use custom NSC file. Second, Android source code has changed and the methods have different parameters now. The excerpt of the relevant Android source code (ManifestConfigSource class) is visible in listing 10. Android is checking if custom NSC file is present in the application (lines 2 and 4). If it is, XmlConfigSource is returned, if it is not present, DefaultConfigSource is returned.

```
1 private ConfigSource getConfigSource() {
2     int configResource = mApplicationInfo.networkSecurityConfigRes;
3     ConfigSource source;
4     if (configResource != 0)
5         source = new XmlConfigSource(mContext, configResource,
          ↪  mApplicationInfo);
6     else
7         source = new DefaultConfigSource(usesCleartextTraffic,
          ↪  mApplicationInfo);
8     return source;
9 }
```

Listing 10: Excerpt of the ManifestConfigSource class.

The first objective is to make sure that the real NSC file is never returned as the source, DefaultConfigSource must always be returned to bypass all the pins specified in the NSC file. DefaultConfigSource does not trust user certificates on Android 7+. The second objective is to ensure that even on Android 7+, user certificates will be trusted.

The first objective can be achieved by modifying the getConfigSource() method to always return DefaultConfigSource. Two parameters are necessary to instantiate a new object of the DefaultConfigSource class. The first parameter is a boolean value if cleartext traffic is allowed, the second is the ApplicationInfo object, which contains information about the application like target the SDK version. Frida script to replace the method functionality is visible in listing 11.

```
ManifestConfigSource.getConfigSource.implementation = function () {
    var context = Java.use('android.app.ActivityThread')
            .currentApplication()
            .getApplicationContext();
    var source = DefaultConfigSource.$new(true,
    ↪  ApplicationInfo.$new(context.getApplicationInfo()));
    return source;
};
```

Listing 11: Frida script to modify getConfigSource method.

The constructor of the DefaultConfigSource class calls getDefaultBuilder method inside the NetworkSecurityConfig class. This builder is responsible for setting up the trusted certificates. Relevant code is visible in listing 12. The code shows that user certificates are only trusted if the application is targeting Android 6 or lower.

```
public static Builder getDefaultBuilder(ApplicationInfo info) {
    if (info.targetSdkVersion <= Build.VERSION_CODES.M &&
    ↪  !info.isPrivilegedApp()) {
        builder.addCertificatesEntryRef(
            new CertificatesEntryRef(UserCertificateSource.getInstance(),
            ↪  false));
    }
    return builder;
}
```

Listing 12: Relevant part of the getDefaultBuilder method.

To bypass this, UserCertificateSource is always manually appended to the builder. First, the original unmodified method is called to instantiate the builder and all the necessary parts. Then, the script appends the UserCertificateSource to the builder. Modified builder is returned and the application will trust user certificates even on Android 7+. This part of the script is visible in listing 13.

```
var NetworkSecurityConfig = Java
    .use("android.security.net.config.NetworkSecurityConfig");
var UserCertificateSource = Java
    .use("android.security.net.config.UserCertificateSource");
var CertificatesEntryRef = Java
    .use("android.security.net.config.CertificatesEntryRef");

NetworkSecurityConfig.getDefaultBuilder
    .overload("android.content.pm.ApplicationInfo")
    .implementation = function (info) {
        var builder = this.getDefaultBuilder.call(this, info);
        builder.addCertificatesEntryRef(CertificatesEntryRef
            .$new(UserCertificateSource.getInstance(), false));
        return builder;
};
```

Listing 13: Frida script to bypass targetSdkVersion check.

Together, these two parts will bypass the custom network security configuration. NSC file will not be checked and user certificates will be trusted.

4.1.2 TrustManager

Bypassing TrustManager SSL pinning is straightforward. The goal is to create custom TrustManager with user certificate and anytime SSLContext is invoked, the supplied TrustManager is replaced by the custom TrustManager. The Frida script is visible in listing 34 (appendix A). The process is the same as trying to implement the TrustManager in Java on Android. The same steps need to be done. The only difference is hooking the SSLContext.init method and swapping the TrustManager parameter with the (previously created) custom TrustManager (that trusts the user certificate). This way, the SSL pinning is bypassed as the application will trust the user-supplied certificate. The script assumes a user certificate that should be trusted with .crt extension in /data/local/tmp directory.

4.2 Root Detection

This chapter focuses on bypassing application-level RASP. SafetyNet assesses the entire device and is validated server-side. Due to these reasons, SafetyNet will not be in the scope of this section. In theory, SafetyNet can be bypassed by hooking the specific function inside the application that processes the output of the attestation. This process can not be generalized as the name of the function is custom and unknown beforehand.

4.2.1 Programmatic Detection

Root detection can be implemented in many possible ways. This thesis focuses on the most common cases. The first case is using the PackageManager.getPackageInfo method to check for the existence of a specific package. Root applications have known package names and the presence of such package on the device is a sign of a rooted device. To bypass this, getPackageInfo method can be hooked, and if the supplied parameter (name of the package that the application wants to check) is in the list of known root applications, the parameter gets changed to non-existent value. The hook is visible in listing 14.

```
1 PackageManager.getPackageInfo.implementation = function (pname, flags) {
2     if (RootPackages.indexOf(pname) > -1) {
3         pname = "random.bypass.package";
4     }
5     return this.getPackageInfo.call(this, pname, flags);
6 };
```

Listing 14: Bypassing getPackageInfo root check.

Applications that try to execute commands like "su" or other root package can be misled into thinking that the command is not available by hooking the Runtime.exec method. It takes one parameter (but has multiple overloads) that is the command to be executed. This parameter is checked if it contains any of the defined root packaged like "su" and if so, gets changed into random non-existent command. This way, the application will receive command not found error and will believe that the root package is not available. The script is visible in listing 15. Note that most of the methods mentioned here have multiple overloads. The hooking process is the same for all of those. The only change is the processing of the input parameters and checking if any of them needs to be changed. The array "RootBinaries" contains

the list of known root binaries. Using an array allows the script to be modular and quickly changed to reflect new packages or some specific package.

```
Runtime.exec.overload('java.lang.String').implementation = function (cmd) {
    if (RootBinaries.indexOf(cmd) != -1) {
        var bypass = "randomcommand";
        send("Bypass " + cmd + " command");
        return exec1.call(this, bypass);
    }
    return exec1.call(this, cmd);
};
```

Listing 15: Bypassing Runtime.exec root check.

It is possible to modify specific system properties on a rooted device like disabling SELinux. These changes can be represented in the build.prop file and applications can check for these properties using SystemProperties.get method. The bypass works by checking the parameter of the method and if it is a parameter that was modified by rooted device, the method returns the value as if the device is not rooted (listing 16).

```
SystemProperties.get.overload('java.lang.String').implementation = function
↪  (name) {
    if (name in RootProperties) {
        send("Bypass " + name);
        return RootProperties[name];
    }
    return this.get.call(this, name);
};
```

Listing 16: Bypassing SystemProperties.get root check.

4.3 Emulator Detection

Emulator detection relies on the values inside build.prop file and TelephonyManager API. Both can easily be hooked to change their return values. For example, the value of Build.BRAND can be set to "google" to pretend to be Google Pixel device. It is possible to find valid build.prop values for devices like Google Pixel 3. These values were used in the script to make it look like the emulator is a Google Pixel

3 device. TelephonyManager APIs were also hooked to return random values that are not known to be an emulator. The sample script is in listing 17. Note that only one of both Build and TelephonyManager calls are shown, others are implemented similarly.

```
var Build = Java.use('android.os.Build');
var TelephonyManager = Java.use('android.telephony.TelephonyManager');

Build.BRAND.value = "google";

Build.getRadioVersion.implementation = function () {
    return "123456";
};

TelephonyManager.getDeviceId.implementation = function () {
    return "123456789";
};
```

Listing 17: Emulator detection bypass.

4.4 Debugger Detection

Native debugger detection is out of the scope of this thesis as it is application-specific and depends on the exact implementation. Frida also uses ptrace to hook into the application. If the debugger detection is done correctly, it can also block Frida (or detect it as a debugger). This situation would require to hook the specific methods and disable them (Frida can be used to bypass Frida detection). A similar case is the timer check method (checks if the runtime of the application is not slowed down due to debugger attached). This situation would be implemented using a custom library or function and can not be bypassed by a general script (but the principle remains the same, find the method, hook it, disable/modify it to prevent the functionality).

The two mentioned examples are both related to JDWP debugger detection in standard Java. The first method uses getApplicationInfo().flags call to obtain a list of flags specified in the application. One of the flags is android:debuggable, which allows the debugging of the application via JDWP. To hide the presence of this flag, the hook intercepts the call, retrieves the value of the flags and subtracts the value of the FLAG_DEBUGGABLE (specified as "1«1" – 2 in decimal). Modified object with changed "flags" value is returned.

The second method uses Debug.isDebuggerConnected to check for the presence of the debugger. This call can easily be intercepted, hooked and modified to always return false.

Both methods are visible in listing 18. Note that the getApplicationInfo is called on "ContextWrapper". The call requires "Context", which is an abstract class and the "ContextWrapper" is the actual implementation.

```
var ContextWrapper = Java.use("android.content.ContextWrapper");
var Debug = Java.use("android.os.Debug");
var FLAG_DEBUGGABLE = 1 << 1;

ContextWrapper.getApplicationInfo.implementation = function () {
    var appInfo = this.getApplicationInfo.call(this);
    appInfo.flags.value &= ~FLAG_DEBUGGABLE;
    return appInfo;
};

Debug.isDebuggerConnected.implementation = function () {
    return false;
}
```

Listing 18: Debugger detection bypass.

5 Advanced Obfuscation Techniques

For the purpose of this thesis, advanced obfuscation techniques will include primarily string encryption. As dynamic analysis is partially dependant on static analysis (you need to analyse decompiled source code to know, which methods to analyse dynamically), several other obfuscation techniques will need to be considered. These techniques include flow control obfuscation, Java reflection, and DEX encryption.

To test these methods, three new obfuscators will be introduced. Allatori[1], DexProtector[2] and DashO[3]. These three companies agreed to provide their respective products for free for this research. DexGuard could not be used due to licensing issues.

Allatori is a paid Java obfuscator. In comparison with ProGuard, Allatori provides additional security features like flow obfuscation or string encryption [15]. These features add additional complexity to the decompiled application, which makes it harder to analyse.

DexProtector is a specific obfuscator for mobile applications (not general for Java applications) supporting both Android and iOS platforms. Unlike other obfuscators, DexProtector supports encryption of classes [16]. In this case, the original source code is encrypted and replaced by custom decryption code. Static analysis of the apk file is not possible as the only visible element is the custom decryption code and not the actual source code of the application. Other features include string encryption, method call hiding, integrity control, tamper notification or native code obfuscation/encryption.

DashO is an obfuscator for Java and Android (similar solutions are available from the same company for iOS, .NET or JavaScript). Features include control flow obfuscation, string encryption, tamper detection, watermarking and others [17].

Versions used can be seen in table 5.1. Additionally, Burp Suite v2020.2 was used together with Brida extension v0.3.

Tab. 5.1: Versions of obfuscators.

	Allatori	**DexProtector**	**DashO**
Version	7.2	10.1.19	10.3.0 (Trial)

[1] http://www.allatori.com/
[2] https://dexprotector.com/
[3] https://www.preemptive.com/products/dasho/overview

5.1 Test Application

To thoroughly test advanced obfuscation techniques, a special application had to be programmed. This is because access to both obfuscated and original source code is ideal for research.

5.1.1 Design and Intentions

Design of the application took into consideration two main points:

- Hardcoded strings are usually very good entry-points for a penetration test because searching for text strings like "Payment successful" or "Activation successful" can quickly identify sensitive methods that need proper testing. If the strings are encrypted, it will be harder to find sensitive methods by static analysis.
- Sensitive applications tend to have custom methods, which are necessary during the penetration test. For example, digital signing of requests sent to the webserver. If the tester can not tamper with requests, the application will not be adequately tested. These custom methods that are obfuscated with control flow obfuscation, string encryption or method call hiding are tough to recreate and understand using static analysis.

The resulting application is mimicking mobile banking application with reduced functionality. User can log into the application and execute the money transfer. The application is communicating with a backend server, and all requests are digitally signed. Hardcoded strings are also included, some of which are necessary from the programming perspective and some are specifically implemented to demonstrate the effect of embedding sensitive information directly into the source code.

The backend server is also simplified and built with Python 3. Every request is first verified if the signature is valid. If the signature is not valid, the server will not execute the request.

The application was designed and implemented prior to any work with any obfuscator and is not specially suited to work with any obfuscator. In other words, the application is not specifically designed to be easy to decompile or to provide an easy workflow for dynamic analysis.

5.1.2 Implementation

The application is built using Android Studio and Android 10. The main activity is the login activity, which allows the user to log into the application. The login function is simplified and the user only specifies the name, no password is used (it is not needed for demonstrative purpose). All data is stored server-side; the

application does not store any data. Data is kept in the SQLite database that the backend server has access to. The database contains only one table ("*accounts*") with name, account number and the amount of money for each individual user (currently only two users are created). HTTP login request can be seen in listing 19.

```
POST /accounts HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 191
Host: 10.0.2.2:8090
Connection: close
Accept-Encoding: gzip, deflate
User-Agent: okhttp/4.4.0

test=Pavol&name=Pavol&timestamp=1590865095110&
signature=d0040a5d093c69cc5d9f4392f90ba7f64b687c07f2d20d096e277d57e75fbd89&
nonce=89db728b6d8007617f97fd07bb8f9fa47fc91ffecf125d78701aba84cf87e190&
```

Listing 19: HTTP login request.

After the user logs into the application, he/she can execute the money transfer. Only the name of the recipient and the amount of money is required.

All requests are digitally signed. The signing algorithm is simple. HTTP Post parameters are in the form of "key=value" separated with "&" symbol. The data is specific to each request. For example, for login, only the name is required (name="Pavol"). The data is in the form of a HashMap, which is making it simple as it also requires key and value (just like HTTP Post). The input into the signature generation function is a HashMap object, which contains all the required parameters for the given request. First, HashMap is appended with a nonce (SHA-256 hash of random bytes) and the current timestamp. An "initialization vector" is used. It is a hardcoded string with a value of "Lorem ipsum dolor sit amet". Signing data is generated by appending key and value of each parameter inside the HashMap together. The first set of key-value is prefixed with the specified initialization vector (IV). An example of the signing data could be: "Lorem ipsum dolor sit ametnamePavolnoncea0b643a491479e22a01b995d38cc4a08db11f0ad52dbdff41a3-6ad6686e955d6timestamp1585854384645". The signature of this data is created using "HMAC-SHA256" algorithm, which is symmetric and does not require private and public key. The key is generated by appending the IV, timestamp and nonce (this way, the key is always different). The signature is appended into the HashMap and this new HashMap is returned and later on parsed into HTTP Post request. The IV is used for a simple reason. It is a string that is not known to

the user, only to the application and the server (if the string is encrypted during obfuscation). Developers may incorrectly believe that since the string is encrypted and obfuscated, user can not obtain this secret and thus produce a valid signature. Also, it acts as a form of "shared secret" between the application and the server. This application is not meant to be secure. The entire signature generation method can be seen in Appendix B.

The backend server is a basic Python 3 server built with Flask framework[4]. Each received request is first verified by checking the signature validity. The signature is calculated exactly the same as in the application. The server calculates its own signature and checks it against the received one. If they are equal, the request is valid and will be executed by the server.

5.2 Testing Methodology

The idea behind this methodology is to save as much time as possible (which also means lower cost of the test itself) and also to allow seamless integration with existing tools, so the penetration tester does not have to work with many different tools. The main goal is that the application already contains all the code and methods that are required for the test. These methods include encryption/decryption methods for strings or signing algorithm for requests. Dynamic analysis (and Frida) allows the usage of these methods on-demand in the context of the application.

All tests were executed on Android 10 emulator based on Pixel 3 device.

5.2.1 Brida

Brida is an extension (plugin) to Burp Suite proxy. It provides the user with a graphical interface for Frida as well as JavaScript editor for injected code. The main benefit is that the user does not have to create a separate tool to run Frida scripts. Brida allows integration with Burp Suite interface and calling Frida functions from the context menu in intercept or repeater tabs. For example, assume that the application is communicating with the webserver and is sending encrypted data. With static analysis, code responsible for encryption can be found and reimplemented in some custom tool. In this case, requests and responses inside the Burp Suite would have to be copied back and forth between the custom tool and Burp Suite. However, there is no need to reimplement the encryption algorithm, which may be difficult or complicated. The application itself contains the code for encryption, Frida allows the usage of this method directly without reimplementing

[4]https://palletsprojects.com/p/flask/

anything. With Brida, the encryption method can be exported and made available to use in Burp Suite. All that is required is to select the encrypted request and execute the hooked decryption function from the context menu (right-click). Brida will output the decrypted request and can also directly replace the encrypted request with decrypted (which is helpful the other way around – plain-text request is provided/tampered with and Brida outputs encrypted version).

The proposed methodology architecture can be seen in figure 5.1. Burp Suite has two primary purposes. To intercept HTTP(S) traffic for potential tampering and also acts as a graphical interface for Frida (note that Brida requires external Pyro4 server for communication between Brida and Frida).

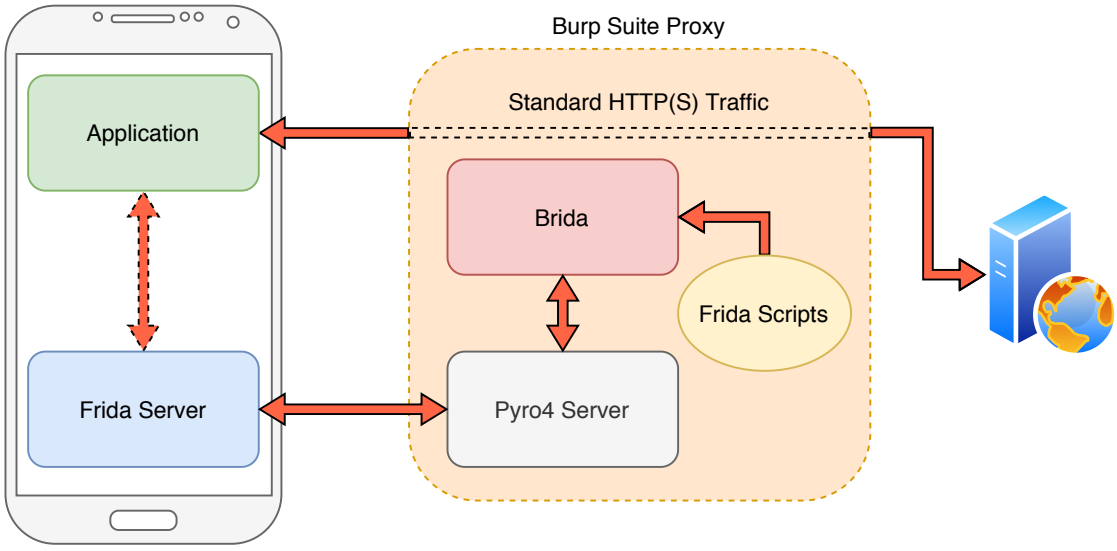

Fig. 5.1: Testing Methodology Architecture.

This architecture is better than using custom Python tool to work with Frida. Penetration testers will use Burp Suite either way and Brida allows much faster and smoother workflow than any other custom tool. There are some negatives to this workflow. Some Frida crashes (due to incorrect injected code or similar) will not be communicated to the tester (no error is displayed). Custom Python tool would many times communicate this error to the tester and he/she can fix it. If the application crashes multiple times, using the same Frida code with the custom tool is recommended to check for potential Frida error.

Brida features six different tabs:

- Configuration – basic configuration includes path to Python binary, path to injected JavaScript file or name of the application to be analysed.
- JS Editor – built-in editor for JavaScript.
- Analyze binary – a list of all methods and their signatures.
- Generate stubs – generates stubs for extensions calling Brida functions.

- Execute method – executes custom methods (inside the JavaScript file).
- Trap methods – shows the hooked methods.

The JavaScript file used with Brida contains specific methods for Brida to work properly and also custom hooks. There are four custom context methods available (contextcustom1, contextcustom2, ...). These methods are accessible by selecting text inside the Burp Suite and right-clicking. The first two methods (numbers 1 and 2) are available in any read/write editor, for example, the Intercept tab and the Repeater tab. The last two methods (numbers 3 and 4) are available in any read-only editor (like proxy history tab).

5.2.2 Test Objectives

Test objectives can be separated into two categories, primary and secondary. Primary objectives are to bypass string encryption and be able to visually see the decrypted versions of strings, and to be able to produce a valid request signature with the tampered data.

Secondary objectives will focus on different parts of obfuscation. At the beginning of each test, the static analysis will be conducted to determine how the obfuscator changed the decompiled source code. Variable renaming, flow control obfuscation, dex file encryption and others will be part of the secondary objectives. It is neither the goal nor intent of this thesis to provide a ranking or rank each individual obfuscator (no ranking will be provided). The goal is to provide testers with examples of different obfuscators and how they obfuscate the code, so they know what to expect and how to work with them.

5.3 Disclaimer

DashO and DexProtector contain functionality that is designed to prevent dynamic analysis from working. This functionality is out-of-scope of this work and was not used. These features are extremely complicated to bypass and would require full thesis only for this functionality. However, only high-end products (and the highest/most expensive versions) contain this functionality and as such, there are many cases, when this feature is not used. Also, many developers are unaware of the security risks and do not use advanced functions.

6 Bypassing Advanced Obfuscation

The results are presented for each individual obfuscator. Each obfuscator has multiple categories. Configuration for each obfuscator is presented as well as results of static analysis, string encryption and signature generation method hooks. All obfuscators were tested with the release build of the application.

6.1 Allatori

6.1.1 Configuration

The configuration used is a slight modification to the default Android configuration, which is shipped with the Allatori. String encryption is explicitly enabled and set to "fast" mode. This is the default variant. An attempt was made to use stronger encryption with multiple options such as "strong" or encryption with "maximum-with-warnings" mode. These modes did not work as the strings were decrypted incorrectly and the application was not functional. The default values were used during testing for this reason. Also, additional control flow obfuscation was used (option "maximum" instead of the default "normal"). The entire configuration is visible in Appendix C (highlighted lines refer to modifications mentioned in this section).

6.1.2 Static Analysis

The original test application contains 5 classes (2 activities and 3 supporting classes). Allatori (and specifically control flow obfuscation) split many functions into multiple parts and into separate classes. The total amount of classes after obfuscation was 10, which is exactly twice as the source code. The source code contains multiple anonymous inner classes (for example as button onClick listener or callback for HTTP request). These inner classes are split into separate classes.

To demonstrate the obfuscation quality of Allatori, consider an excerpt of signature generation function in listing 20 (the entire original method is in Appendix B). Only the first part of the function is shown (the function is too long). The displayed part in the listing is almost the same as the original code, only variable names are obfuscated and strings are encrypted.

Allatori (for the most part) is comparable to the standard ProGuard (R8) with the addition of some advanced techniques like string encryption or control flow obfuscation.

```
1  public static Map<String, String> m(Map<String, String> map) {
2      String m = m();
3      String valueOf = String.valueOf(System.currentTimeMillis());
4      map.put(f.m("\u0019\u0018\u0019\u0014\u0012"), m);
5      map.put(m("s/j#t2f+w"), valueOf);
6      String m2 = f.m(";\u0018\/*...*/");
7      boolean z = true;
8      String str = "";
9  . . .
```

Listing 20: Allatori – signature generation method.

6.1.3 String Encryption

Encrypted strings can be seen in listing 20 (lines 4 to 6). The principle of string encryption is simple. Every hardcoded string gets replaced with a call to the decryption method, which takes the encrypted string as the parameter and returns the decrypted value. The decryption function is visible in listing 21. There is no variation in the decryption methods, and all are the same. The only change is in certain values (for example the exponent – lines 8 and 13 in listing 21).

```
1  public static String m(String str) {
2      String str2 = str;
3      int length = str2.length();
4      char[] cArr = new char[length];
5      int i = length - 1;
6      while (i >= 0) {
7          int i2 = i - 1;
8          cArr[i] = (char) (str2.charAt(i) ^ 'w');
9          if (i2 < 0) {
10             break;
11         }
12         i = i2 - 1;
13         cArr[i2] = (char) (str2.charAt(i2) ^ 'w');
14     }
15     return new String(cArr);
16 }
```

Listing 21: Allatori string decryption function.

6.1.4 Signature Generation

The first step is to find a method that is generating the signature. Due to obfuscation, this may be hard as the methods are split into multiple parts and variables have meaningless names. A good way to begin is to decrypt all encrypted hardcoded strings. Using Brida, the hooking process does not have to be manual. Brida features "Inspect" function (inside "Analyze Binary" tab – right-click on the method name and select "Inspect!"). This function hooks the specified method and prints out the input parameters, runs the original method and prints out the output. Using the decrypted strings, a function that was appending values into HashMap object with keys "nonce", "timestamp" and "*signature*" was found. In a large application, this could have been more difficult. However, Brida brings much automation into the process as you do not have to write each individual hook manually but just right-click the function in the Brida. For example, you can also hook all functions inside a single class just by right-clicking the class. The name of the decryption function was *o.c* and the output after inspecting this function is visible in listing 22 (note that the input parameters are Unicode symbols and as such are shown in either escaped variant or as a blank character).

```
*** entered cz.vutbr.pavol.obfuscation.experimental.main.o.c
Parameters:
arg[0]: $\u0014=\u0018#\t1\u0010
*** exiting cz.vutbr.pavol.obfuscation.experimental.main.o.c
Return value:
    retval: timestamp

*** entered cz.vutbr.pavol.obfuscation.experimental.main.o.c
Parameters:
    arg[0]: #\u00147\u00131\t%\u000f5
*** exiting cz.vutbr.pavol.obfuscation.experimental.main.o.c
Return value:
    retval: signature
```

Listing 22: Allatori string decryption method inspection output.

After the signing function was discovered, it can be hooked and utilized for request modification. One of the Brida context methods can be used (for example number 1 as it can be used in the writable editor). The signature generation function (listing 20) expects HashMap as input and generates nonce, timestamp and signature for the provided values. So the first step is to create HashMap with all the parameters necessary. The output of the signing function is also a HashMap

(the same HashMap that was the input for the function, just appended with additional values). This HashMap is than processed by OkHttp library and parsed into the POST request body. The custom function created this way has three steps: take the current request parameters, parse it and create HashMap for the signing function; call the signing function with the custom HashMap; parse the output into the POST request and replace the selected request parameters with the signed parameters. The knowledge of the custom signing algorithm is not necessary. By hooking the signature generating function, a valid signature can be produced for any request and any parameter can now be tampered with, allowing full testing of the application.

The entire method is visible in listing 23. The first line specifies the name of the method (contextcustom1 – name is required to be in this format by Brida) and an input parameter *message* (in this case the message will be the request with a tampered parameter that the signature is needed for). The input parameter is always passed as a hexadecimal string. This is automatically done by Brida so that the Brida can work with binary data. On line number 5, the message is transformed from hexadecimal format to ASCII string called *request*. The request variable contains typical POST request body string (for example "name=Pavol&account_num=123"). So the request contains multiple "key=value" pairs separated by "&" symbol. The next step is to split this long string into individual entries of the new HashMap (line 6). Lines 7 and 8 create a new HaspMap object.

The next step is to populate the new HashMap with the correct values. The HashMap is basically a set of key-value pairs. In this case, the key is the name of the HTTP parameter and value is the value of the parameter. For example, given the HTTP request body above ("name=Pavol&account_num=1234") the HashMap that is required can be written as follows: "name":"pavol"; "account_num":"1234" (notation is "key:value" and individual pairs are separated by semicolon). This is exactly the purpose of the for loop (lines 10-17). The request is parsed into the HashMap, note that certain keys are skipped (line 12). These keys include "nonce", "timestamp", and "signature". These values are present in the request but after tampering the request, they will get generated by calling the signature function. That is the reason these values are not appended to the HashMap.

After the HashMap is prepared and populated with the correct values, the signature generation method can be called (lines 18-21). The output of the function is a HashMap appended with additional values such as "nonce", "timestamp" and "signature". The last part of this process is to parse this new HashMap into valid HTTP POST body (lines 22-30). This process is the same as the process to create the HashMap from the request; only it is inverted. The last step is to return the HTTP POST body string. The string must be in hexadecimal format (line 32).

```
contextcustom1: function (message) {
    return new Promise(function (resolve, reject) {
        if (Java.available) {
            Java.perform(function () {
                var request = hexToString(message);
                var fields = request.split("&");
                const HashMap = Java.use("java.util.HashMap");
                const map = HashMap.$new();
                var index;
                for (index = 0; index < fields.length; ++index) {
                    var parameter = fields[index].split("=");
                    if (["nonce", "timestamp",
                    ↪  "signature"].indexOf(parameter[0]) >= 0) {
                        continue;
                    }
                    map.put(parameter[0], parameter[1]);
                    console.log("adding to map key: " + parameter[0] + " value:
                    ↪  " + parameter[1]);
                }
                const SignClass =
                ↪  Java.use("cz.vutbr.pavol.obfuscation.experimental.main.o");
                console.log("* after class");
                var method = SignClass.c.overload("java.util.Map");
                var signedMap = method.call(SignClass, map);
                var HashMapNode = Java.use("java.util.HashMap$Node");
                var iterator = signedMap.entrySet().iterator();
                var body = "";
                while (iterator.hasNext()) {
                    var entry = Java.cast(iterator.next(), HashMapNode);
                    console.log(entry.getKey() + "=" + entry.getValue());
                    body += entry.getKey() + "=" + entry.getValue() + "&";
                }
                body = body.substring(0, body.length - 1);
                try {
                    resolve(stringToHex(body));
                } catch (e) {
                    reject(e);
                }
            });
        }
    });
},
```

Listing 23: Allatori – Brida signature generation function.

Note that the entire functionality (lines 5-35) is wrapped inside *Java.available()* and *Java.perform()* calls. Both are needed for the proper functioning of Frida. Also, note that the function does not return the value using the standard keyword *return*. Instead, the function returns *Promise* object (line 2) by using the keyword *resolve* (line 32). Java.perform is potentially asynchronous and during testing, the standard *return* did not work. The function was functioning properly, but the return value was always null. To bypass this issue, the Promise with resolve workflow was used (as recommended by the author of Frida[1]).

To use this contextcustom1 method, a valid request can be intercepted inside the Intercept tab or the Repeater tab. Any parameter can be modified and by selecting the entire POST body, right-clicking and selecting "Brida Custom 1" the custom method is called and the output of the custom function replaces the selected text. So in this case, the request with modified values (for which the signature is invalid) gets replaced with a new string with the modified values and nonce, timestamp and signature generated for these modified values. This request (even with modified values) is treated as valid by the server as the signature is correct.

6.2 DashO

6.2.1 Configuration

The default configuration was used. The only exception is the string encryption level setting (default was 2). This setting got increased to the maximum available value (10). The number of decryption functions was also increased to 10. Control flow obfuscation was turned on. Due to short license time (7 days), it was not possible to evaluate all the options provided by the DashO. The entire configuration is visible in listing 24.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <dasho mode="android" version="10.2.0">
3     <controlflow option="on"/>
4     <stringencrypt level="10" implementations="10" option="on"/>
5 </dasho>
```

Listing 24: DashO configuration.

[1] https://github.com/frida/frida/issues/380

6.2.2 Static Analysis

The APK contains several dozens, up to hundreds of classes. Variables, class and method names have meaningless names. Methods are much larger than their original counterparts and contain additional code (junk code) to confuse the reverse engineer. For example, the signature generation methods, which contains ≈30 lines of code is obfuscated into ≈560 lines of code. Several methods even failed to decompile correctly (using Jadx). DashO is delegating the variable/class renaming to the default R8 obfuscator.

The control flow is obfuscated really well. Several methods return different return values. For example, the signature generation function normally returns HashMap object. However, after obfuscation, the method returns a custom object, which includes two List objects and multiple methods (one List for the keys of the HashMap and the second for the values). This adds another layer of security and confusion. DashO also adds many if-else statements into methods. A lot of the junk code is actually if-else statements. An example of these if-else statements is visible in listing 25 (the code is just an excerpt, the full code contains multiple if-else statements in a row). During the manual reading of the decompiled code, it is very hard (or nearly impossible) to keep track of all the values used and approximate, which code execute and which does not.

```
1  if (Integer.parseInt(str4) != 0) {
2      i = 8;
3      str = str4;
4      i3 = 1;
5      i2 = 1;
6  } else {
7      i3 = 24;
8      i2 = 24;
9      i = 14;
10     str = str5;
11 }
12 if (i != 0) {
13     i2 = i2 + i3 + i3;
14     str = str4;
15     i4 = 0;
16 } else {
17     i4 = i + 7;
18 }
```

Listing 25: DashO control flow obfuscation using if-else statements.

6.2.3 String Encryption

As specified in the configuration section, the maximum number of decryptors was used (10) as well as the highest level of encryption (10). All the decryption methods are similar. They always return a String object (the decrypted string) and takes a different amount of parameters. All of the methods take a String as their input. This is the encrypted string that is supposed to be decrypted. The other parameters are one to two integers. Some of the decryption method signatures can be seen in listing 26. The decryption methods are also obfuscated and hard to read. One of the decryption methods can be seen in Appendix D.

The hooking process is very similar to Allatori (section 6.1.3). It is easy to identify the string decryption methods (find all the encrypted strings and they will always be used as the input for these methods). These decryption methods can be used using Brida Inspect feature and all strings are decrypted.

```
1 public static String a(int i, int i2, String str);
2 public static String a(int i, String str);
3 public static String a(String str, int i);
4 public static String a(String str, int i, int i2);
```

Listing 26: DashO string encryption methods signatures.

6.2.4 Signature Generation

The signature generation method can be located the same way as specified in Allatori (section 6.1.4). Hooking of all string decryptors can be easily hooked using Brida and the keyword "signature" will identify the correct method. The method is too long to be displayed (over 500 lines). Manual reading of the method is almost impossible as many if-else statements are included and multiple calls to different methods are made. The benefit of using dynamic analysis is that it is not necessary to actually understand the method fully and still, it is possible to leverage it to create a valid signature for any data.

The excerpt of the method can be seen in listing 27. The first interesting part is the method signature. The return value is not a HashMap object but an object of class "b0" (the fully qualified name is "e.b0"), which is an abstract class. This class is extended by the "r" class. The object of the class "r" is returned on line 5. The two input parameters are both List objects. So the control flow obfuscation changed not only the look of the method but also the return type is different. In practice, this means that for a reverse engineer, it is going to be very hard to follow the "flow" of

the program. HashMap is a standard Java type and the reverse engineer knows how it works and what to expect from it (like what functions it uses or what purpose it serves). However, if the code is working with custom classes that are also inheriting or extending other classes, the reverse engineer has only one option, to manually check all of these classes and figure out what they do. In a large project (or even in small one such as the test application), this is very time-consuming. Dynamic analysis is not hindered by this obfuscation.

```
1 public static b0 a(Map<String, String> map) {
2     ...
3
4     if (aVar != null) {
5         return new r(aVar.a, aVar.b);
6     }
7     throw null;
8 }
```

Listing 27: DashO signature generation method excerpt.

As the "new" keyword is used to instantiate the new object, the class constructor is called. This constructor can be seen in listing 28. Notice that the class contains two global variables (List objects) "b" and "c" (lines 2 and 3). The class constructor (line 5) takes two List objects as input, processes them by calling "d.b" method and this value is set to the corresponding global List variable "b" or "c".

After seeing this workflow (and with the knowledge of how the original function is supposed to work), it was assumed that the first List would contain the "key" values of the original HashMap and the second List would contain the "value" part of the HashMap.

To hook this method and to produce a valid signature, a similar workflow is used as with Allatori (section 6.1.4). The first part of the hook method is the same; HTTP POST parameters are parsed into the HashMap object. The middle part is different as it is needed to call a different method and also process the return value differently. This section can be seen in listing 29 (only this part is shown, the first and the last part of the hook is the same as in listing 23).

The first two lines set the variables to the classes that are both necessary. The "c.a.a.a.a.a.c" class is the one that contains the signature generation method (called "a") and the "e.r" class is the return type. Next, the original method is called (notice that it is called with the specified overload as multiple methods can have the same name). The "map" input parameter is the HashMap generated in the first step (not shown in this excerpt). The output of the signature generation method

```
1  public final class r extends b0 {
2      public final List<String> b;
3      public final List<String> c;
4      public r(List<String> list, List<String> list2) {
5          h.b(list, a.a.k.a.a(3, "ntjw#3!\u001ab\u007fdc"));
6          String a2 = a.a.k.a.a(2, "owkx\"0 \u0005c}ujm");
7          if (list2 != null) {
8              this.b = d.b(list);
9              this.c = d.b(list2);
10             return;
11         }
12         h.a(a2);
13         throw null;
14     }
15 }
```

Listing 28: DashO "r" class constructor.

is stored in the "tmp" variable. To work with this object, it is needed to cast it into its proper class (which is "e.r"). After the cast, all the variables and methods of the object are accessible (without the cast, these would not be accessible). The for loop (lines 6-8) parses the key and values from the two lists into the required format ("key=value&key=value&..."). As both lists are global variables, they can be accessed as in standard Java (in this case, output.c or output._b). Notice that the List "b" is accesses with an underscore (output._b). This is because the "e.r" class contains a variable named "b" (the global List) and also a method called "b". To differentiate the two, Frida uses underscore to access the variable. If the underscore would not be used, Frida would instead call the method "b". Another thing to note is that to access the value of the List a ".value" is required before List-specific methods like get() or add() can be used (line 7).

An interesting finding is that even such a significant change to the workflow of the method (such as changing the return type) has only small impact on the changes required for the dynamic instrumentation hook/function to work even given the changes to the code.

```
var SignClass = Java.use("c.a.a.a.a.a.c");
var er = Java.use("e.r");
var tmp = SignClass.a.overload("java.util.Map").call(SignClass, map);
var output = Java.cast(tmp, er);
var str = "";
for (var index = 0; index < output._b.value.size(); ++index) {
    str += output._b.value.get(index) + "=" + output.c.value.get(index) + "&";
}
```

Listing 29: DashO signature generation method hook excerpt.

6.3 DexProtector

6.3.1 Configuration

Default configuration was used with the addition of resource encryption (should have no effect on the output). Class encryption and string encryption was enabled by default. The entire configuration can be seen in listing 24 (Appendix E).

6.3.2 Static Analysis

Static analysis is a bit different with DexProtector compared to Allatori or DashO. DexProtector has class encryption feature. This means that the original source code (dex files) are encrypted and replaced by custom code. During runtime, DexProtector first decrypts native library, loads this library into memory (and instantly deletes the library from the filesystem), (likely) decrypts the encrypted dex files. This is a general workflow. The output apk file is basically a packer[18]. When the apk file is statically analysed, only the custom code is visible, not the original. The tester will not learn any relevant information. The custom code is fully obfuscated. Decryption is (likely) done by the native library. During the testing, it was discovered that only relevant parts of the source code are loaded into memory (this may mean that only parts of the code are decrypted, or the entire code is decrypted and dynamically loaded using Java class loader).

Numerous techniques can be used to obtain the original dex files[18, 19, 20]. Some of these techniques rely on custom Android kernel, and others use memory dumps. During the testing, the memory dumping technique was used with no results. This protection is out-of-scope of this thesis. For the purpose of this thesis, it is beneficial that there is no access to the decompiled source code. This situation will allow for usage of a different approach to the signature generation.

6.3.3 Signature Generation

To hook the signature generation method, first, its name is required. However, without access to the decompiled code, this can be a challenge. This "blind" approach requires two things. Knowledge of the application and stack traces. The knowledge (or assumptions) is used to limit the number of methods (that may contain the sought functionality) from all to a smaller set. For example, the test application visibly uses nonces and timestamps (both are visible in the HTTP requests). It can be assumed that during signature generation (either in the same method or as a method call), these values are generated. Timestamps can be generated with known standard Android API, same for nonces (by visual observation, it can be assumed that some form of 256 bit hash function is used). This knowledge allows the hooking of these standard APIs and observe methods, where they are called. The stack traces come into play here. The Stack trace is basically a log of methods called in succession (it shows the current method name and the execution flow prior to calling this method). Combination of both the knowledge/assumptions and the stack traces can pinpoint the specific method that is generating the signature in this case.

To better work with Brida "Inspect with Backtrace" feature (prints out the stack trace), it is a good idea to modify the Brida function used to log the stack trace. In some instances (like multiple methods inspected at once), the output can be mixed and confusing (console.log is not synchronous). Also, multiple methods can have the same name, but different signature (return value or parameters). The proposed change is to always print the method signature in both the method name and also the return value (so the return value can be linked to specific function if the console log is mixed). The proposed change can be seen in listing 30.

As stated above, the first step to locate the signature generation method is to hook any known method that is assumed to be used in conjunction with it or directly inside of it. One of such methods can be *System.currentTimeMillis()*, which produces timestamp. Using Brida, this function can be inspected with backtrace (in Brida, the stack trace is called "backtrace", for the purpose of this thesis, both terms are equivalent). Any time the function is called a stack trace is printed to the console. Also, the return value is logged.

It can be assumed that the signature generation method is called right before the HTTP request is generated (the signature is not generated beforehand as the values are inputted by the user). In this case, the signature is likely generated after executing an action that results in an HTTP request. This action is also clicking a button (for example, to login or execute money transfer). During the login action (after clicking the button), Brida log is observed for the traced currentTimeMillis()

```
@@ -446,7 +446,7 @@ function traceJavaMethod(pattern,backtrace) {
         // print args
         if (arguments.length) console.log("Parameters:");
         for (var j = 0; j < arguments.length; j++) {
-                console.log("\targ[" + j + "]: " + arguments[j]);
+                console.log("\targ[" + j + "]: " + arguments[j] + " Signature:
↪  " + pattern);
         }
         // print backtrace
         if(backtrace === "true") {
@@ -464,7 +464,7 @@ function traceJavaMethod(pattern,backtrace) {
         var retval = this[targetMethod].apply(this, arguments);
         console.log("*** exiting " + targetClassMethod);
         console.log("Return value:");
-        console.log("\tretval: " + retval);
+        console.log("\tretval(" + pattern + "): " + retval);
         return retval;
         }
 }
```

Listing 30: Proposed change to Brida logging.

method. Multiple calls to this method are observed. To pinpoint the one that is being used during signature generation will be the one, where the return value is the same as the value in the HTTP request that is sent. The stack trace can be seen in listing 31.

On line 5, the call to System.currentTimeMillis() is made. The stack trace is written in a way that at the top are the most recent methods and at the bottom the methods that called the methods above them (so the stack trace should be read from bottom to top). This stack trace is generated as expected. The button click generated an *onClick* event (line 9). Inside the onClick method the function *a.a.k.p.a* is called (line 7) and later on, this method calls the currentTimeMillis() method (line 5). The return value (line 14) matches the timestamp value in the HTTP request. This stack traces indicate that the a.a.k.p.a function is generating the signature. One of the ways to confirm this is to use knowledge of some other method that is likely used during the signature generation and check if both are being called by this function. The other way is to directly inspect with backtrace the method a.a.k.p.a (class "a.a.k.p" and method name "a"). The downside of stack traces is that only method names are printed; however, in Java, multiple methods can have the same name (the only difference is either return value or input parameters). This is also the case in this application. Class "a.a.k.p" contains dozens of methods with the

```
1  *** entered java.lang.System.currentTimeMillis
2  Backtrace:
3          dalvik.system.VMStack.getThreadStackTrace(Native Method)
4          java.lang.Thread.getStackTrace(Thread.java:1720)
5          java.lang.System.currentTimeMillis(Native Method)
6          cz.vutbr.pavol.obfuscation.experimental.main.LibApplication.i(Native
           ↪  Method)
7          a.a.k.p.a(:95)
8          cz.vutbr.pavol.obfuscation.experimental.main.LibApplication.i(Native
           ↪  Method)
9          cz.vutbr.pavol.obfuscation.experimental.main.MainActivity$a.onClick(
           ↪  Unknown Source:66)
10         android.view.View.performClick(View.java:7125)
11         ...
12 *** exiting java.lang.System.currentTimeMillis
13 Return value:
14         retval(long java.lang.System.currentTimeMillis()): 1586954102067
```

Listing 31: DexProtector – currentTimeMillis() stack trace.

name "a" and there is no way to tell, which one is the one in the stack trace (note that all classes and methods need to be loaded with the Java class loader and as such can be printed during runtime, this is a valid way to get a list of all classes and methods inside them, Brida "Analyze binary" feature can be used to list the classes and methods in a tree view).

To solve this problem, all of the methods in the class a.a.k.p can be inspected with backtrace at the same time (the output stack traces can be mixed up if multiple methods are called simultaneously, in that case, the inspected methods can be split into multiple smaller parts). The output of the stack trace should look very similar to the output in listing 31 (lines 7-9 should be the same). This is because it can be assumed that only one method "a.a.k.p.a" is called in the onClick event (note the semicolon and the number in the parentheses on line 7, this is line number of the method in the code, without the code, this is not very helpful, but can be used to differentiate different methods in the same class).

So the next step is to inspect (with backtrace) all of the methods with the name "a" inside the "a.a.k.p" class. Note that the "Inspect with backtrace" is not just a stack trace print. It is a hook that prints out the method name (and with the proposed changes specified in listing 30 also method signature) and also return value. The stack trace is just a bonus. This allows the linking of the stack trace to the unique function (as the signature of the function is unique). After hooking all of the

functions and clicking the login button again, multiple stack traces are observed. Only one is "valid" in terms of the assumptions made (that it will be similar with the one in listing 31). This output can be seen in listing 32. In the stack trace, it can be seen that the call to the method a.a.k.p.a originated from the onClick event (as expected) but now the trace is linked to specific method signature (line 3 or 14). The signature is e.e0 a.a.k.p.a(java.util.Map), so the return value is of type "e.e0" and there is one input parameter of type java.util.Map. At this point, it is very likely that the correct method was found.

```
1  *** entered a.a.k.p.a
2  Parameters:
3          arg[0]: [object Object] Signature: e.e0 a.a.k.p.a(java.util.Map)
4  Backtrace:
5          dalvik.system.VMStack.getThreadStackTrace(Native Method)
6          java.lang.Thread.getStackTrace(Thread.java:1720)
7          a.a.k.p.a(Native Method)
8          cz.vutbr.pavol.obfuscation.experimental.main.LibApplication.i(Native
           ↪  Method)
9          cz.vutbr.pavol.obfuscation.experimental.main.MainActivity$a.onClick(
           ↪  Unknown Source:66)
10         android.view.View.performClick(View.java:7125)
11     ...
12 *** exiting a.a.k.p.a
13 Return value:
14         retval(e.e0 a.a.k.p.a(java.util.Map)): e.u@a16a887
```

Listing 32: DexProtector – a.a.k.p.a method trace.

Both the DexProtector and the DashO use R8 for optimizations and name obfuscation. This explains why the return type of the signature generation function is "e.e0" (in case of DashO it was "e.b0"). Both classes are the same, which means that R8 is responsible for this obfuscation in both cases. DexProtector also returns object of type "e.u", because "e.e0" is an abstract class and the "e.u" is the actual implementation. Once the correct method was found, it can be hooked. The hook itself is very similar to the hook used in DashO (listing 29); the only changes are the class/function names. The excerpt of the hook can be seen in listing 33.

6.4 Summary

This chapter provided research on a practical approach to certain obfuscation techniques. The results show that no protection is absolute. A quote from Ralph Waldo

```
var SignClass = Java.use("a.a.k.p");
var eu = Java.use("e.u");
var tmp = SignClass.a.overload("java.util.Map").call(SignClass, map);
var output = Java.cast(tmp, eu);
var str = "";
for (var index = 0; index < output._b.value.size(); ++index) {
    str += output._b.value.get(index) + "=" + output.c.value.get(index) + "&";
}
```

Listing 33: DexProtector signature generation method hook excerpt.

Emerson can be used here: "It is not the destination, it is the journey." No security defences will protect any application completely. The job of the obfuscation is to make it as hard and as time-consuming as possible.

The string encryption by itself is not very effective and can easily be located and bypassed using dynamic analysis. On the other hand, flow obfuscation is an effective measure. The decompiled code is very hard to follow and comprehend. However, flow obfuscation does not affect dynamic analysis. Class encryption proved to be very effective but still, it is possible to utilise dynamic analysis. In practice, the more obfuscation techniques are employed, the harder it is to analyse the application. The individual techniques are not very effective, but when multiple techniques are combined, the resulting application is hard to analyse and the process is also time-consuming, which is exactly the goal of the obfuscation.

There are other obfuscation techniques, for example, hook detection, which *should* prevent tools like Frida from working. Both the DexProtector and DashO support this feature. This could, in theory, prevent dynamic analysis completely. Hook detection was not in the scope of this thesis because attempting to bypass this feature would require the entire thesis to specialize for this purpose only (similar to class encryption, both topics are complicated and extremely time-consuming).

Both the DexProtector and DashO use the default R8 for name obfuscation and optimization. This may be a potential negative as the R8 is the default obfuscator/optimizer. This means that penetration testers will be familiar with it and static analysis can be faster than using some custom tool. With the usage of additional controls like flow obfuscation, the downside of R8 usage is minimal (but still present).

Conclusion

The theoretical part of this thesis is about obfuscation, techniques used by the obfuscators and provides the necessary information to understand the obfuscation. Two popular obfuscation tools are presented: ProGuard and DexGuard. Also, the impact of the obfuscation on security testing is presented.

The thesis continues with the chapter about effective strategies to analyse the application. These include static and dynamic analysis. As the obfuscation targets the source code, the dynamic analysis proves to be more effective as it is not impacted by the obfuscation as much as the static analysis is. The main standard in mobile application security, the OWASP Mobile Security Testing Guide is mentioned to provide basic information about security testing of Android applications.

The following chapter presents the self-protection mechanisms of the applications. Android developers use these protections in order to better protect their applications against threats like running on a rooted device or inside an emulator. These mechanisms act as an obstacle during security testing and must be bypassed.

The next chapter describes methods to bypass the self-protection mechanisms of the application using dynamic instrumentation. As this thesis showed, dynamic instrumentation (and especially Frida toolkit) can be very effective at removing these protections. Almost any protection mechanism can be removed or rendered useless using this technique.

The last two chapters are about advanced obfuscation techniques. A specially designed application was used to analyse three obfuscators: Allatori, DashO and DexProtector. The main focus was on string encryption as it can be the entry gate during penetration test but multiple obfuscation techniques were analysed. Results proved that dynamic analysis could bypass all techniques, which were in the scope of this thesis. This also applies, when class encryption is used and "blind" dynamic analysis is performed. However, with a more complex application, "blind" analysis could be impossible or time-consuming (but is still an option). On the other hand, if multiple obfuscation techniques would be used together, the resulting application would still be very time-consuming to analyse. This thesis showed that dynamic analysis is very effective against obfuscated applications.

The unobfuscated application should always be provided for the penetration test. Bypassing all RASP defences and obfuscation techniques is a time-consuming process and will make the penetration test more expensive or less reliable. Some parts of the application could be untested as the obfuscation would be impossible to bypass in the given timeframe or certain functionality could be missed altogether.

Bibliography

[1] HAMILTON, James a Sebastian DANICIC, 2009. An Evaluation of Current Java Bytecode Decompilers. *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation* [online]. IEEE, 2009, 129-136 [cit. 2019-10-20]. DOI: 10.1109/SCAM.2009.24. ISBN 978-0-7695-3793-1. Available at: http://ieeexplore.ieee.org/document/5279917/

[2] WANG, Yan a Atanas ROUNTEV, 2017. Who Changed You? Obfuscator Identification for Android. *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)* [online]. IEEE, 2017, 154-164 [cit. 2019-10-20]. DOI: 10.1109/MOBILESoft.2017.18. ISBN 978-1-5386-2669-6. 1710.01139. Available at: http://ieeexplore.ieee.org/document/7972730/

[3] *Android Runtime (ART) and Dalvik* [online], [cit. 2019-10-27]. Available at: https://source.android.com/devices/tech/dalvik

[4] *ProGuard and R8: a comparison of optimizers* [online], [cit. 2019-10-27]. Available at: https://www.guardsquare.com/en/blog/proguard-and-r8

[5] *ProGuard manual* [online], [cit. 2019-10-27]. Available at: https://www.guardsquare.com/en/products/proguard/manual

[6] *DexGuard vs. ProGuard* [online], [cit. 2019-10-27]. Available at: https://www.guardsquare.com/en/blog/dexguard-vs-proguard

[7] FARUKI, Parvez, Ammar BHARMAL, Vijay LAXMI, Vijay GANMOOR, Manoj Singh GAUR, Mauro CONTI a Muttukrishnan RAJARAJAN, 2015. *Android Security: A Survey of Issues, Malware Penetration, and Defenses* [online]. **17**(2), 998-1022 [cit. 2019-10-20]. DOI: 10.1109/COMST.2014.2386139. ISSN 1553-877X. Available at: https://ieeexplore.ieee.org/document/6999911/

[8] WERMKE, Dominik, Nicolas HUAMAN, Yasemin ACAR, Bradley REAVES, Patrick TRAYNOR a Sascha FAHL, 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In: *Proceedings of the 34th Annual Computer Security Applications Conference on - ACSAC '18* [online]. New York, New York, USA: ACM Press, 2018, s. 222-235 [cit. 2019-11-04]. DOI: 10.1145/3274694.3274726. ISBN 9781450365697. Available at: http://dl.acm.org/citation.cfm?doid=3274694.3274726

[9] DONG, Shuaike, Menghao LI, Wenrui DIAO, et al., 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the

Wild. *Security and Privacy in Communication Networks* [online]. Cham: Springer International Publishing, 2018-12-29, 172-192 [cit. 2019-10-20]. DOI: 10.1007/978-3-030-01701-9_10. ISBN 978-3-030-01700-2. Available at: http://link.springer.com/10.1007/978-3-030-01701-9_10

[10] SCHLEIER, Sven, Bernhard MUELLER a Jeroen WILLEMSEN, 2019. *OWASP Mobile Security Testing Guide* [online]. [cit. 2019-08-09]. Available at: https://github.com/OWASP/owasp-mstg/releases/download/1.1.3/MSTG-EN.pdf

[11] HAUPERT, Vincent, Dominik MAIER, Nicolas SCHNEIDER, Julian KIRSCH a Tilo MÜLLER, 2018. Honey, I Shrunk Your App Security: The State of Android App Hardening. *Detection of Intrusions and Malware, and Vulnerability Assessment* [online]. Cham: Springer International Publishing, 2018-06-08, 69-91 [cit. 2019-12-01]. Lecture Notes in Computer Science. DOI: 10.1007/978-3-319-93411-2_4. ISBN 978-3-319-93410-5. Available at: http://link.springer.com/10.1007/978-3-319-93411-2_4

[12] Network security configuration, *Android Developers* [online]. [cit. 2019-12-01]. Available at: https://developer.android.com/training/articles/security-config

[13] *SafetyNet Attestation API* [online], [cit. 2019-12-05]. Available at: https://developer.android.com/training/safetynet/attestation

[14] VILLA, Adrian, 2017. *Bypassing Android-s Network Security Configuration* [online]. [cit. 2019-12-17]. Available at: https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2017/november/bypassing-androids-network-security-configuration/

[15] *Allatori FEATURES* [online], [cit. 2020-03-03]. Available at: http://www.allatori.com/features.html

[16] *DexProtector User Manual* [online], [cit. 2020-03-27]. Available at: https://dexprotector.com/docs

[17] *PreEmptive Protection DashO for Android & Java* [online], [cit. 2020-03-27]. Available at: https://www.preemptive.com/dasho/pro/userguide/en/index.html

[18] YANG, Wenbo, Yuanyuan ZHANG, Juanru LI, Junliang SHU, Bodong LI, Wenjun HU a Dawu GU, 2015. AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware. In: *Research in Attacks, Intrusions, and Defenses*. Cham: Springer International Publishing, 2015-12-12. Lecture

Notes in Computer Science. DOI: 10.1007/978-3-319-26362-5_17. ISBN 978-3-319-26361-8. Available at: http://link.springer.com/10.1007/978-3-319-26362-5_17

[19] ZHANG, Yueqian, Xiapu LUO a Haoyang YIN, 2015. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In: *Computer Security – ESORICS 2015*. Cham: Springer International Publishing, 2015-11-18. Lecture Notes in Computer Science. DOI: 10.1007/978-3-319-24177-7_15. ISBN 978-3-319-24176-0. Available at: http://link.springer.com/10.1007/978-3-319-24177-7_15

[20] CAN, Ahmet, *N Ways to Unpack Mobile Malware* [online]. March 13, 2019 [cit. 2020-04-14]. Available at: https://pentest.blog/n-ways-to-unpack-mobile-malware/

List of appendices

A TrustManager Bypass

```
var CertificateFactory = Java.use("java.security.cert.CertificateFactory");
var FileInputStream = Java.use("java.io.FileInputStream");
var BufferedInputStream = Java.use("java.io.BufferedInputStream");
var KeyStore = Java.use("java.security.KeyStore");
var TrustManagerFactory = Java.use("javax.net.ssl.TrustManagerFactory");
var SSLContext = Java.use("javax.net.ssl.SSLContext");
var certificateFactory = CertificateFactory.getInstance("X.509");
try {
    var fileInputStream = FileInputStream.$new("/data/local/tmp/cert-der.crt");
} catch (error) {
    console.log("[ERROR] " + error);
}
var bufferedInputStream = BufferedInputStream.$new(fileInputStream);
var ca = certificateFactory.generateCertificate(bufferedInputStream);
bufferedInputStream.close();
var keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", ca);
var trustManagerFactory =
↪  TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
trustManagerFactory.init(keyStore);
//script will wait untill the application calls the SSLContext init method
SSLContext.init.overload("[Ljavax.net.ssl.KeyManager;",
↪  "[Ljavax.net.ssl.TrustManager;",
↪  "java.security.SecureRandom").implementation = function (keyManager,
↪  trustManager, secureRandom) {
        //TrustManager is swapped for the previously created
        SSLContext.init.overload("[Ljavax.net.ssl.KeyManager;",
        ↪  "[Ljavax.net.ssl.TrustManager;",
        ↪  "java.security.SecureRandom").call(this, keyManager,
        ↪  trustManagerFactory.getTrustManagers(), secureRandom);
}
```

Listing 34: Frida script to bypass TrustManager SSL pinning.

B Signature Generation

```
1  public static Map<String, String> generateSignature(Map<String, String> inputs)
   ↪  {
2      String nonce = Util.getNonce();
3      String timestamp = String.valueOf(System.currentTimeMillis());
4      inputs.put("nonce", nonce);
5      inputs.put("timestamp", timestamp);
6      String iv = "Lorem ipsum dolor sit amet";
7      boolean first = true;
8      String encoded = "";
9      for (final Map.Entry<String, String> entrySet : inputs.entrySet()) {
10         if (first) {
11             encoded = iv + entrySet.getKey() + entrySet.getValue();
12             first = false;
13             continue;
14         }
15         encoded += entrySet.getKey() + entrySet.getValue();
16     }
17     try {
18         final String hashingAlgorithm = "HmacSHA256"; //or "HmacSHA1",
           ↪  "HmacSHA512"
19         byte[] bytes = hmac(hashingAlgorithm, (iv + timestamp +
           ↪  nonce).getBytes(), encoded.getBytes());
20         final String messageDigest = bytesToHex(bytes);
21         inputs.put("signature", messageDigest);
22         return inputs;
23      } catch (Exception e) {
24         e.printStackTrace();
25         return null;
26      }
27 }
```

Listing 35: Signature generation method.

C Allatori Configuration

```
<config>
    <input>
        <dir in="${classesRoot}" out="${classesRoot}-obfuscated"/>
    </input>
    <classpath>
        <jar name="${androidJar}"/>
        <jar name="${classpathJars}"/>
    </classpath>
    <keep-names>
        <class template="public class * instanceof android.app.Activity">
            <method template="public void *(android.view.View)"/>
        </class>
        <class template="public class * instanceof android.view.View">
            <method template="void set*(**)"/>
            <method template="get*()"/>
        </class>
        <class template="class * extends java.lang.Enum">
            <method template="values()"/>
            <method template="valueOf(java.lang.String)"/>
        </class>
        <class template="class * implements android.os.Parcelable">
            <field template="public static final android.os.Parcelable*Creator
            ↪  *"/>
        </class>
        <class template="class *.R*">
            <field template="public static *"/>
        </class>
        <class template="public class * instanceof android.app.Application"/>
        <class template="public class * instanceof android.app.Service"/>
        <class template="public class * instanceof
        ↪  android.content.BroadcastReceiver"/>
        <class template="public class * instanceof
        ↪  android.content.ContentProvider"/>
        <class template="public class * instanceof
        ↪  android.app.backup.BackupAgentHelper"/>
        <class template="public class * instanceof
        ↪  android.preference.Preference"/>
        <class template="public class
        ↪  com.android.vending.licensing.ILicensingService"/>
        <class template="public class
        ↪  com.google.android.vending.licensing.ILicensingService"/>
    </keep-names>
    <property name="string-encryption" value="enable"/>
```

```
    <property name="string-encryption-type" value="fast"/>
    <property name="extensive-flow-obfuscation" value="maximum"/>
    <property name="log-file" value="${logFile}"/>
</config>
```

Listing 36: Allatori Configuration.

D DashO String Decryption

```
public static String a(int i, int i2, String str) {
    int i3;
    String str2;
    char[] cArr;
    int i4;
    int i5;
    int i6;
    int i7;
    String str3 = "0";
    int i8 = 5;
    int i9 = 4;
    if (Integer.parseInt(str3) != 0) {
        str2 = str3;
        i3 = 4;
    } else {
        str2 = "32";
        i3 = 5;
        i8 = 8;
    }
    char[] cArr2 = null;
    int i10 = 1;
    if (i8 != 0) {
        i -= 3;
        cArr = str.toCharArray();
        i5 = 0;
        i4 = 0;
    } else {
        i4 = i8 + 8;
        str3 = str2;
        cArr = null;
        i5 = 1;
    }
    if (Integer.parseInt(str3) != 0) {
        i6 = i4 + 12;
        i7 = 1;
    } else {
        i7 = cArr.length;
        i6 = i4 + 10;
    }
    if (i6 != 0) {
        i9 = 4 << i3;
        cArr2 = cArr;
        i10 = i5;
```

```
    }
    char c2 = (i9 - 1) ^ ' ';
    while (i10 != i7) {
        char c3 = cArr2[i10] ^ (i & c2);
        i += i2;
        cArr2[i10] = (char) c3;
        i10++;
    }
    return String.valueOf(cArr2, 0, i7).intern();
}
```

Listing 37: DashO string decryption method.

E DexProtector Configuration

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<dexprotector>
    <verbose>true</verbose>
    <optimize>false</optimize>
    <signMode>release</signMode>
    <v1SigningEnabled>true</v1SigningEnabled>
    <v2SigningEnabled>false</v2SigningEnabled>
    <stringEncryption mode="all">
        <filters>
            <filter>
                <![CDATA[glob:!android/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!com/crashlytics/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!com/facebook/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!com/google/commons/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!com/activeandroid/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!com/twitter/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!com/flurry/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!org/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!java/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!javax/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!rx/**]]>
            </filter>
            <filter>
```

```
            <![CDATA[glob:!retrofit/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!io/fabric/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!twitter4j/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/newrelic/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/squareup/**]]>
        </filter>
    </filters>
</stringEncryption>
<classEncryption mode="all">
    <filters>
        <filter>
            <![CDATA[glob:!android/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/crashlytics/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/facebook/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/google/commons/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/activeandroid/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/twitter/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/flurry/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!org/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!java/**]]>
        </filter>
        <filter>
```

```
            <![CDATA[glob:!javax/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!rx/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!retrofit/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!io/fabric/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!twitter4j/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/newrelic/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/squareup/**]]>
        </filter>
    </filters>
</classEncryption>
<hideAccess mode="all">
    <filters>
        <filter>
            <![CDATA[glob:!android/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/crashlytics/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/facebook/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/google/commons/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/activeandroid/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/twitter/**]]>
        </filter>
        <filter>
            <![CDATA[glob:!com/flurry/**]]>
        </filter>
        <filter>
```

```
                <![CDATA[glob:!org/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!java/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!javax/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!rx/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!retrofit/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!io/fabric/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!twitter4j/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!com/newrelic/**]]>
            </filter>
            <filter>
                <![CDATA[glob:!com/squareup/**]]>
            </filter>
        </filters>
    </hideAccess>
    <resourceEncryption mode="all">
        <assets mode="all"/>
    </resourceEncryption>
    <integrityControl mode="all">
        <checkCertificate>on</checkCertificate>
    </integrityControl>
    <antiDebug>true</antiDebug>
</dexprotector>
```

Listing 38: DexProtector configuration.